Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Université de Neuchâtel unine

iCoSys

Institute of Complex Systems

Parallel/Distributed systems

and High Performance Computing

Professeur:
Pierre Kuonen (HEIA-FR)

Notice to the audience

From the source... means: Taken from other sources

Context

- The terms "parallelism", "parallel or distributed or concurrent programming"... are used in different contexts leading to different interpretations.
- Indeed, we can cite six main reasons for "parallelising" an application
  - Fault tolerance : *Avoid system failures due to design or implementation errors*
    - Tolérance aux fautes
  - Resistance to failures : *Avoid system failures due to unpredictable hardware failures (breakdowns)*
    - Résistance aux pannes
  - The power of expression : *Freeing oneself from the obligation to express things sequentially*
    - Puissance d'expression
  - Inherently parallel problems : *The problem to solve is physically parallel (e.g.: web app.)*
    - Problème intrinsèquement parallèle
  - Computer performance : *Get the best possible computing performance*
    - Puissance de calcul
  - The size of the problem : *Solve larger problems*
    - Taille du problème

Purpose of this presentation

A bit of vocabulary (my own)

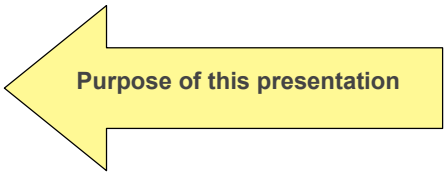
- Concurrent programming
  - The term "concurrent" is used to indicate that there is no imposed order in the execution of some instruction sequences or tasks.
  - They can be executed in sequence, in time sharing, or simultaneously.
  - The problems to be solved deal mainly with synchronization and mutual exclusion (race condition).
- Parallel programming
  - Often confused with concurrent programming
  - Some instructions or tasks must be executed simultaneously
  - Compared to concurrent programming, the problem of load balancing must be taken into account.
- Distributed programming
  - The interaction between tasks executed simultaneously is not only through variables but also by sending messages or information.
  - Compared to parallel programming, the problem of communications has to be added.
- Distributed programming : variant (in French "programmation répartie")
  - I reserve this term for the use of redundancy in the case of fault tolerant systems.
  - This is a very interesting problem in the world of parallel systems but we will not deal with it in this course.

Scientific Computing and HPC

- Computers have originally been conceived to achieve scientific computing
  - Physic simulation (Fundamental physic, military, ...)
- In this context the main (only ?) issue is the computing performance
  - High Performance Computing (HPC)
- There are three ways to get performance (power)
  - To work harder -> more powerful processing unit (CPU)
  - To work smarter -> better algorithms
  - **To get help -> to parallelize / to distribute**

From the source...

Pfister in search of clusters,1998



How to measure the performance

- By analysing the computer structure
  - We determine the characteristics of the Hardware such as clock frequency, memory size, bus bandwidth, number of cores etc.. and we deduce (compute) the possible performance of the system

- By using benchmark applications
  - We run specific applications, measure the computing time and then calculate the performance achieved
  - Benchmark applications
    - Linpack: the historical one and the most used
    - HPCG: High Performance Conjugate Gradient
    - Graph 500

Peak or Sustained performance

- Peak performance obtained by analysing the computer structure
  - Official definition:
    The maximum computing performance of computer

    My definition:

    The computing power that the constructor guarantees that you will never get !

- Sustained performance obtained using benchmarks
  - The observed performance on some benchmark programs
    - Usually (up to now) LINPACK

Performance measurement unit

- **Flops** : FLoating-point Operation per Second.
  - 1 Flops = one floating point operation in one second
    - Mflops = Megaflops
      - 10^6 floating point operation in one second
    - Gflops = Gigaflops
      - 10^9 floating point operation in one second
    - Tflops = Teraflops
      - 10^{12} floating point operation in one second
    - Pflops = Petaflops
      - 10^{15} floating point operation in one second
    - Eflops =Exaflops
      - 10^{18} floating point operation in one second

415.53 Pflops **sustained**

513.85 Pflops **peak**

  - MIPS : Million of Instructions Per Second
    - 1 MIPS = one million of instructions in one second
      - "Meaningless Indication of Processor Speed"

LINPACK

- LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems
  - The LINPACK Benchmark was introduced by Jack Dongarra during 70's, mainly for vector computers
  - To solve a dense system of linear equations
    - Since the problem is very regular, the performance achieved is quite high, and the performance numbers give a good correction of peak performance.

- HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers
  - The HPL software package requires the availability on your system of an implementation of the Message Passing Interface
    MPI (1.1 compliant)

The TOP 500 list (www.top500.org)

- The TOP500 table shows the 500 most powerful commercially available computer systems known to us.
  Part information indicated are:
  - **Rank** - Position within the TOP500 ranking
  - **Manufacturer** - Manufacturer or vendor
  - **Computer/Year Vendor**- self explained
  - **Site** – Customer
  - **Cores** - Number of cores
  - **Rmax** - Maximal LINPACK performance achieved
  - **Rpeak** - Theoretical peak performance
  - **Power** - Electric power consumption in KWatt

From the source...

http://www.top500.org/

Current list: June 2020

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

R_{peak} values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
7	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	34,568.6	1,344
8	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas	448,448	23,516.4	38,745.9	

From the source...

https://www.top500.org/lists/top500/2020/06/

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

https://www.cscs.ch/

10	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384

The most powerful !

SUPERCOMPUTER FUGAKU - SUPERCOMPUTER FUGAKU, A64FX 48C 2.2GHZ, TOFU INTERCONNECT D

Site:	RIKEN Center for Computational Science
System URL:	https://www.r-ccs.riken.jp/en/fugaku/project
Manufacturer:	Fujitsu
Cores:	7,299,072
Memory:	4,866,048 GB
Processor:	A64FX 48C 2.2GHz
Interconnect:	Tofu interconnect D
Performance	
Linpack Performance (Rmax)	415,530 TFlop/s
Theoretical Peak (Rpeak)	513,855 TFlop/s
Nmax	20,459,520
HPCG [TFlop/s]	13,366.4

Power Consumption	
Power:	28,334.50 kW (Submitted)
Power Measurement Level:	2
Measured Cores:	995,328
Software	
Operating System:	Red Hat Enterprise Linux
Compiler:	FUJITSU Software Technical Computing Suite V4.0
Math Library:	FUJITSU Software Technical Computing Suite V4.0
MPI:	FUJITSU Software Technical Computing Suite V4.0

No.2: Summit 2,925.75

The Fugaku supercomputer

Name: fugaku

Developed by: RIKEN and fujitsu

From the source...

https://www.designboom.com/technology/japans-fugaku-fastest-supercomputer-in-the-world-06-23-2020/

Installed at the RIKEN center for computational science (R – CCS) in kobe, japan, the fugaku system marks the successor to the K computer, which was previously crowned the world's fastest back in 2011.

Set to be fully operational by april 2021, the development of fugaku is part of a national plan to help address social and scientific issues, including finding a treatment for COVID-19

In addition to taking the top spot on the TOP500, fugaku has also been crowned number one on both the HPCG and graph500, marking the first time in history a supercomputer has taken all three awards at the same time

HPCG: High Performance Conjugate Gradient

HPCG Benchmark

The High Performance Conjugate Gradients (HPCG) Benchmark project is an effort to create a new metric for ranking HPC systems. HPCG is intended as a complement to the High Performance LINPACK (HPL) benchmark, currently used to rank the TOP500 computing systems. The computational and data access patterns of HPL are still representative of some important scalable applications, but not all. HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

From the source...

https://www.hpcg-benchmark.org/

HPCG is a complete, stand-alone code that measures the performance of basic operations in a unified code:

- Sparse matrix-vector multiplication.
- Vector updates.
- Global dot products.
- Local symmetric Gauss-Seidel smoother.
- Sparse triangular solve (as part of the Gauss-Seidel smoother).
- Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids.
- Reference implementation is written in C++ with MPI and OpenMP support.

Graph 500 Benchmark

1 Brief Description of the Graph 500 Benchmark

Data-intensive supercomputer applications are an increasingly important workload, but are ill-suited for platforms designed for 3D physics simulations. Application performance cannot be improved without a meaningful benchmark. Graphs are a core part of most analytics workloads. Backed by a steering committee of over 30 international HPC experts from academia, industry, and national laboratories, this specification establishes a large-scale benchmark for these applications. It will offer a forum for the community and provide a rallying point for data-intensive supercomputing problems. This is the first serious approach to augment the Top 500 with data-intensive applications.

The intent of benchmark problems ("Search" and "Shortest-Path") is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, undirected graph. In addition to a kernel to construct the graph from the input tuple list, there are two additional computational kernels to operate on the graph.

This benchmark includes a scalable data generator which produces edge tuples containing the start vertex and end vertex for each edge. The first kernel constructs an *undirected* graph in a format usable by all subsequent kernels. No subsequent modifications are permitted to benefit specific kernels. The second kernel performs a breadth-first search of the graph. The third kernel performs multiple single-source shortest path computations on the graph. All three kernels are timed.

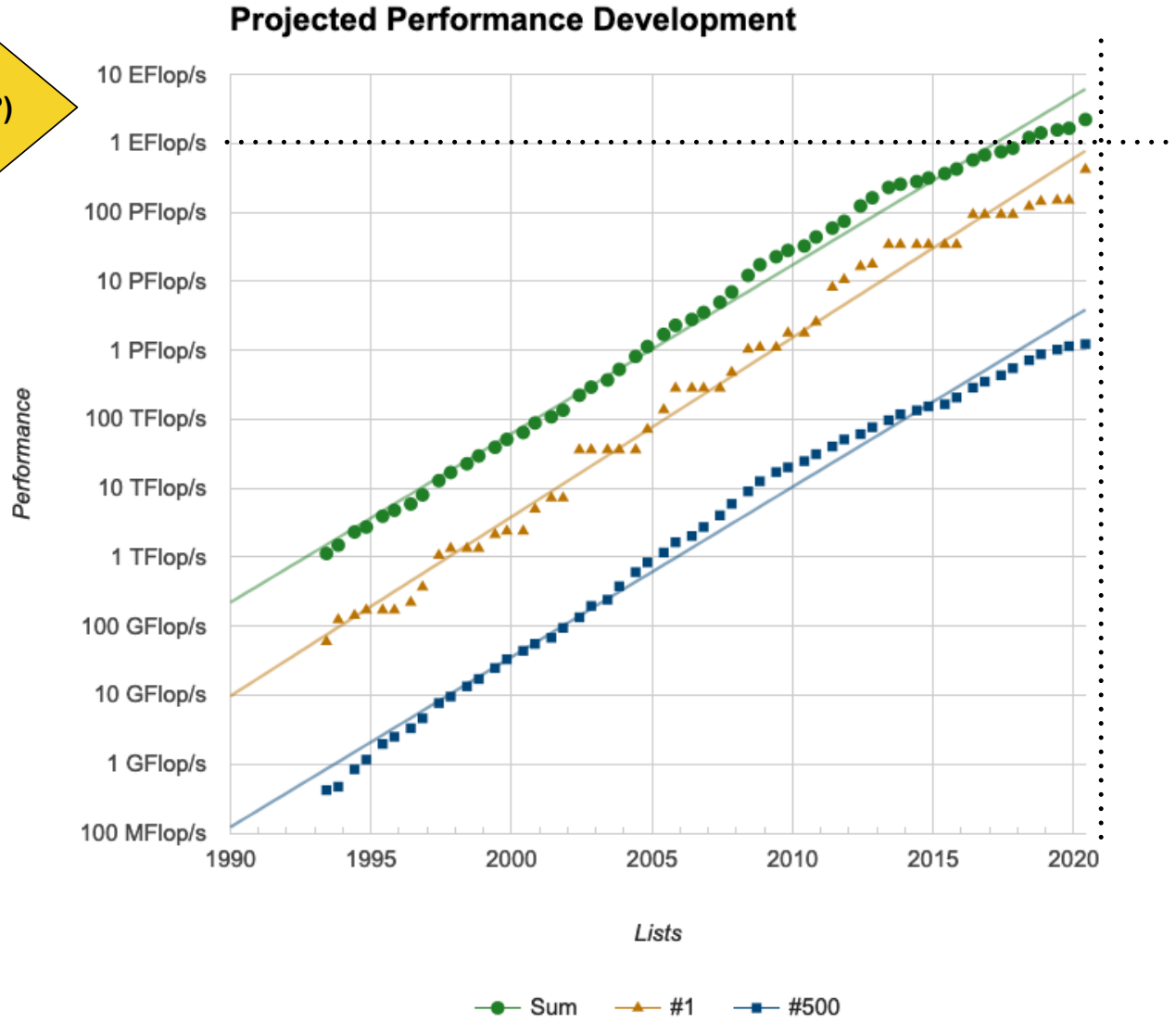
GRAPH 500

From the source...

https://graph500.org/

Projection

10^{18} Flops by 2021 (??)



Green 500 list: June 2020

Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	393	**MN-3** - MN-Core Server, Xeon 8260M 24C 2.4GHz, MN-Core, RoCEv2/MN-Core DirectConnect, Preferred Networks Preferred Networks Japan	2,080	1,621.1	77	21.108
2	7	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	1,344	20.518
3	468	**NA-1** - ZettaScaler-2.2, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz, PEZY Computing / Exascaler Inc. PEZY Computing K.K. Japan	1,271,040	1,303.2	80	18.433
4	204	**A64FX prototype** - Fujitsu A64FX, Fujitsu A64FX 48C 2GHz, Tofu interconnect D, Fujitsu Fujitsu Numazu Plant Japan	36,864	1,999.5	118	16.876
5	26	**AiMOS** - IBM Power System AC922, IBM POWER9 20C 3.45GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM Rensselaer Polytechnic Institute Center for Computational Innovations (CCI) United States	130,000	8,339.0	512	16.285
8	2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	10,096	14.719
9	1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	28,335	14.665
10	9	**Marconi-100** - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	1,476	14.661

From the source...

https://www.top500.org/lists/green500/2020/06/

A small exercise

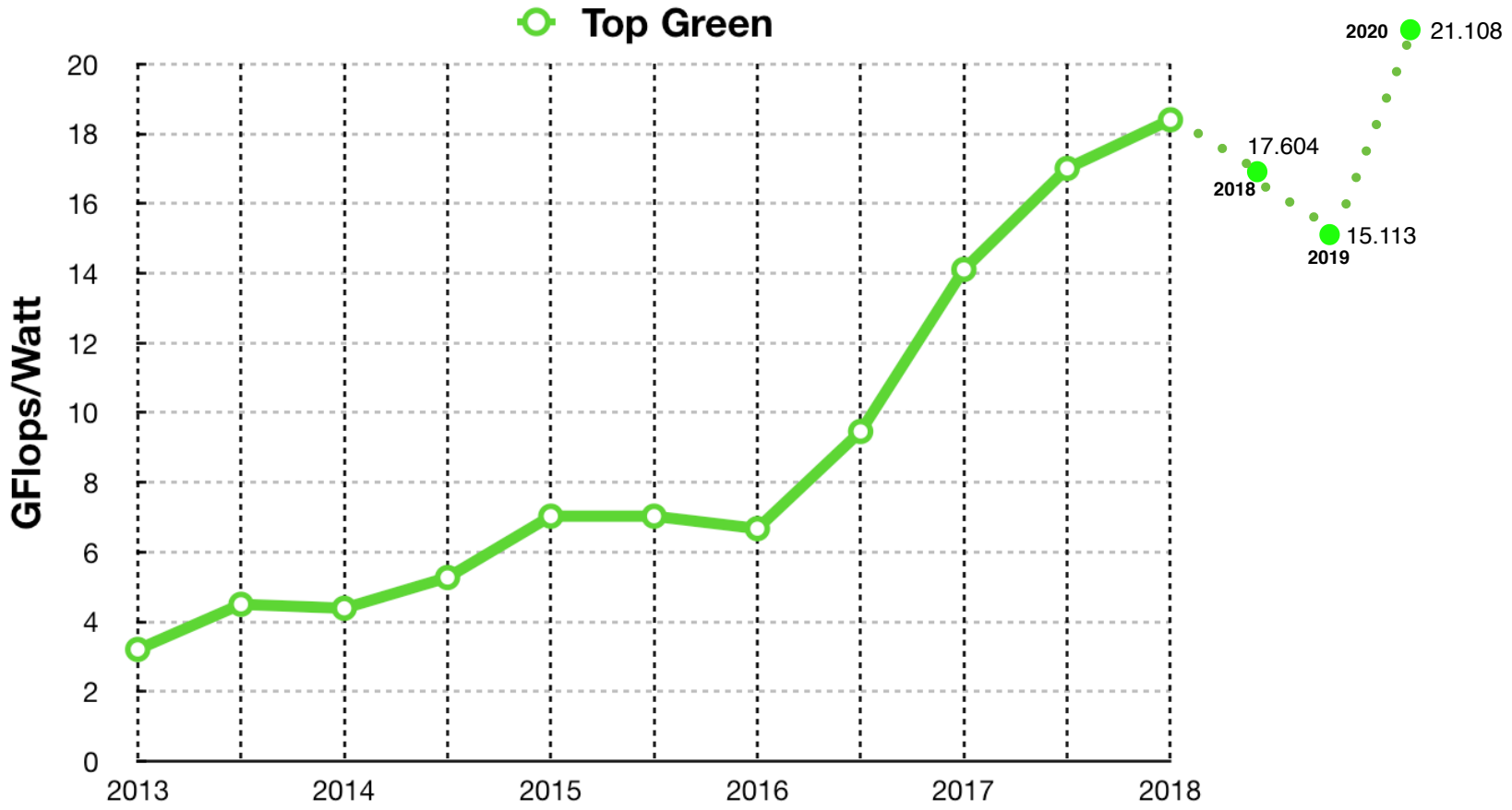
- Based on this data, make an estimate of the electrical power required to operate a supercomputer with a power of 1 ExaFlops.
  - Le No1 de la liste Green 500 : 21.108 GFlops/watt
  - Le No1 de la liste Top 500 : 14.665 GFlops/watt

The **Grande Dixence Dam** is a concrete gravity dam on the Dixence at the head of the Val d'Hérémence in the canton of Valais in Switzerland. At 285m high, it is the tallest gravity dam in the world, fifth tallest dam overall, and the tallest dam in Europe. It is part of the Cleuson-Dixence Complex. With the primary purpose of hydroelectric power generation. The dam supplies several power stations with a total installed capacity of 2,069 MW, which is enough to supply 400,000 Swiss households.

From the source...
Wikipedia

The The Grande Dixence dam mainly supplies three hydroelectric plants:
- Fionnay Power Station: 290 MW. After arriving at the Fionnay Power Station from the Grande Dixence Dam, water then travels through a 16 km (10 mi) pressure tunnel which eventually leads to the
- Nendaz Power Station: **390 MW**
- Bieudron power station: 1200 MW
World's highest waterfall (1,880 m)

TOP Green : Evolution



HPC and Web

The CERN data centre in 2010 housing some WWW servers

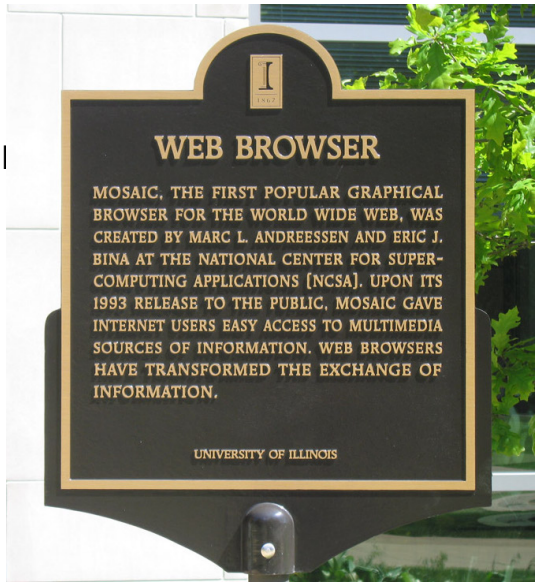
- It is interesting to note that the most important pieces of the world wide web have been realized in institutions where HPC is the major activity
  - The concept of World Wide Web has been invented and implemented at **CERN**
    - English scientist Tim Berners-Lee invented the World Wide Web in 1989. He wrote the first web browser computer program in 1990 while employed at CERN in Switzerland.
  - Mosaic, is an early web browser that has been credited with popularizing the Web. It was developed at **National Center for Supercomputing Applications** (NCSA)
    - Despite that Mosaic was not the very first Web browser, Its intuitive interface, reliability, Windows port and simple installation all contributed to its popularity within the web, as well as on Microsoft operating systems
    - In 1993, Mosaic v 1.0 broke away from the small pack of existing browsers by including features—like icons, bookmarks, a more attractive interface, and pictures—that made the software easy to use and appealing to “non-geeks."
    - It is the precursor of the Netscape Navigator

From the source...
http://www.ncsa.illinois.edu/enabling/mosaic
Wikipedia

HPC and Web

- It is interesting to note that the most important pieces of the world wide web have been realized in institutions where HPC is the major activity
  - The concept of World Wide Web has been invented and implemented at **CERN**
    - English scientist Tim Berners-Lee invented the World Wide Web in 1989. He wrote the first web browser computer program in 1990 while employed at CERN in Switzerland.
  - Mosaic, is an early web browser that has been credited with popularizing the Web. It was developed at **National Center for Supe** (NCSA)
    - Despite that Mosaic was not the very first Web browser, Its intuitive interface, reliability, Windows port and simple installation all contributed to its popularity within the web, as well as on Microsoft operating systems
    - In 1993, Mosaic v 1.0 broke away from the small pack of existing browsers by including features—like icons, bookmarks, a more attractive interface, and pictures—that made the software easy to use and appealing to “non-geeks."
    - It is the precursor of the Netscape Navigator

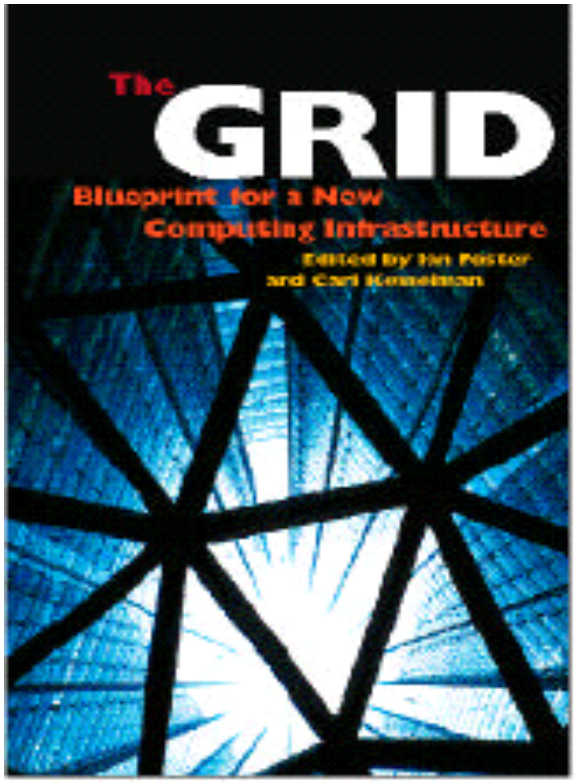
The CERN data centre in 2010 housing some WWW servers



From the source...
http://www.ncsa.illinois.edu/enabling/mosaic
Wikipedia

An alternative : the GRID

- Idea:
  - To use the unused computer cycles spread over the world
- The origin
  - I. Foster and C. Kesselman (Eds.).
    "The GRID: Blueprint for a New Computing Infrastructure",
    Morgan Kaufman Publishers, San Francisco, 1999
    - « We probably see the spread of "computer utilities", which, like present electric and telephone utilities, service individual homes and offices across the country (…) A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities »



- The SETI@home project
  - SETI@home is a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI).
  - You can participate by running a free program that downloads and analyzes radio telescope data.
  - http://tharkun.free.fr/SETI/explication.htm
- History
  - Original idea in 1994, team formation in 1996
  - SETI@home server started in May 1999
  - 15 Decembre 2005 they turn off the SETI@home server
  - A new version of the project is continuing : SETI@home/BOINC
    - http://boinc.berkeley.edu/
- Results
  - We did not find Extraterrestrial Intelligence
  - 2 millions of accumulated computing years
  - Popularized the concept of meta computing

From the source...

SETI HOME

The World's Largest and most Sensitive Radiotelescope located in Arecibo, Puerto Rico

BOINC

https://boinc.berkeley.edu/

- BOINC is the successor of SETI@home
  - BOINC lets you help cutting-edge science research using your computer. The BOINC app, running on your computer, downloads scientific computing jobs and runs them invisibly in the background. It's easy and safe.
    - many different programs

Name Mouse over for details; click to visit web site	Category	Area	Sponsor	Supported platforms
Acoustics@home	Physical Science	Physics, Underwater acoustics	V.I. Ilichev Pacific Oceanological Institute, Far Eastern Federal University, Matrosov Institute for System Dynamics and Control Theory, A.A. Kharkevich Institute for Information Transmission Problems, Dorodnicyn Computing Centre	Details
Asteroids@home	Physical Science	Astrophysics		Details
CAS@home	Multiple applications	Physics, biochemist others		Details
Citizen Science Grid	Multiple applications	Molecular biology, Computer Science		Details
Climateprediction.net	Earth Sciences	Climate study		Details
Collatz Conjecture	Mathematics, computing, and games	Mathematics	Independent	Details
Cosmology@Home	Physical Science	Astronomy	University of Illinois at Urbana-Champaign	Details
DENIS@Home	Biology and Medicine	Medical physiology	San Jorge University, Zaragoza, Spain	

BOINC computing power

Totals

24-hour average: 27.635 PetaFLOPS.
Active: 86,536 volunteers, 985,594 computers.
Daily change: +31 volunteers, +9452 computers.

GRID: The analogy..

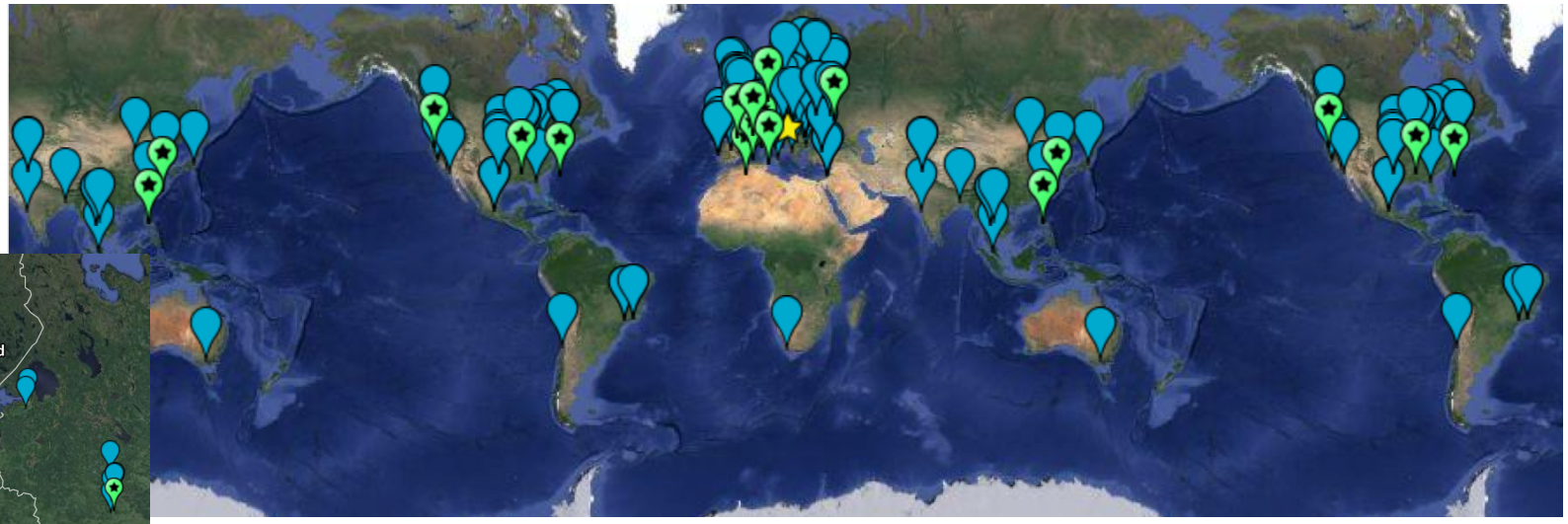
- The electricity network provide me the necessary electricity power when I need it but I do not know which power plants provide me this power.

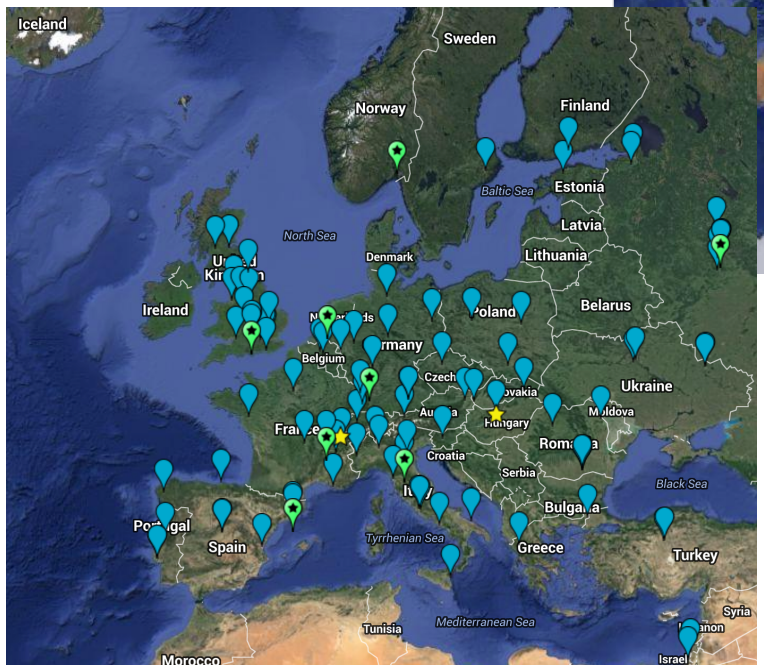
From the source...

Thierry Priol

WLCG
Worldwide LHC Computing Grid

- The Worldwide LHC Computing Grid (WLCG) project is a global collaboration of more than 170 computing centres in 42 countries, linking up national and international grid infrastructures.





The mission of the WLCG project is to provide global computing resources to store, distribute and analyse the ~30 Petabytes (30 million Gigabytes) of data annually generated by the Large Hadron Collider (LHC) at CERN on the Franco-Swiss border.

Knowledge GRID: example

BBC Mobile | News | Sport | Weather | Travel | TV | Radio | More

NEWS TECHNOLOGY

Home | UK | Africa | Asia-Pac | Europe | Latin America | Mid-East | South Asia | US & Canada | Business | Health | Sci/Environment | Tech

20 September 2011 Last updated at 12:33 GMT

2,808 Share

Online game Foldit helps anti-Aids drug quest

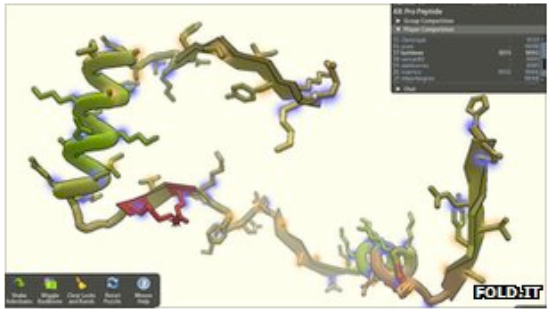
By Katia Moskvitch
Technology reporter, BBC News

An online game has helped determine the structure of an enzyme that could pave the way for anti-Aids drugs.

The game, called Foldit, allows players to create new shapes of proteins by randomly folding digital molecules on their computer screens.

In the journal Nature Structural and Molecular Biology, scientists write that they have been puzzled by the protein's structure for over a decade.

But it took the online community just a few days to produce the enzyme's model.



Retroviral proteases have a critical role in the development of an Aids virus

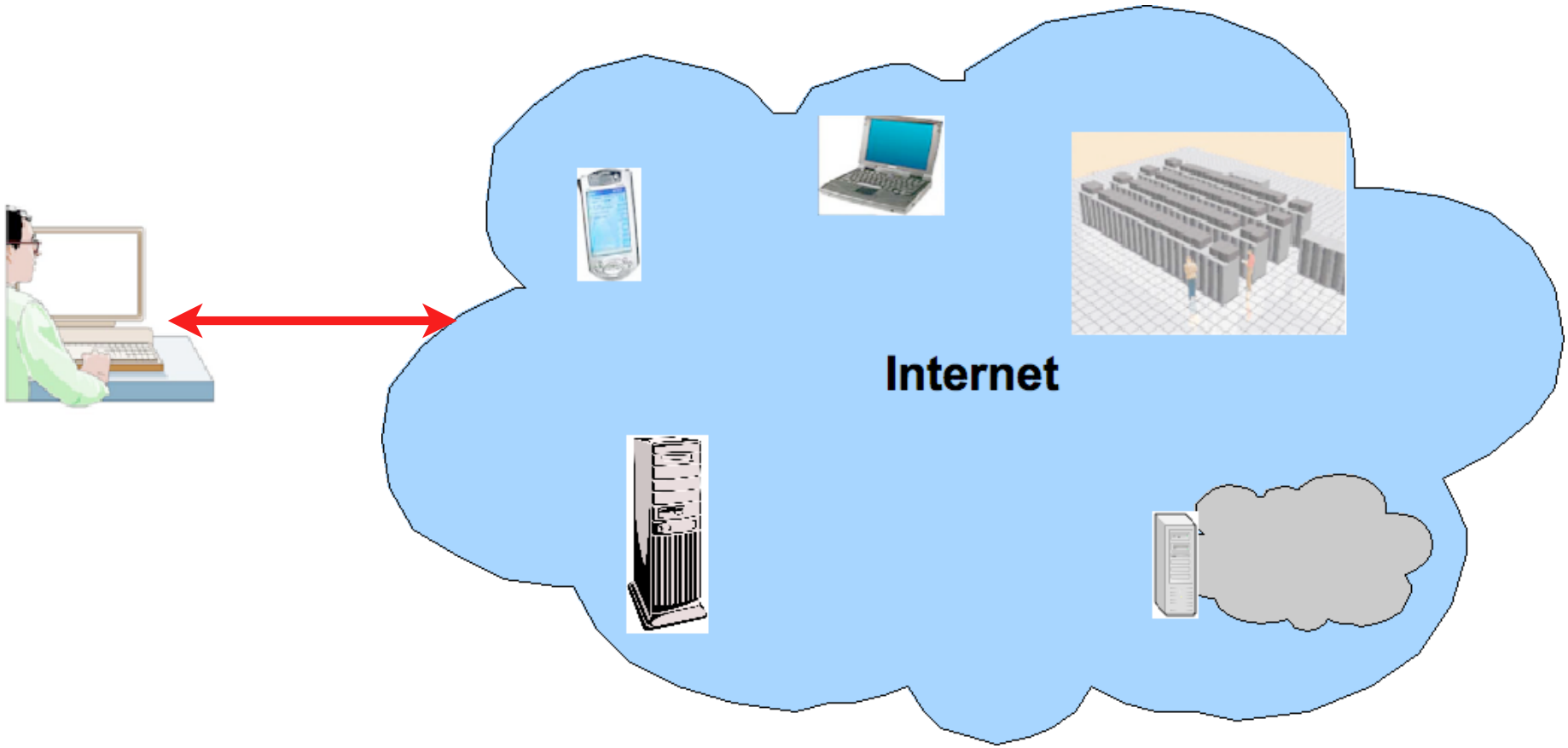
Related Stories

Top Sto

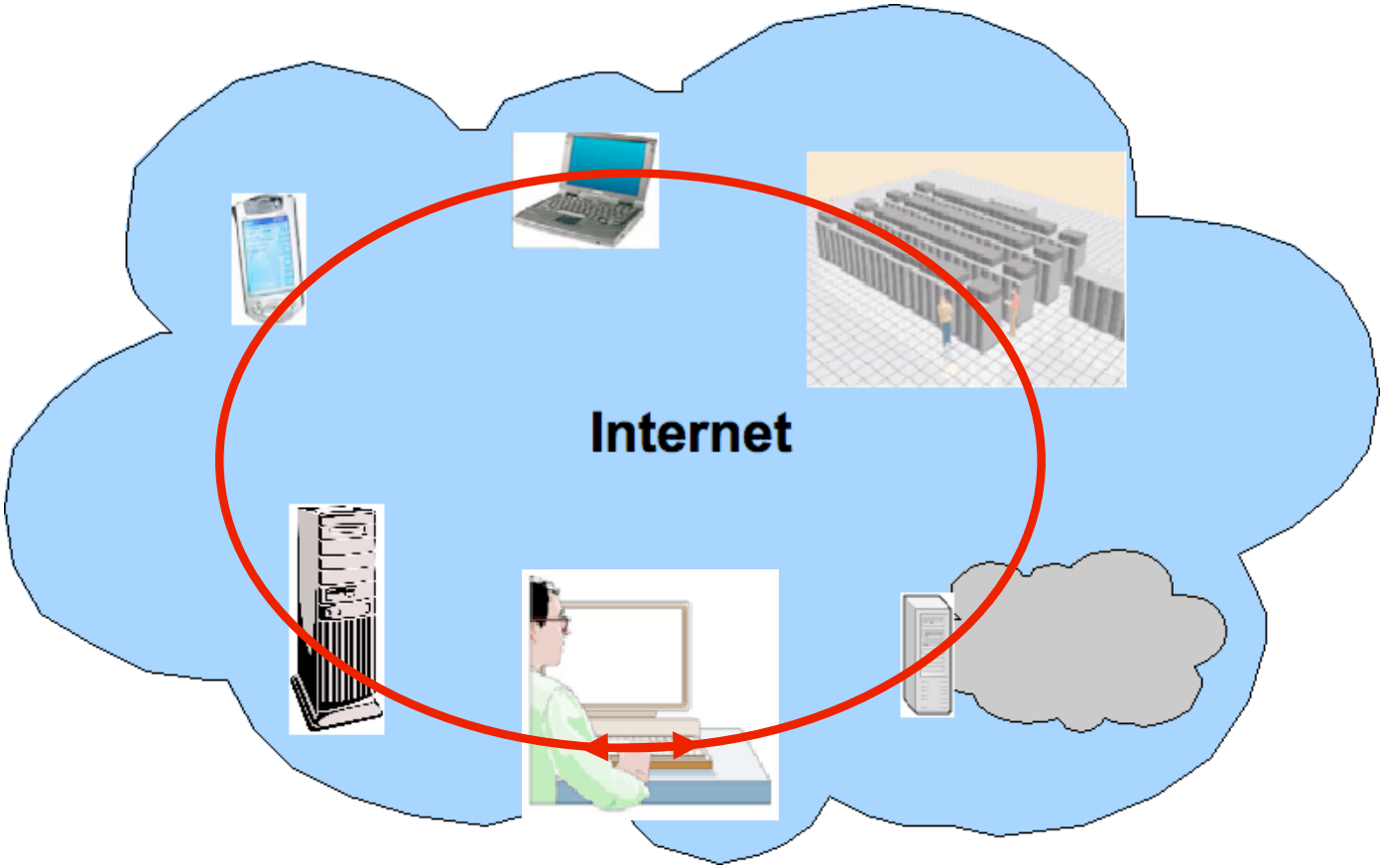
NZ races to e
Embattled De
Brain 'rejects
Cameron to r

http://fold.it/portal/

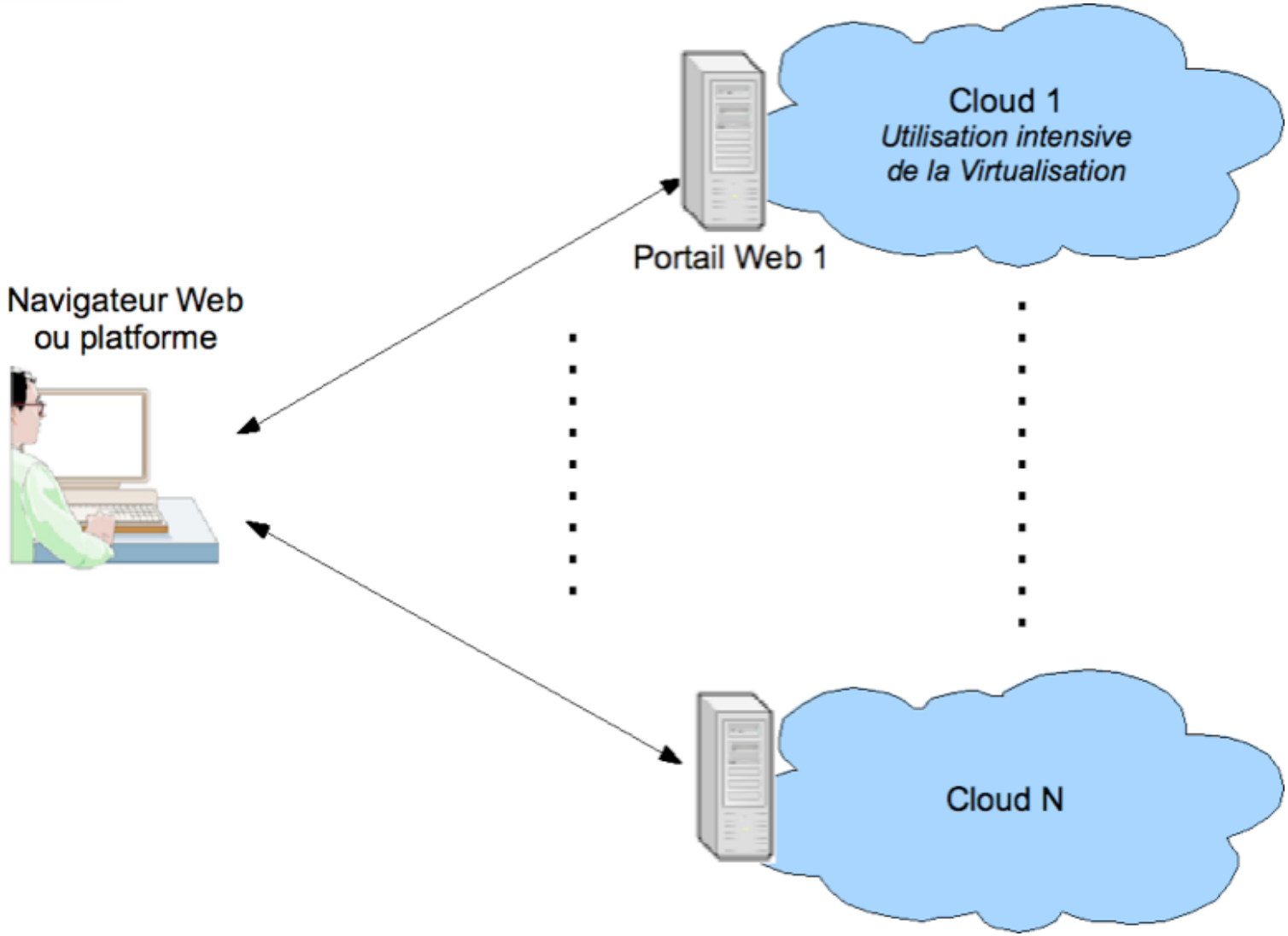
The GRID concept



The GRID concept

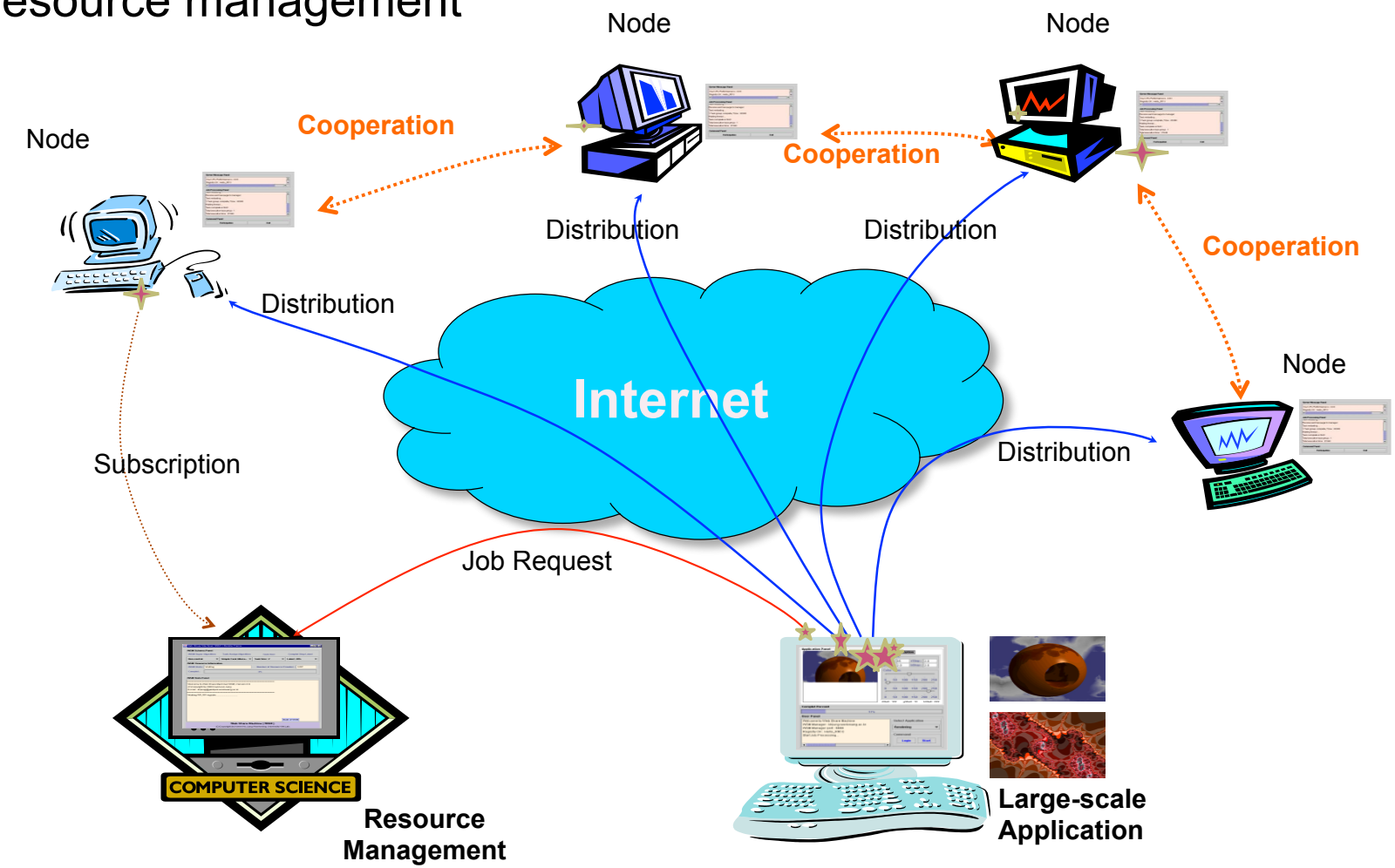


The Cloud Concept



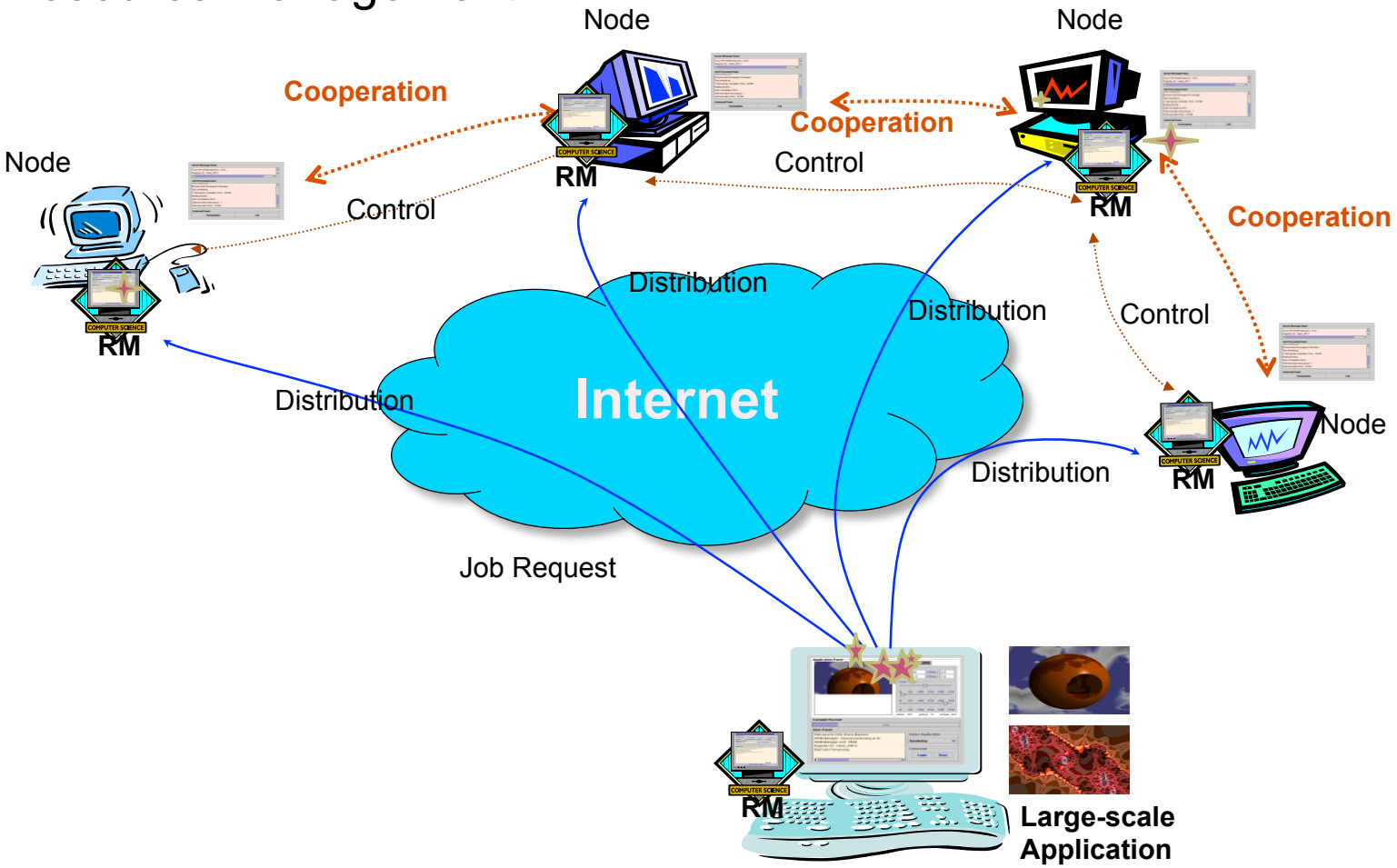
Resource management (1)

- With **centralized** resource management
  - **CLOUD**



Resource management (2)

- With **decentralized** resource management
  - **GRID**



How to realize a Distributed System

- Centralized
  - Information is centralized
  - Global view of the resources
  - Resources accessed through a portal
- Fully distributed
  - Information is distributed
  - No global view of the resources
- Heterogeneous
  - A mix of both

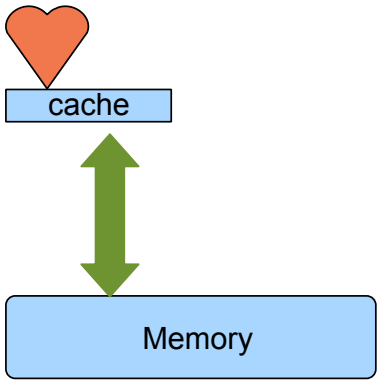
Client/Server (C/S)
Cloud

Peer-To-Peer (P2P)

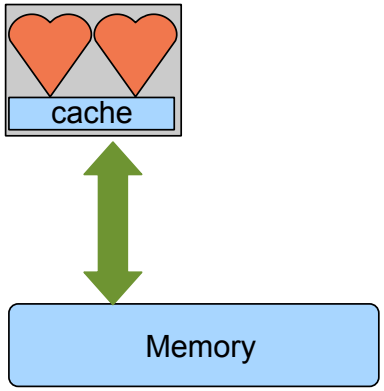
GRID

Distributed systems

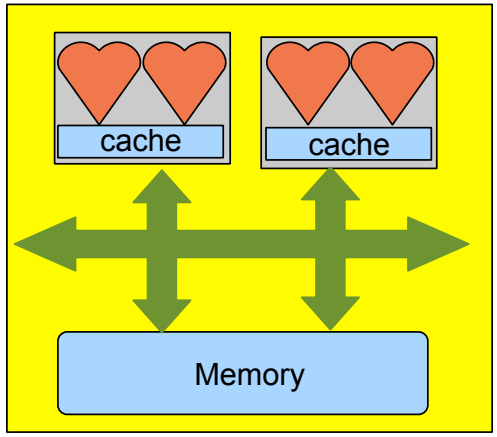
A Core



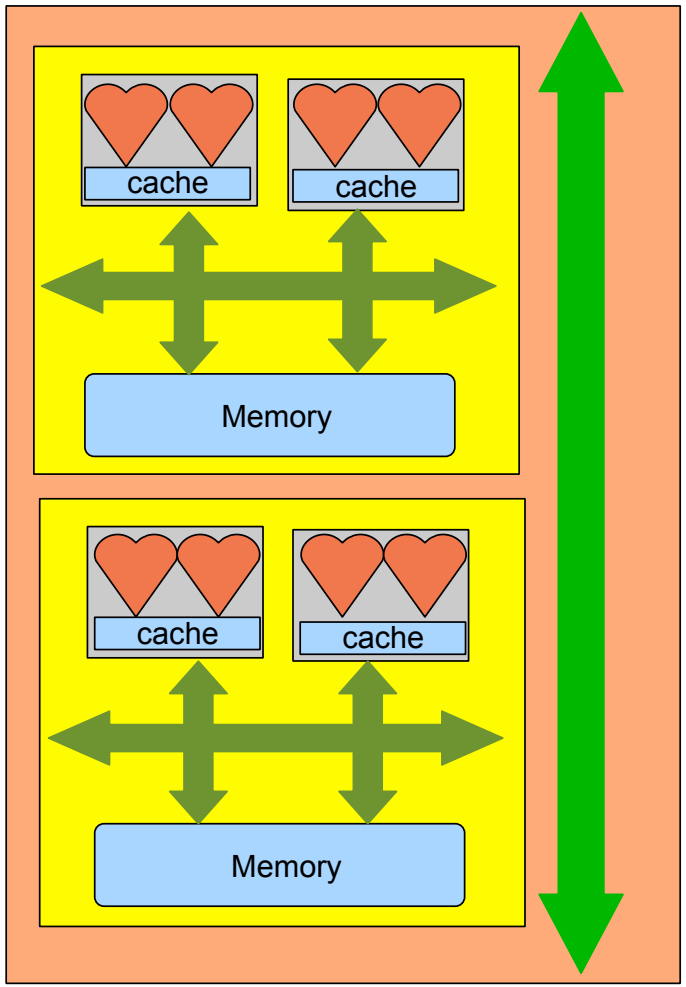
A Multi-Core Processor



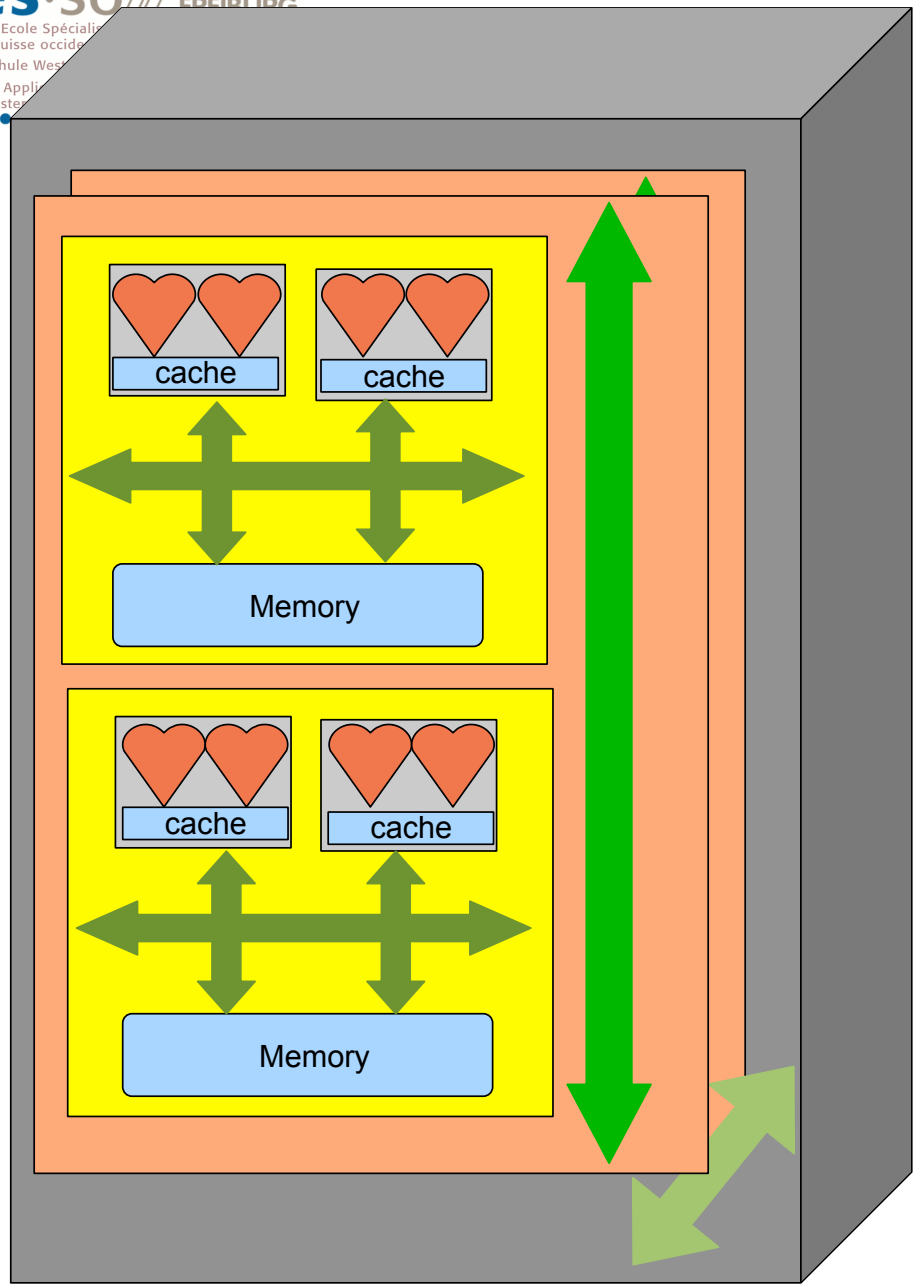
A Multi-processor Board



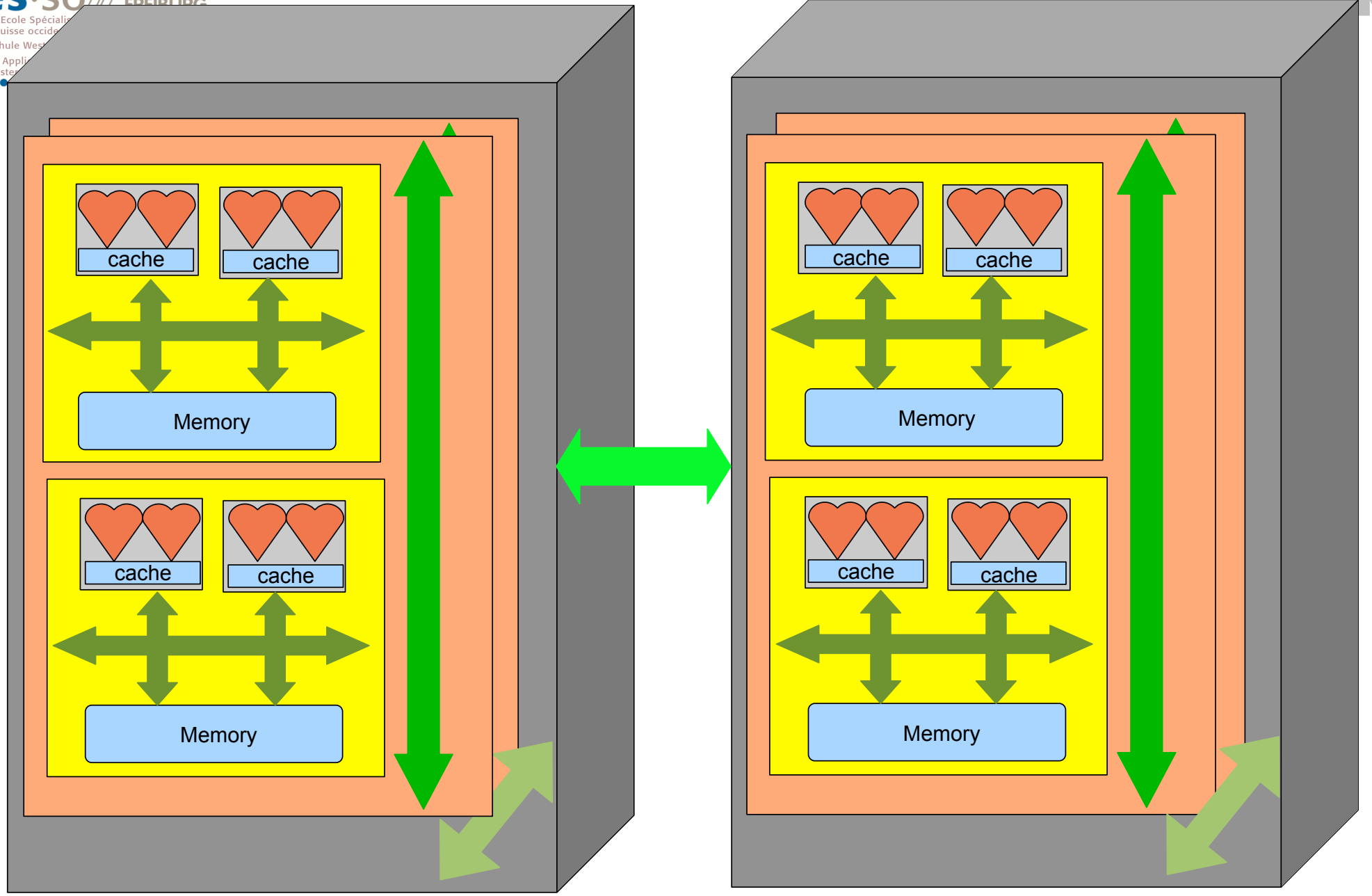
COW/NOW (Cluster/Network)



A local GRID/Cloud



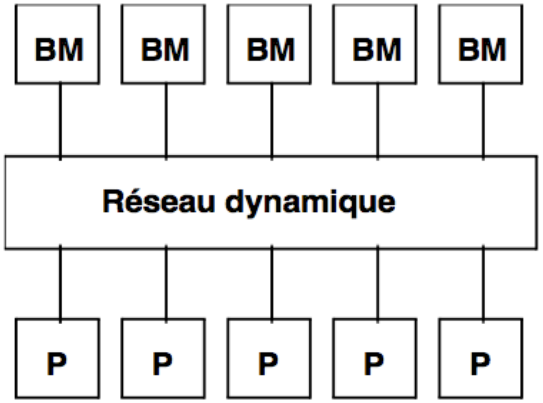
a GRID/Cloud


cache
cache
Memory
cache
cache
Memory
cache
cache
Memory
cache
cache
Memory

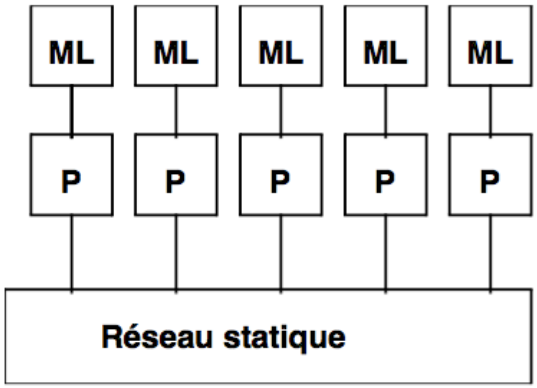
Multiprocessors computers

- Key-point component
  - The communication network: Bus, point-to-point, routed, dynamic, static,.....

-Shared memory



-Distributed

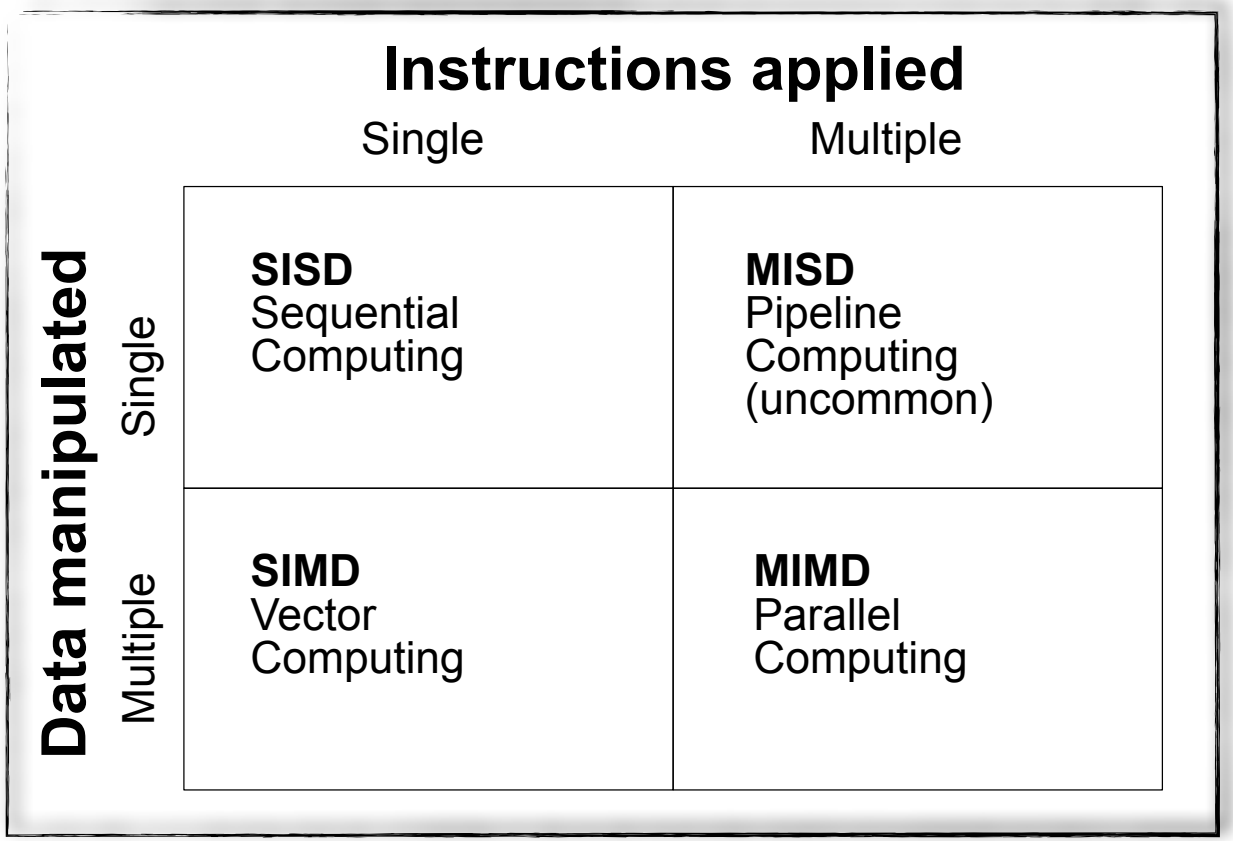


Shared vs Distributed memory

- Due to the rapid increasing of the numbers of connected processors it rapidly appeared that purely “shared memory” supercomputer is not an option (memory bottleneck).
  - Vocabulary
    - Processing unit
      - Smallest sequential processing element: Example a core
    - SMP
      - Single Memory Multi Processor: Shared memory multi processing unit: Example a multicore
    - Node
      - Components which are connected through the network in a distributed memory multiprocessors computers
- Distributed computers: two options for getting the power
  - Many low power nodes : Massively Parallel Processor (MPP) -> “army of ants”
    - PCs have boosted this approach
  - Few high power nodes : connecting SMP nodes -> “herd of elephants”
    - Multi-Core architecture have re-launched the concept of SMP

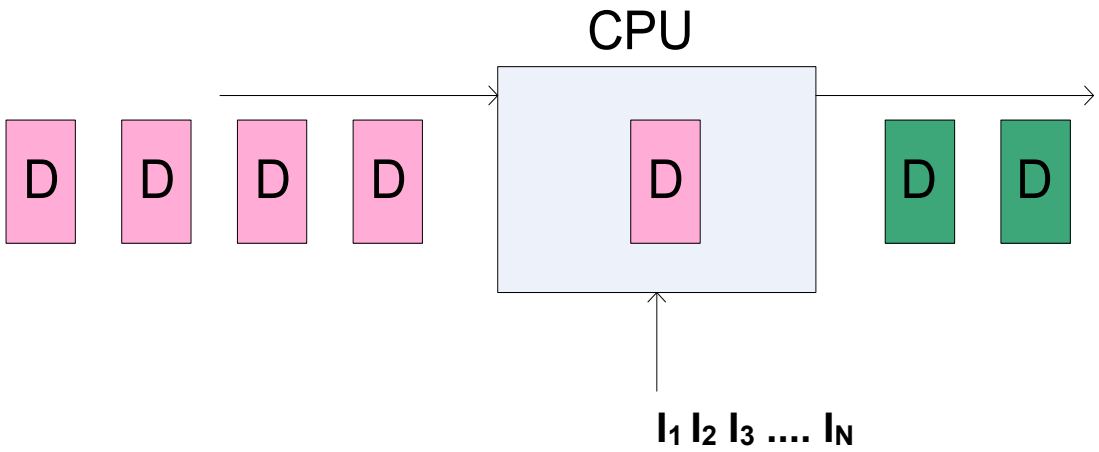
Flynn's Taxonomy of Computers

- The most known and used classification
  - Not recent: 1966
- Based on two criteria
  - Instructions
  - Data
- Originally conceived for the classification of the hardware
  - Can be extended to classify models of computing



SISD

- One Processing element, One flow of data
  - Sequential computing

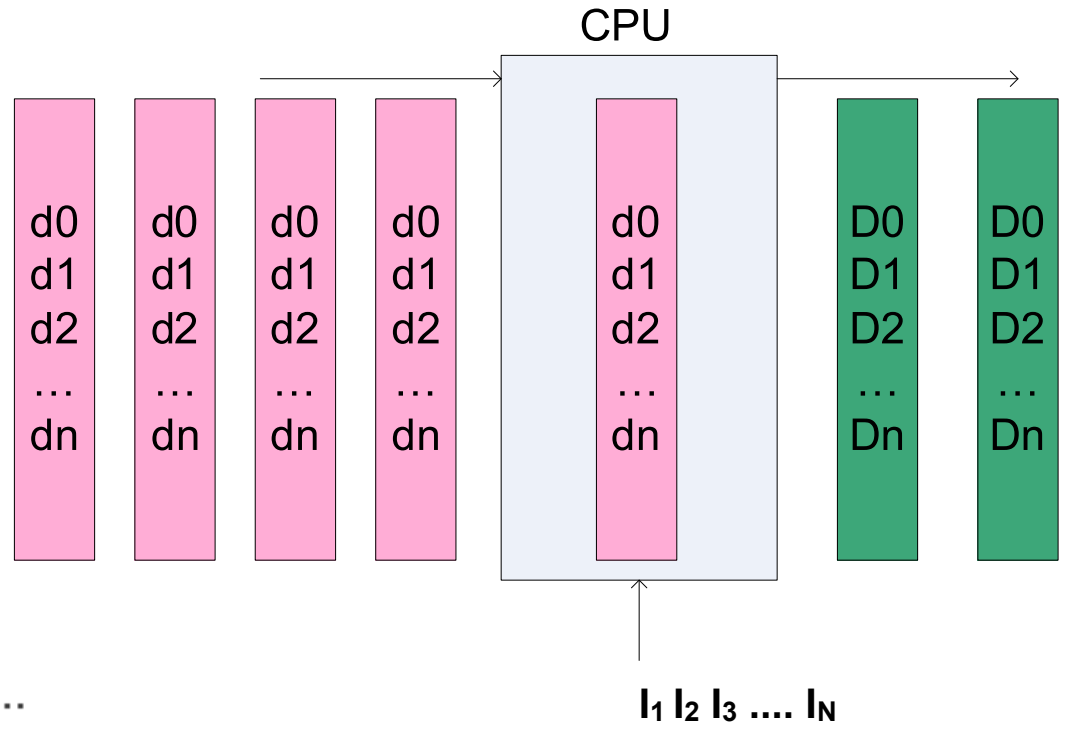


From the source...

Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

SIMD

- One processing element acting on several data
  - Vector computing

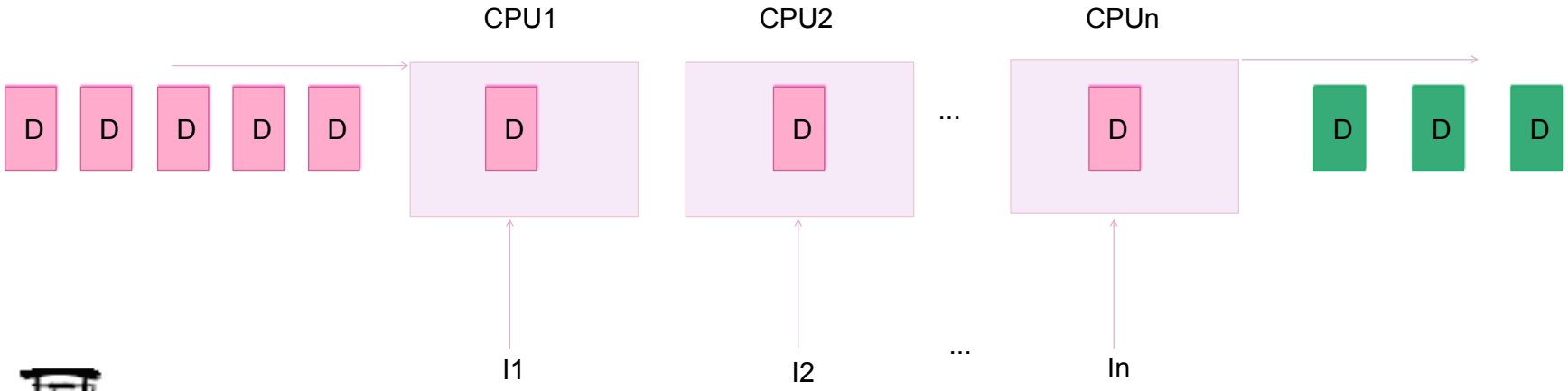


From the source...

Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

MISD

- Several computing units acting on one flaw of data
  - Pipeline computing



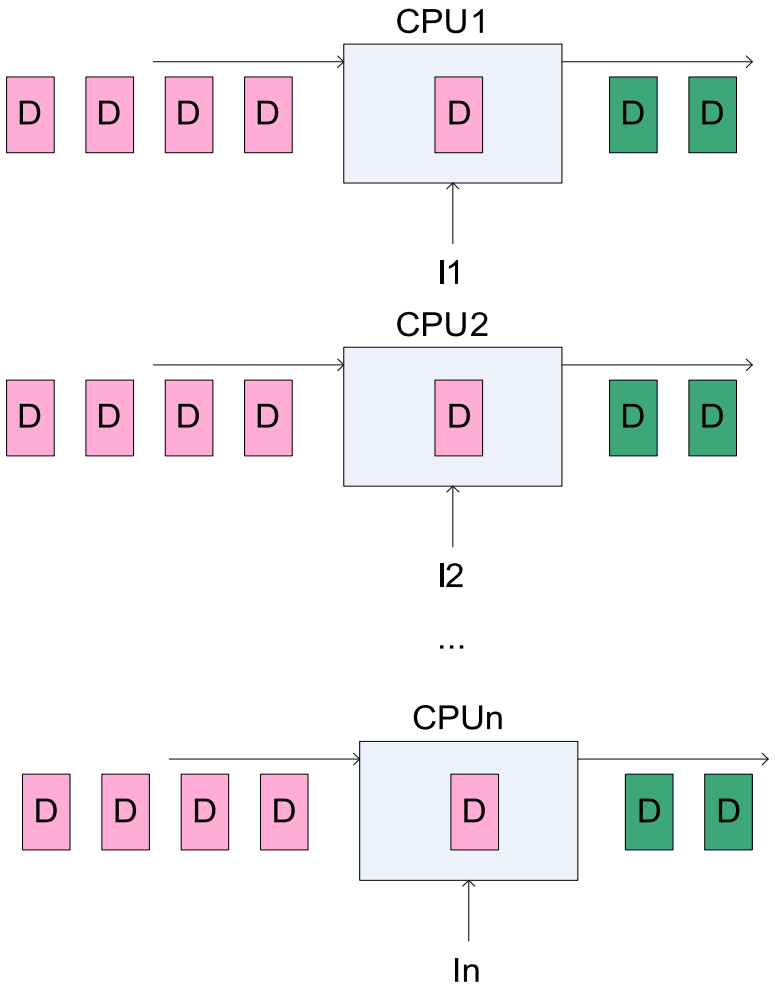
From the source...

Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

MIMD

- Several processing units acting on several flaws of data
  - Parallel computing

- What about ?
  - Synchronization
  - Memory



From the source...

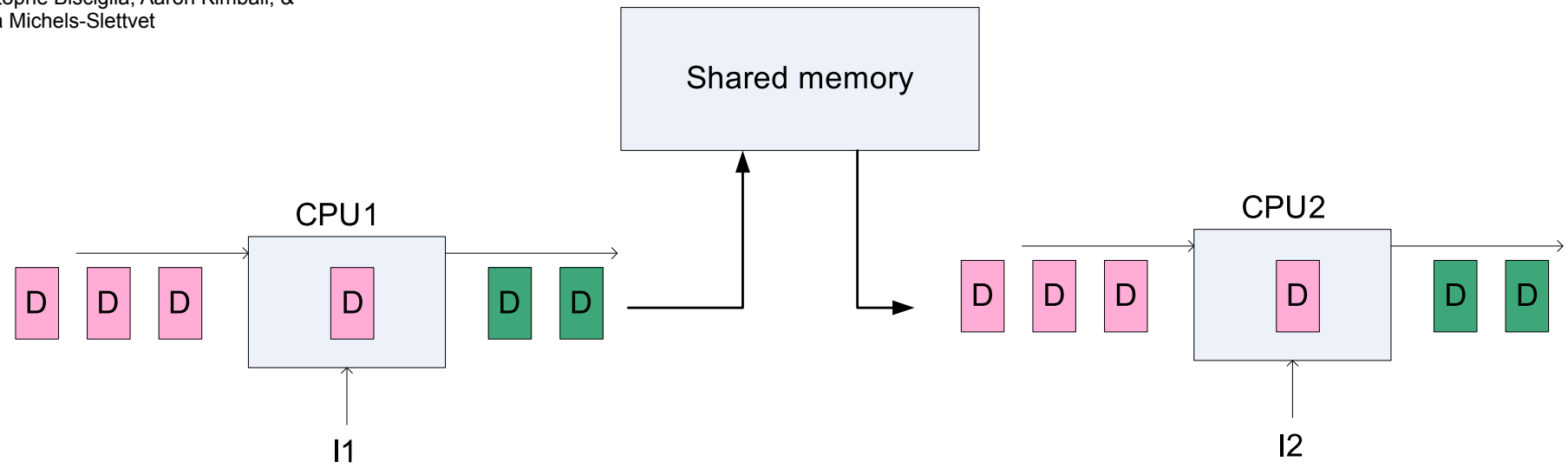
Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

Parallel vs. Distributed Computation

- Parallel processing refers to multiple CPUs within the same shared-memory machine performing computation

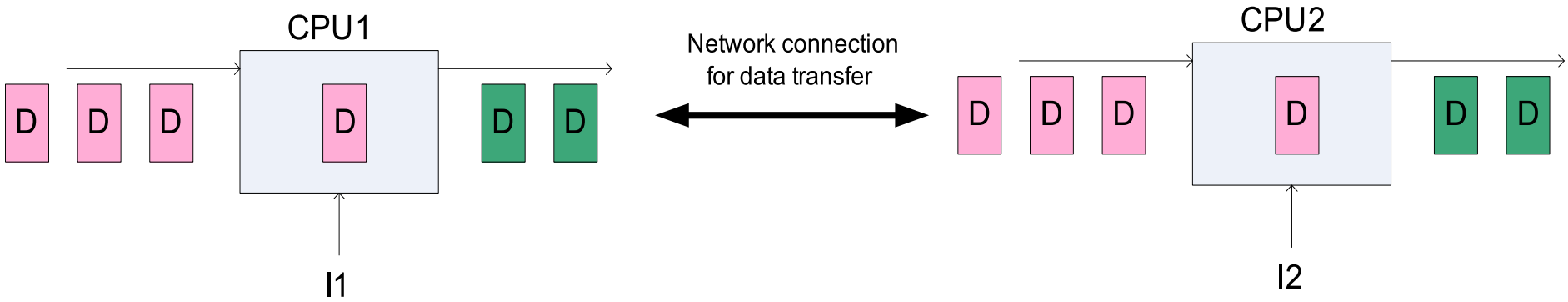
From the source...

Christophe Bisciglia, Aaron Kimball, &
Sierra Michels-Slettvet



Parallel vs. Distributed Computation

- Distributed computation involves multiple computers with their own memory communicating over a network



- This approach solve the problem of the memory (shared or distributed)
  - But what about the synchronization (the control of the computing) ??

From the source...

Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

Sources of parallelism

- One can identify three sources of parallelism

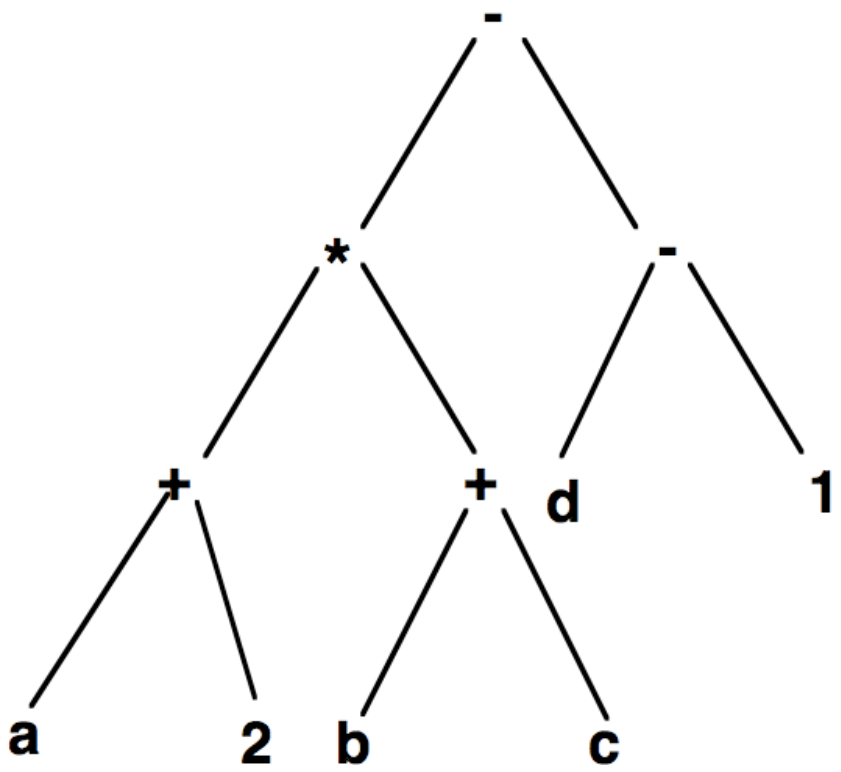
  - Control parallelism
    - To do several different things in the same time

  - Data parallelism
    - To do the same thing on several different data

  - Flow parallelism
    - Line work

Control parallelism

- Example
  - To compute (a+2)*(b+c) – (d-1)
  - We can represent this expression in the form of an evaluation tree



Dependency

- This tree suggests a way to calculate this expression in three steps
  - 1) to compute x1 = (a+2) and x2=(b+c) and x3=(d-1)
  - 2) to compute x4 = x1*x2
  - 3) to compute x5 = x4 - x3

- In the list above, each line can be calculated only if the previous line has been calculated
  - It is said that there are dependencies between the instructions

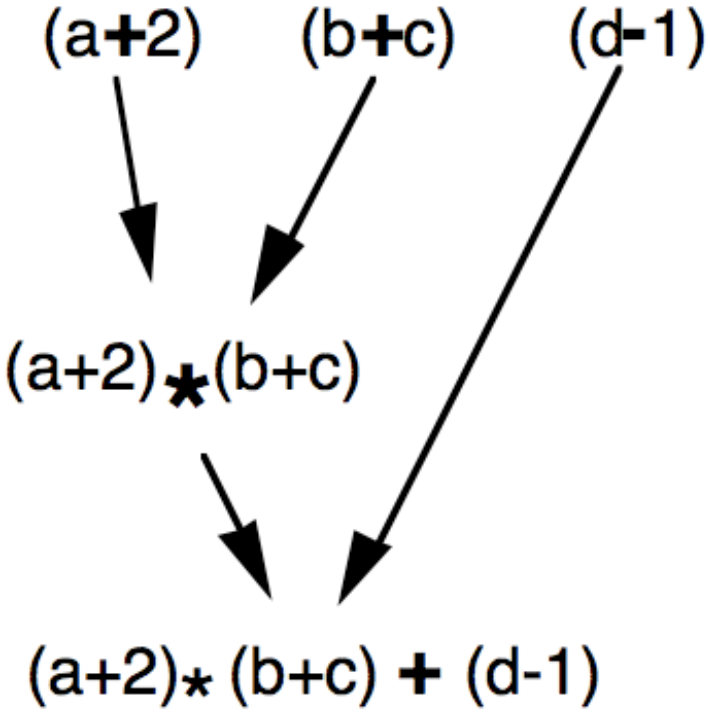
- These dependencies can be graphically represented by a **dependency graph**

Dependency graph

- The dependency graph of the expression : (a+2)*(b+c) – (d-1)

- By observing the graph we can note that there are several ways to calculate this expression that are not all necessarily equivalent
  - Example:
    - 1) (a+2) et (b+c)
    - 2) (a+2) * (b+c) et (d-1)
    - 3) (a+2) * (b+c) - (d-1)

    - Requires only two processors



Exercise: Control parallelism

- Parallelize, at best, the calculation of the following expression:

$$(X*(Y+X^2))-((Z^2+2)*(X-Y)*4*(Y-4)*X^3)$$

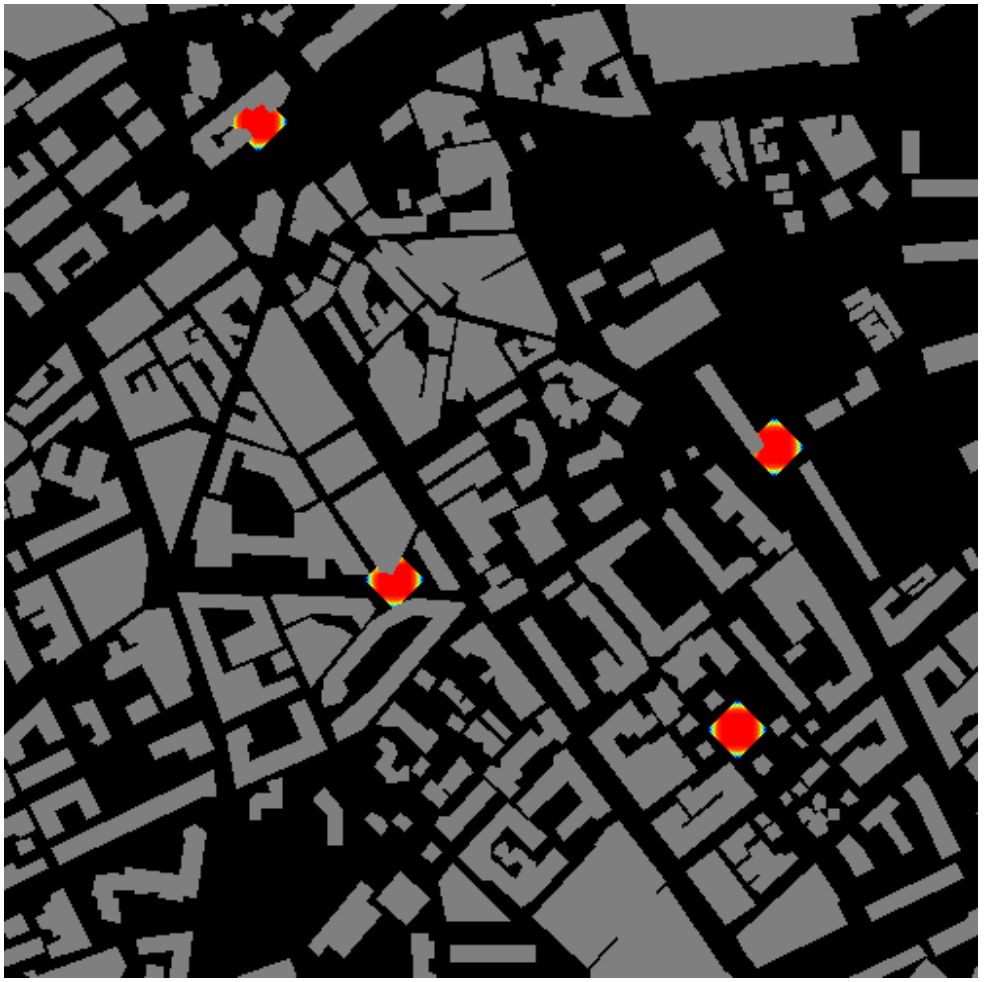
Data parallelism

- Data parallelism comes from the observation that, often, we have to repeat the same action or operation on different data.
  - e.g image processing

- Many applications use large amounts of data (such as, for example, arrays) on which you must repeat the same action.
  - This is particularly true in the case of scientific calculations making intensive use of linear algebra and therefore operations on very large matrices
    - e.g.: matrix multiplication

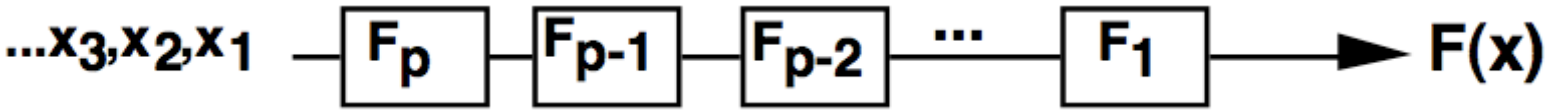
Example

- ParFlow + +
  - 2D simulation of radio waves propagation in urban environment
  - Used in the European project STORMS :
    - Software Tools for the Optimization of Resources in Mobile Systems



Flow parallelism

- The flow parallelism stems from the fact that in some applications one can operate according to a **line work**.
  - The typical use case of flow parallelism is when you apply to the data a function F which can be decomposed into several functions:
    - $F(x) = F_1(F_2(F_3(...F_p(x)..))) = (F_p \circ F_{p-1} \circ ... \circ F_3 \circ F_2 \circ F_1)(x)$
  - In such a case the computation can be divided into P stages as follows:

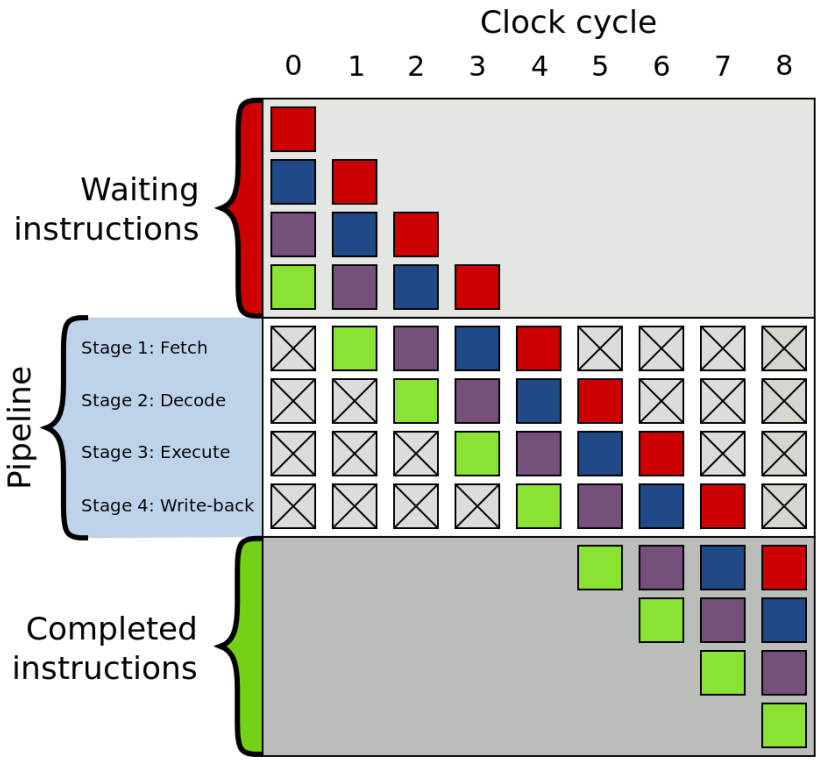


- Such an architecture is called: **a pipeline**
  - To efficiently operate a pipeline it is necessary to have an input data stream of sufficient length. Indeed, P steps are necessary to fill the pipeline.

Pipelining in modern processors

- Instruction pipelining
  - To speed up execution, modern processors execute multiple instructions in parallel
  - The instructions are separated into multiple stages. E.g. 4 steps
    - Fetch (Read the instruction)
    - Decode (Determine what the instruction does)
    - Execute (Execute the instruction)
    - Write-back (Save the changed values to registers)
  - Modern CPUs have up to 20 stages for their pipelines
    - Intel core I9 Skylake 7980XE
  - It is the compilers job to reduce dependencies of instructions close to each other
    - Example

```
1: add 1 to R5
2: copy R5 to R6
```



Branch prediction

- When pipelining, branches are problematic
  - Which side of the branch to put in the pipeline?
- If wrong branch is selected, pipelined instructions need to be discarded
- Different strategies exist
  - Static branch prediction
    - Assume no jump is taken
  - Dynamic branch prediction
    - Relies on statistics of previous executions of the branch
      - Various strategies exist for this, including neuronal networks

Speculative execution

- Tasks are executed even if not sure that they will be used
  - Branch prediction is one example but also:
    - Prefetching memory/files

- This can cause security problems, as recently shown by Spectre and Meltdown
  - They are essentially timing attacks, exploiting cached data through speculative execution

Speedup

- Definition:
  - Given T_1 the necessary execution time of a program that solves a problem A on a sequential computer and $T_{//p}$ the necessary execution time of a parallel program to solve the same problem A on a parallel computer using p processing elements , then the **Speedup** is the ratio:

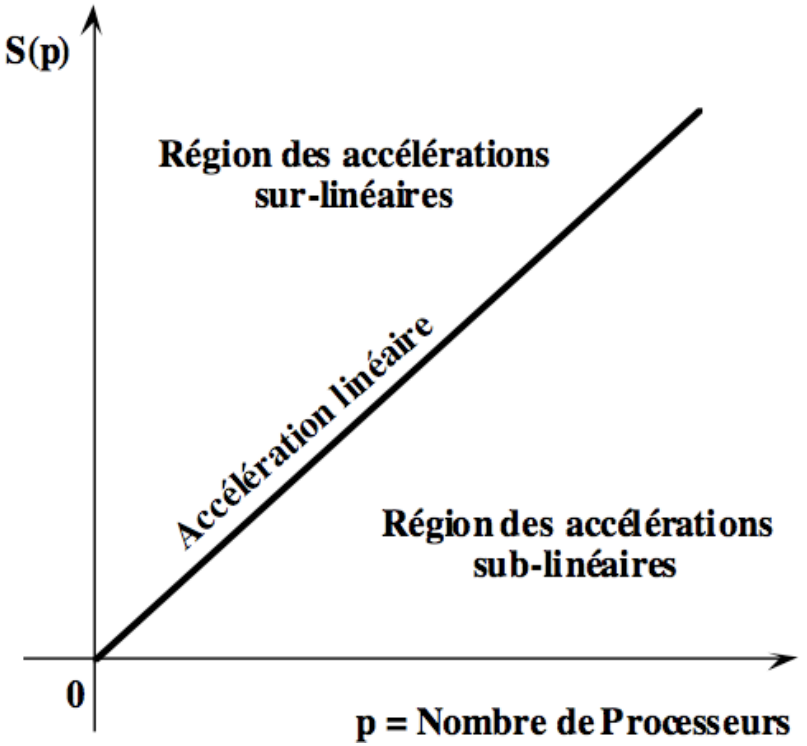
    $S(p)=T_1 / T_{//p}$

- Requires a clear definition of the following elements:
  - A sequential program that solves the same problem
  - A sequential computer
  - p processing elements
- Always specify how the speedup has been calculated
  - What we have exactly compared

Characteristics of the speedup

- The reference value considered as **ideal** is the function S(p) = p
  - it is called **linear speedup**

- An important question is:
  - Can the speedup to go over above the linear speed up??
    - In other words is S(p)>p possible

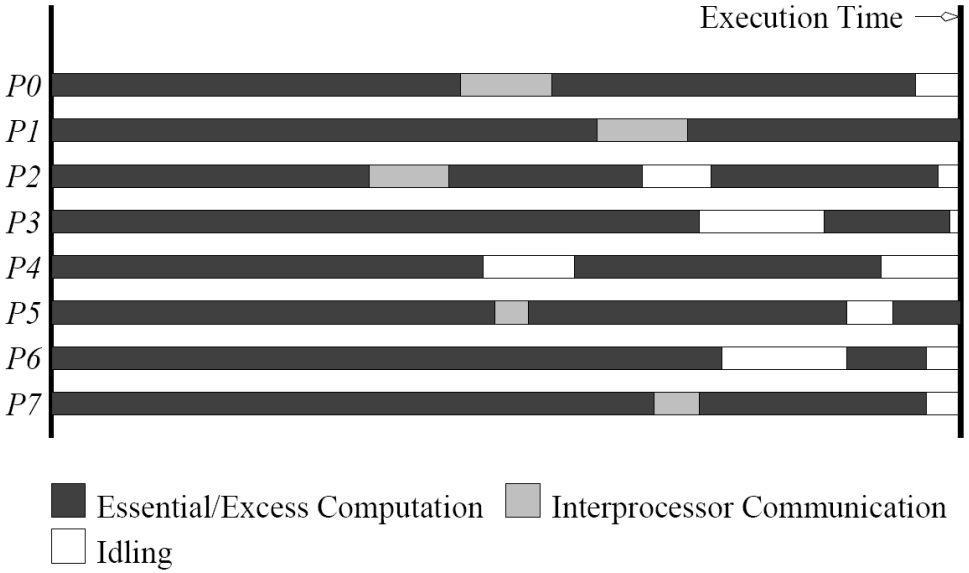


Question ?

- If I use N processors, shouldn't my program run N times faster?
- Usually NO

– A number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.

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



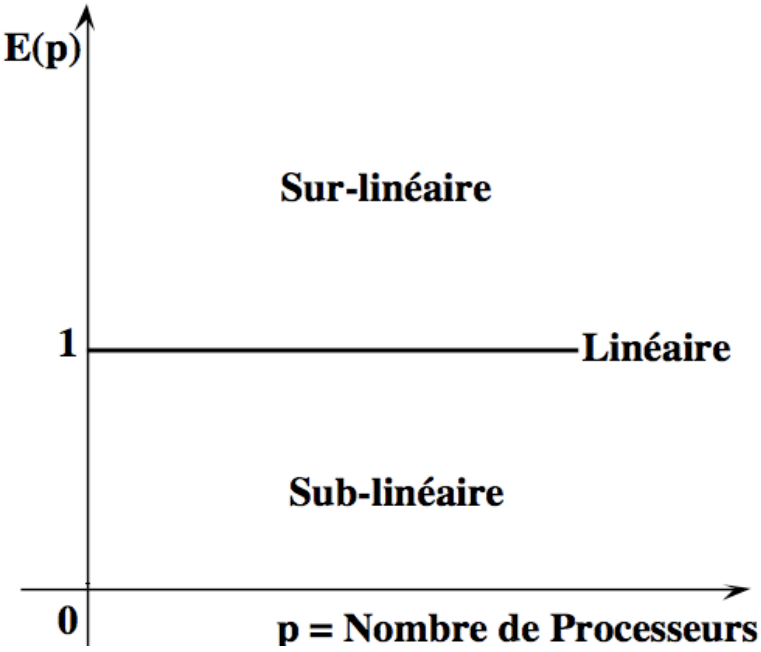
Exercise: Pipeline and speedup

- Determine the speedup of a pipeline on length p when the length N of the input stream tends to infinity.
  - We assume that the calculation of the F_i (for i = 1 .. p) takes one unit of time

Efficiency

- Definition :
  - The efficiency E(p) is the ration between the speedup S(p) and the number of processing elements : E(p) = S(p)/p

- The efficiency is à normalization of the speedup versus the number of processing elements.
- An efficiency of 1 corresponds to a linear speedup i.e. to an optimal usage of the processing elements.



The Amdahl law (1967)

- The Amdahl law considers that the execution time T_1 of a sequential program can be split in two parts:
  - a time T_s used to execute the part of the program which is inherently sequential
  - a time T_p used to execute the "parallelisable" part of the program.

- We obtains the following equation: $T_1 = T_s + T_p$
  - Only T_p can be decreased thanks to the parallelization and, in the ideal case, with a linear speedup. Thus, using p processing elements we can, at best, have a time T_p/p to execute the "parallelisable" part.

- As a consequence the execution time $T_{//p}$ of a parallel program on p processing elements (the parallel execution time) is subject to the following constraint:
  - $T_{//p} \geq T_s + T_p/p$

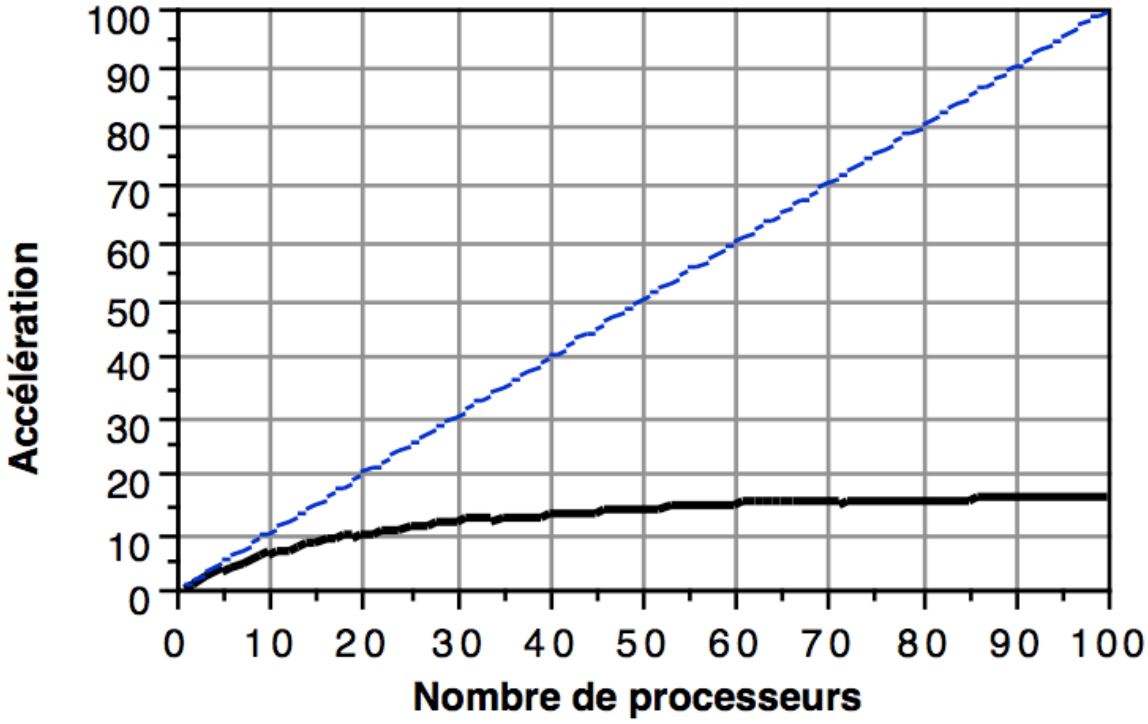
Amdhal's law and speedup

- If we introduce the two above relations in the definition of speedup we obtain:
  - $S(p) \leq (T_s + T_p) / (T_s + T_p/p)$
- If we now turn to the asymptotic value of the speedup that is the limit to which the acceleration tends when the number of processors tends to infinity we obtain:
  - $S(\infty) = (T_s + T_p)/T_s = T_1/T_s = 1/(T_s/T_1)$
- The value "T_s/T_1" defines the proportion of the program which is inherently sequential. We obviously have: $T_s/T_1 \leq 1$
  - If we denote **f** = T_s/T_1 this value, we obtain:
    - $S(\infty) = 1/f$
      - Amdahl's law is usually presented in this way

Consequences of Amdahl's Law

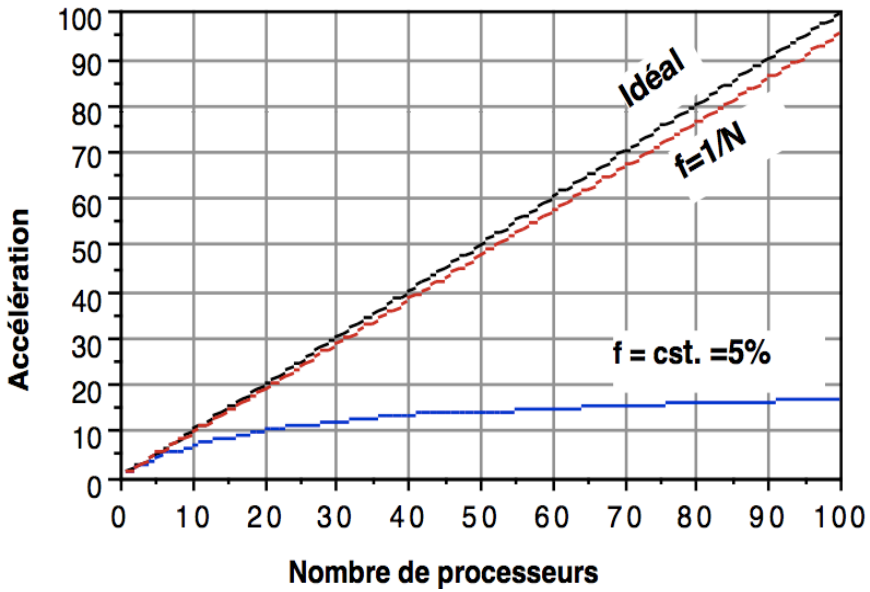
- Amdahl's Law expresses the intuitive idea that the upper limit of the speedup is inversely proportional to the proportion of inherently sequential code contained in the program

  - The figure below shows an example of a program that contains 5% of inherently sequential code
  - We clearly see that the speedup tends to 20 (=1/0.05)
  - The Amdahl law is a strong limitation of gaining performances when parallelizing a program



The Gustafson law

- In 1988 Gustafson et Barsis have proposed that, in most of applications, the proportion of inherently sequential code decreases as the size of the problem increases.
  - In other words: the larger the problem, the more parallel it becomes.
    Let's denote N the size of the problem:
    - We have : $T_s(N) = f(N) \cdot T_1(N)$ and $T_p(N) = T_1(N) - T_s(N)$
    - In $T_{//p}(N) \geq T_s(N) + T_p(N)/p$ we replace T_s and T_p
      - $T_{//p}(N) \geq f(N) \cdot T_1(N) + (T_1(N) - T_s(N))/p$
      - $T_{//p}(N) \geq f(N) \cdot T_1(N) + (T_1(N) - f(N) \cdot T_1(N))/p$
      - $T_{//p}(N) \geq f(N) \cdot T_1(N) + (1-f(N)) \cdot T_1(N) /p$
      - $T_{//p}(N) \geq T_1(N) \cdot (f(N) + (1-f(N))/p$
  - We finally obtain for the acceleration
    - $S_p(N) = T_1(N)/T_{//p}(N) \leq p/(p*f(N)+1-f(N))$
    - **$S_p(N) \leq p/[1+f(N)(p-1)]$**
- The figure on the right shows two cases
  - (1) f(N)=1/N and (2) f(N)=cst=5%
    - when f(N) = cst, we have the Amdahl's Law

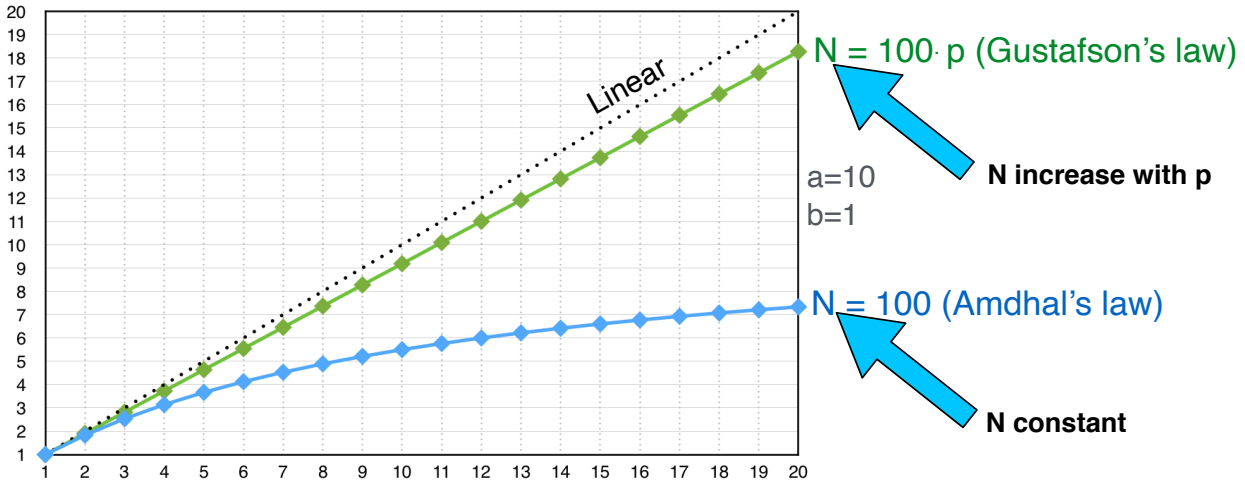


where the size of the problem linearly increases with p

The Gustafson/Amdhal's laws: example

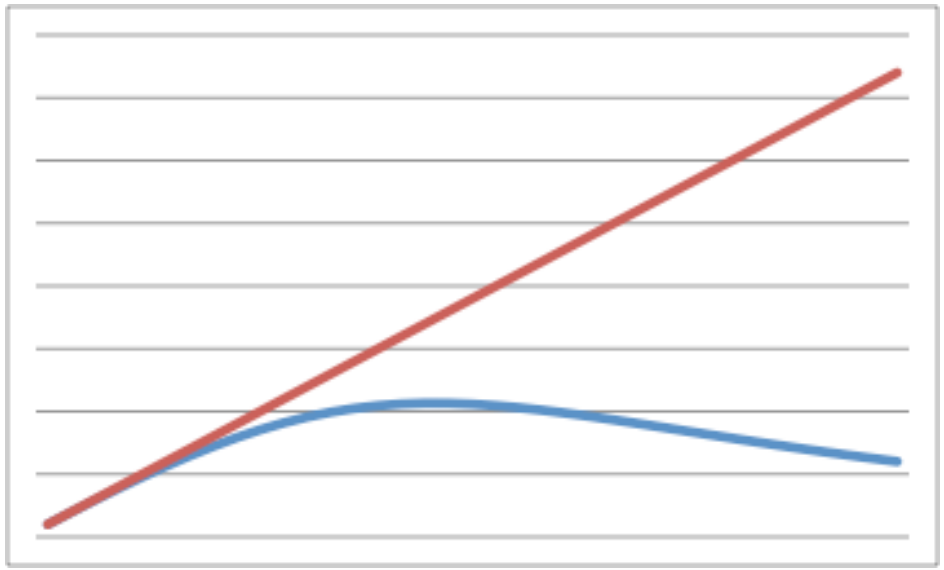
- Let's assume an algorithm A(N) which works this way:
  - It reads the data of size N using a·N units of time (*a is constant)*
  - It computes the result using $b.N^2$ units of time (*b is constant)*
- => $T_1(A(N)) = a.N + b.N^2$
- We write a parallel version $A_p(N)$ of this algorithm which works this way:
  - It reads the data of size N using a·N units of time
  - Each processing element computes the result using $b.N^2/p$ units of time (*p processing elements)*
- => $T_{//p}(A_p(N)) = a.N + b.N^2/p$
- $S_p(N) = T_1(A(N))/ T_{//p}(Ap(N))$

$$= (a \cdot N + b \cdot N^2)/(a \cdot N + b \cdot N^2/p)$$
$$= N \cdot (a + b \cdot N)/N \cdot (a + b \cdot N/p)$$
$$= (a + b \cdot N)/(a + b \cdot N/p)$$
$$= p(a + b \cdot N)/(p \cdot a + b \cdot N)$$

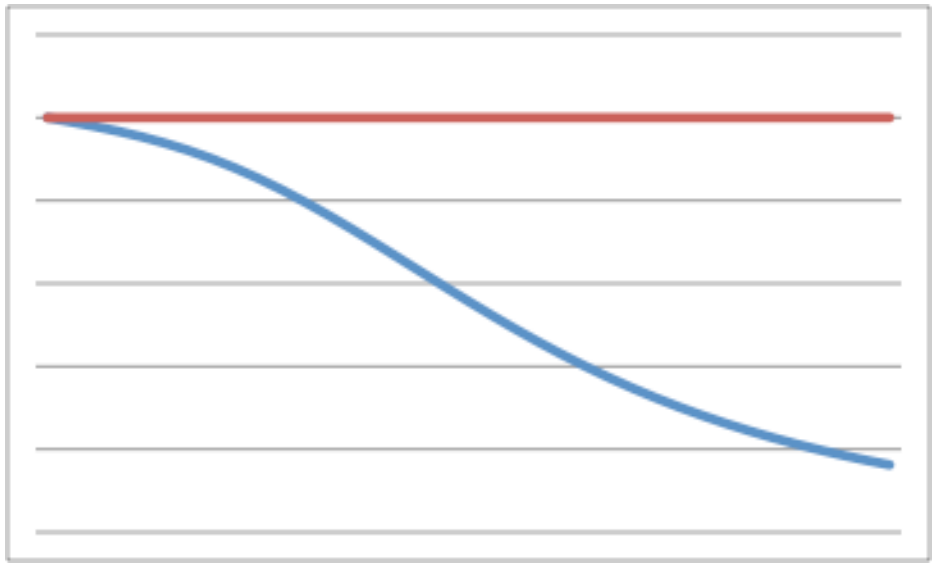


Speedup and efficiency

- In most cases the actual speedup and efficiency curves have the following general shape



Speedup



Efficiency

The performance

- In this course we are interested by : The performance
- We have seen how to evaluate and compare performance of computers
- We have now to evaluate an compare performance of programs
  - How to measure or to evaluate this performance?
    - To measure the execution time
    - To evaluate the Flops (Floating points operation per second)
    - .....
- Two types of performance
  - Time performance
    - Amount of time taken by an algorithm
  - Memory performance
    - Amount of memory space taken by an algorithm

Intuitive Performance Measures

- Wall-clock time

– the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble

– Problem: How does this scale when the number of processors is changed or the program is ported to another machine?

- How much faster is the parallel version?

– Problems

  - What's the baseline sequential version with which we compare?
  - Can we use a sub-optimal sequential program to improve our parallel program?
  - Which model (or real) computers are we speaking about ?

Performance: difficulty

- The main problem with the notion of performance is that
  - It is dependent of the computer we are using
- This is why we prefer the notion of **Complexity**
- As for the performance there are two types of complexity
  - time complexity
  - memory complexity
- More generally one says : Computational complexity

Computational Complexity

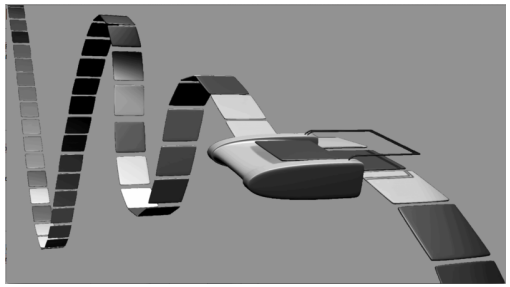
- Definition (Wikipedia (almost))
  - The computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying algorithms solving computational problems according to their inherent difficulty.

- What is an algorithm ?
  - Does an algorithm exist to solve any problems?
    - Computability theory

- The Church-Turing thesis
  - States that a function is algorithmically computable if and only if it is computable by a Turing machine.
  - Mathematicians and computer scientists agree that the concept of computability is accurately characterized by the Turing machine or other equivalent models
    - λ-calculus
    - Cellular Automata
    - ...

The Turing machine (1936)

- The (detreministic) Turing machine is a abstract model of the functioning of computers. It is composed of
  - An infinite tape divided in squares, each square can contain a symbol belonging to a given alphabet
  - A read/write head that can read a symbol, write a symbol in square, or move the tape left and right one (and only one) cell at a time.
  - A status register that stores the current sate of the machine. There are a finite number of possible sates.
  - A finite table (also called transition function) of instructions that, given the current state of the machine and the symbol it is reading on the tape (symbol currently under the head) tells the machine to do the following (unique) sequence:
    - Either erase or write a symbol (replacing) and then
    - Move the head left or right or do not move and then
    - Assume the same or a new state as prescribed, go to this state.



Why the Turing machine is useful

- Theoretical definition of the performance of an algorithm
  - Independent of any specific execution on any specific computer

- Definition
  - Time complexity: Number of steps to execute the algorithm on a Turing machine
  - Memory complexity: Size of tape needed to execute the algorithm
    - The input data of the algorithm is written on the tape before starting the execution and the size of this data (space used on the tape to write the data) defines the size of the instance of the problem we are solving

- What makes the turing machine very powerful is its universality
  - The Turing machine can calculate any function that any sequential computer can calculate
  - The complexity of the algorithm (time and memory) is “equivalent” on the Turing machine and on any sequential computer

Complexity

- Intuitive definition
  - Based on the previous slides we “intuitively understand” that the time and the memory complexities are the time and the memory space used to execute the algorithm to solve a problem of a given size

- The theory of complexity give a more precise definition
  - This theory concerns
    - “Worst case” complexity
    - Order of magnitude
    - Asymptotic behavior
  - In other words
    - We want to know:
      what happens when solving the most difficult instance of a given problem
      what are the main parameters that influence the complexity
      what happens when the size of the problem goes to infinite

Complexity: Example

- Assume that the number of steps requires to solve the most difficult instance of a problem P with the algorithm A for an data input size of N is:
  - $T_C(P, A) = 3N^2+N+12$

- Then the "asymptotic time complexity" of the algorithm a is: N^2
  - Because when N becomes very big, N+12 becomes negligible towards $3N^2$
  - By neglecting the multiplicative factor (here '3') we have the following result
    - The resulting complexity (here N^2) complexity of a given algorithm is the same on an deterministic Turing machine than on any sequential computers

- In the remaining of this course, if nothing else it is explicitly stated, when we say "complexity" we assume "asymptotic complexity".
  - We note $O(3N^2+N+12) = N^2$

Complexity and parallel algorithm

- All what we presented concerning the Turing machine and the complexity is valid for sequential computers and sequential algorithms

- What about parallel algorithms and parallel complexity
  - Unfortunately the situation is not clear
  - The (deterministic) Turing Machine (TM) is not equivalent to all parallel machines or algorithms
  - What about the non-deterministic Turing (NTM) machine
    - TM and NTM are equivalent in the sense of calculability
      - TM and NTM can compute the same set of functions (the calculable functions) BUT
      - not necessarily with he same complexity
  - In addition
    - All models of parallel machines are not equivalent in term of complexity

Parallel complexity

- Time parallel complexity
  - The parallel time complexity of a parallel algorithm depends on
    - the input size N,
    - the number of processors p,
    - and the communication parameters of the machine.
  - An algorithm must therefore be analysed in the context of the underlying platform.
    - The same applies to the memory parallel complexity

- Parallel System
  - A parallel system is a combination of a parallel algorithm and an underlying parallel platform.

- Models of platforms
  - A unique model (as the Turing machine for sequential computing) does not exist for parallel computing

PRAM: a Ideal Parallel Computer

- PRAM : Parallel Random Access machine
  - a natural extension of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine (PRAM)
  - consists of **p** processors and a global memory of unbounded size that is uniformly accessible to all processors (UMA)
  - processors share a common clock but may execute different instructions in each cycle
- Handling of simultaneous memory accesses
  - Exclusive-read, exclusive-write (EREW)
  - Concurrent-read, exclusive-write (CREW)
  - Exclusive-read, concurrent-write (ERCW)
  - Concurrent-read, concurrent-write (CRCW)
- Suitable to model shared memory parallel architecture

What does concurrent write mean, anyway?

Common:	write only if all values are identical.
Arbitrary:	write the data from a randomly selected processor.
Priority:	follow a predetermined priority order.
Sum:	write the sum of all data items.

Constant time sorting algorithm Θ(1) !

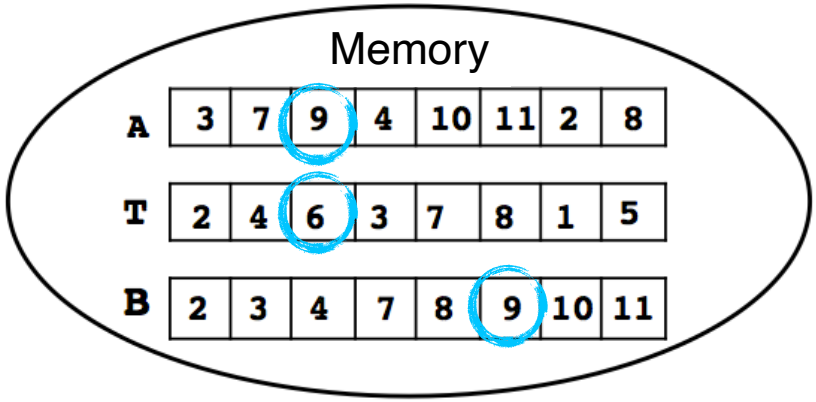
- Goal: to sort a vector of integers of size N
- This algorithm run on a PRAM CRCW with the following writing policy:
  - When several processors write in the same memory location the result is the sum of the written values by each processors.
- The algorithm needs N^2 processors (N is the size of the vector to sort)
- Data in shared memory
  - the vector A to sort of size N
  - two non initialised vectors B and T of size N
- The algorithm is
  - Each processor p executes

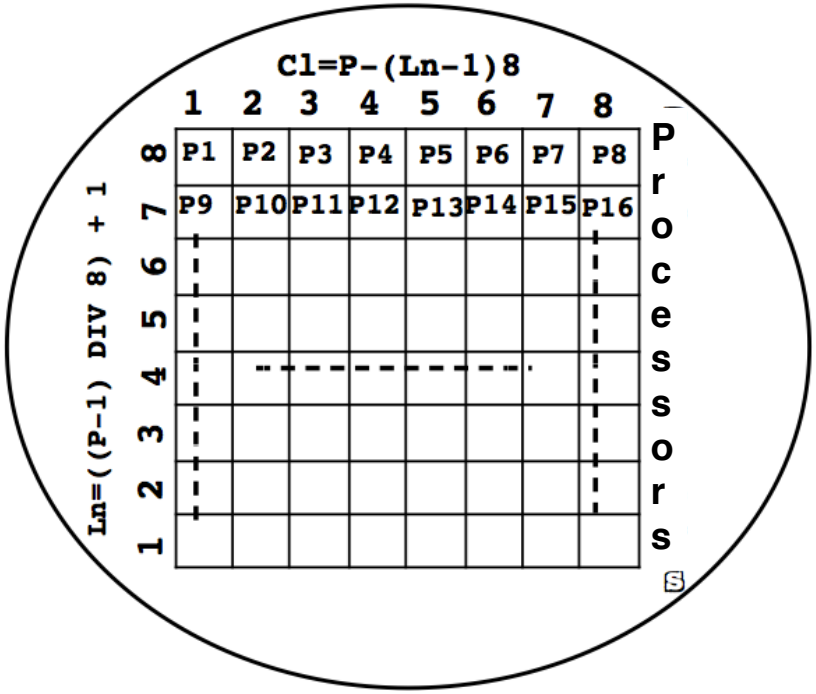
    -where p = 1..N^2, is the id. of the processors

```
procedure Tri_Constant()
begin
   int Ln = ((p-1)/N)+1
   int Cl = p - (Ln-1)*N
   if A(Ln)≤A(Cl) then T(Ln):=1;
   if p≤N then B(T(Ln)) := A(Ln);
end Tri_Constant;
```

Explanations

- The element i of the vector T contains the index in the sorted array B of the element i of the array to sort A.
  - Example :
    - T[3] = 6 because the position of A[3]=9 is 6 in B (B[6]=9=A[3]).
  - The sum of the values 1 written by processors of column i, is equal to the number of values in A which are smaller or equal to A[i]
    - if A(Ln)≤A(Cl) then T(Ln):=1;

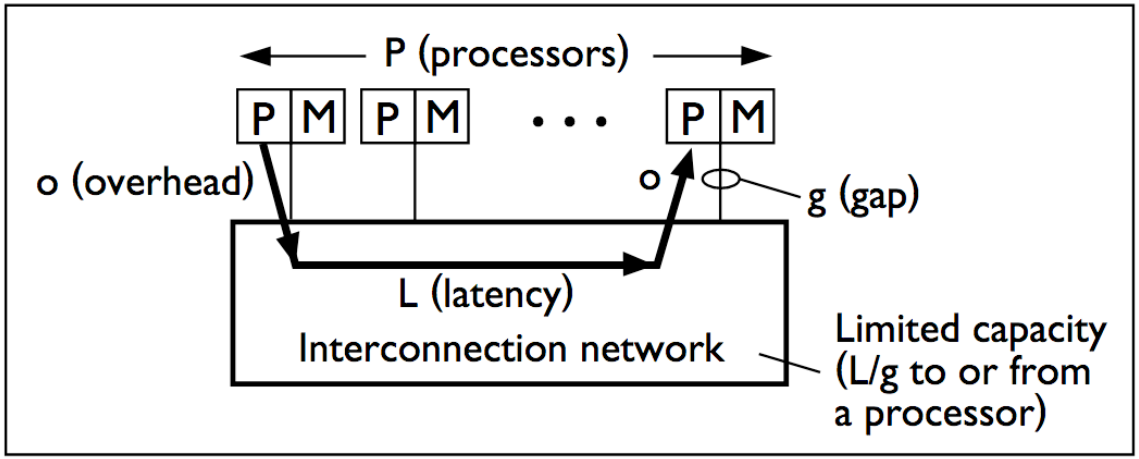




LogP Model

- The **LogP** model (Culler et al. 1993) enables modelling of overlap by modelling the cost of sending a message of one 'datum' in terms of
  - **L** : network latency cost (processor free)
  - **o** : sender/receiver sequential overhead (processor occupied)
  - **g** ≥ o : the necessary gap between two message sending/receiving
    - This parameter can be interpreted as the inverse of the bandwidth of the communication channel between processors
  - **P** : the number of processors
- No consensus on topology
- No consensus on programming model

Nothing to do with logarithm but with the names of the parameters of the model

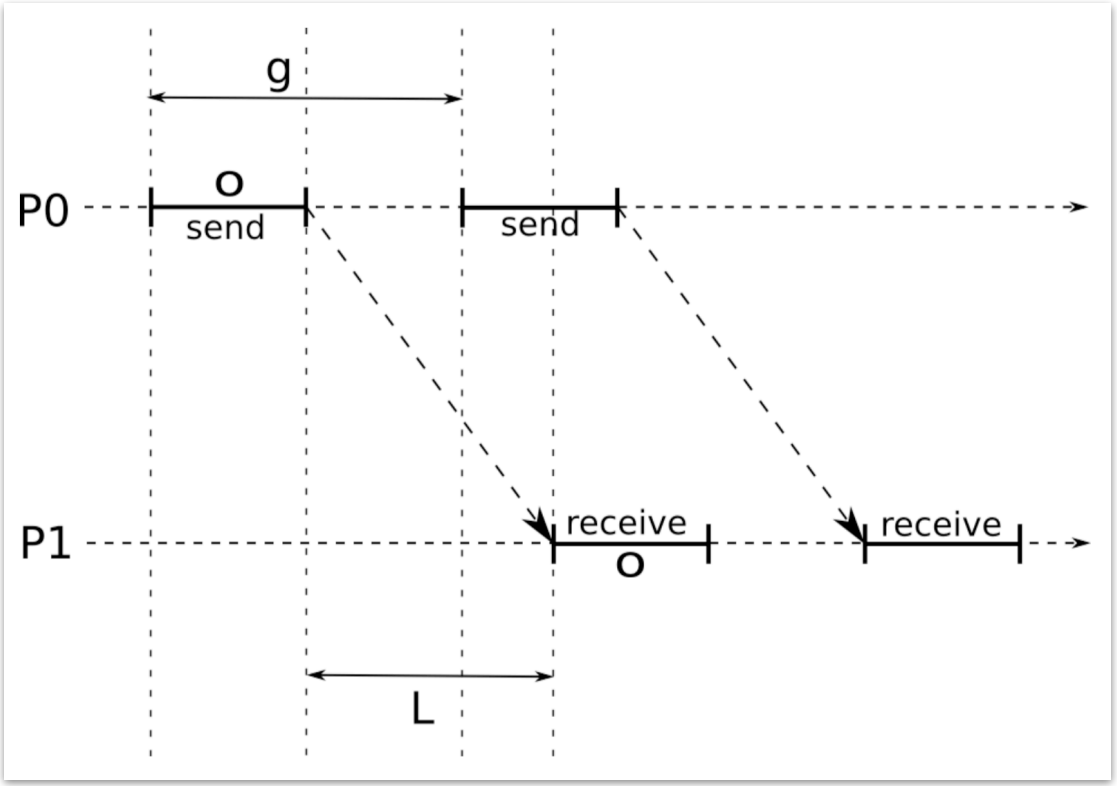


David Culler, Richard Karp, David Patterson et Abhijit Sahay, «LogP: towards a realistic model of parallel computation, LogP: towards a realistic model of parallel computation», ACM SIGPLAN Notices, vol. 28, no 7, 1er août 1993, p. 1, 1–12,

Messaging in the LogP model

- the LogP communication cost for sending a message of s packets is

$$T(s) = 2o + L + (s-1)g$$



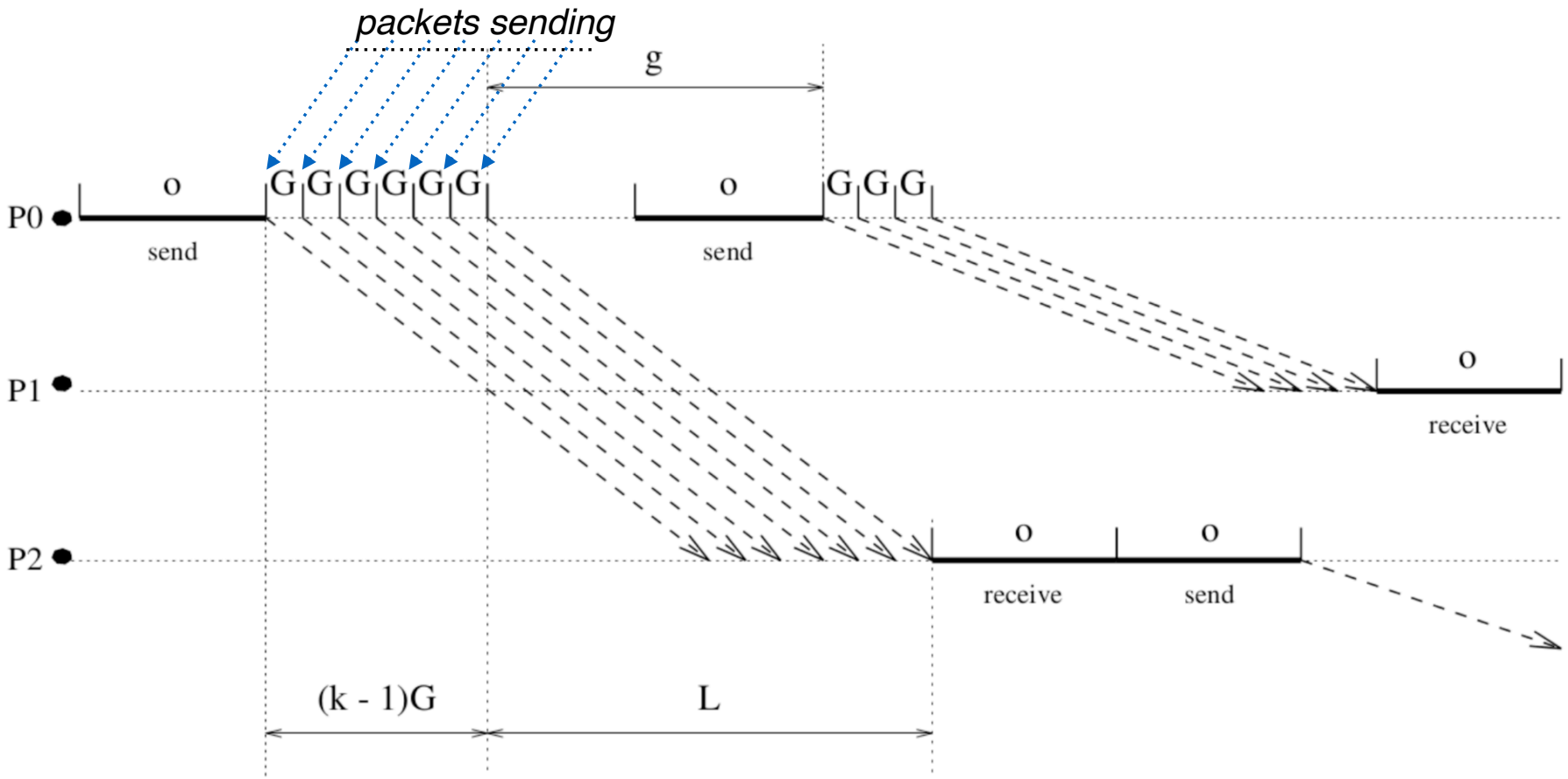
LogP model: problem

- The LogP model parameter *g* is associated with the packet size
  - this injection rate implies a fixed-sized packet can be sent anywhere after a time interval of *g*
  - modern computer networks do not have a small fixed packet size and achieve higher bandwidth for large messages

- To solve this problem we can use a different *g* (we will call *G*) value for large messages
  - model LogGP
    - Introduced by Alexandrov et al. 1997
      - "LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation", Albert Alexandrov, Mihai F.Ionescu, Klaus E.Schauser, Chris Scheiman, Journal of Parallel and Distributed Computing, Volume 44, Issue 1, 10 July 1997, Pages 71-79
  - the LogGP time for sending a large message of s packets is *(similar to LogP)*

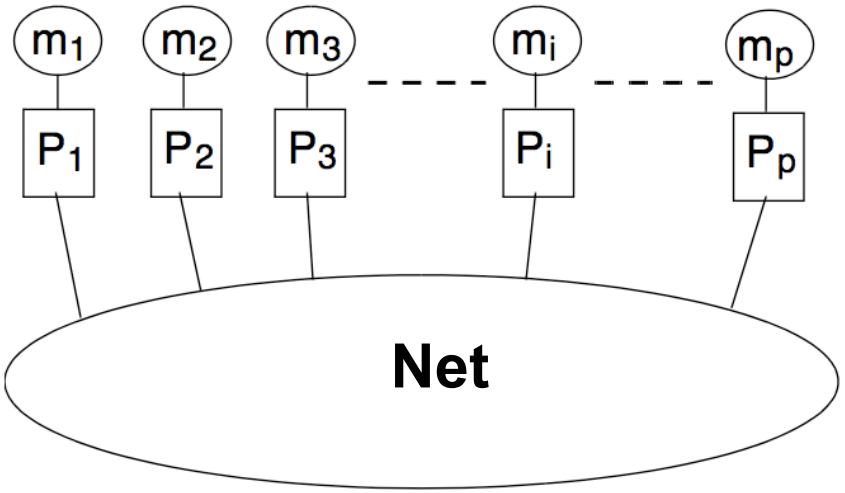
$$T(s) = 2o + L + (s-1)G$$

Messaging in the LogGP model



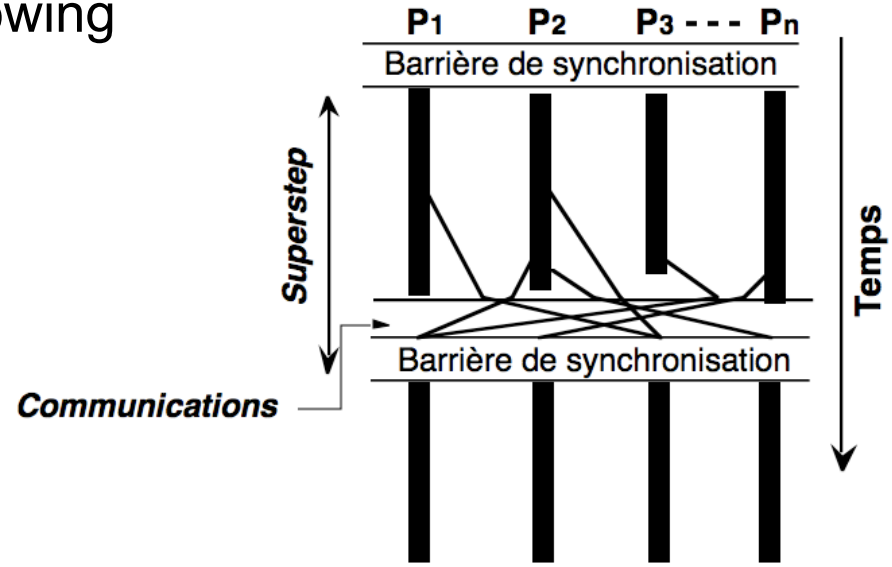
BSP: The Bulk Synchronous Parallel

- BSP: The Bulk Synchronous Parallel abstract computer
  - Introduced in 1990 by L.G.Valiant, University of Harvard
  - Take into account the communication and synchronization costs
  - More suitable than PRAM for to model distributed memory computers
  - Is composed of:
    - A set pairs: processors-local memory
    - A communication network
    - A efficient synchronization mechanism



BSP: Execution model

- The execution model of the BSP is the following
  - It is a sequence of **supersteps**
  - During a superstep each processor can
    - do any local computation
    - send requests to other processors
    - send data to other processors



- At the **end of each supersteps** there is a synchronization phase between all processors where all communication are effectively realized.

Performance Metrics for Parallel Systems

- Execution Time
  - Sequential time T_1
    - the time elapsed between the beginning and the end of the execution on a sequential computer
      OR
    - number of steps on the Turing machine
  - Parallel runtime $T_{//p}$
    - the time that elapses from the moment the first processor starts to the moment the last processor finishes execution (wall-clock time)
      OR
    - number of steps on an adequate model of parallel machine

- Reminder:
  - Speedup $S(P) = T_1/T_{//p}$
    - the ratio of the serial runtime of **the best (or the same) sequential algorithm** for solving a problem to the time taken by the parallel algorithm to solve the same problem on a parallel computer with p **identical** processing elements each processor having the same computing power than the sequential one.
  - Efficiency $E(p) = S/p <= T_1/(p \cdot T_{//p})$

Example

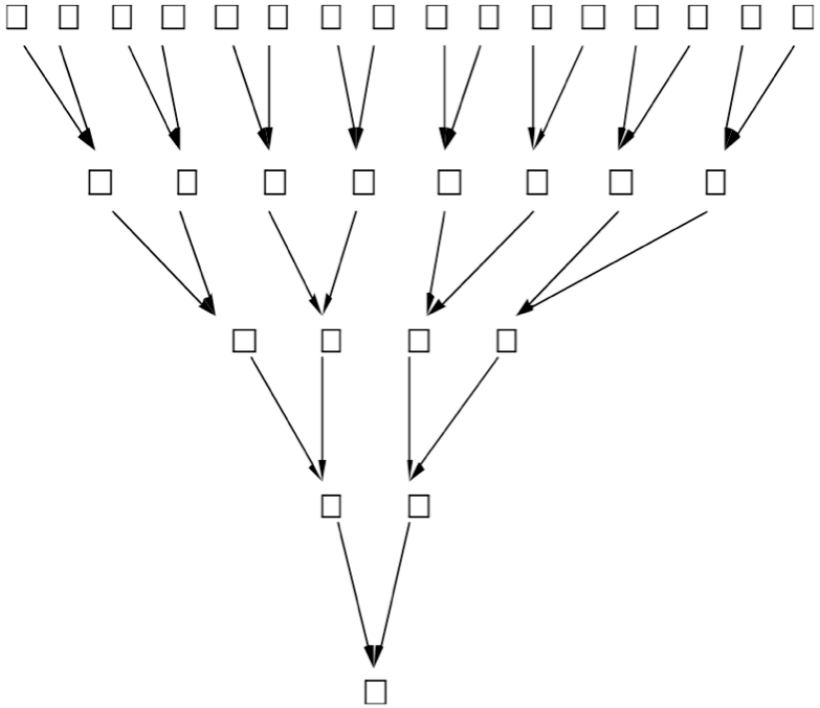
- Problem
  - to compute the maximum of N numbers
- Sequential Algorithm

```
FOR i IN 2..N LOOP
   IF (a[i]>a[1]) THEN a[1]=a[i]; -- The final result is in a[1]
END LOOP
```

  - The complexity is : O(T)= N
  - There is a dependency between each iteration !!
- This dependency is artificial
  - It is a consequence of the programming model used to implement the algorithm
    - "FOR" is a sequential loop
  - In fact, the comparisons can be made in any order (the max operator is commutative and associative)

Parallel Algorithm: basic idea

- We start by doing N/2 comparisons in parallel on N/2 different pairs of the initial numbers
- We repeat this process
  - At this stage, to simplify, we made the hypothesis that $N = 2^m$

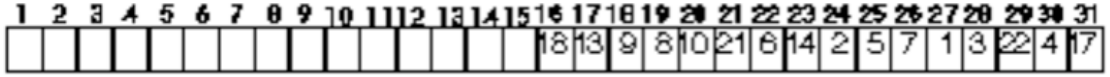


Sequential implementation

- Here is a sequential implementation of this algorithm:

```
FOR i IN REVERSE 0..m-1 LOOP          // i: 3->0, problem size = 2^m
   FOR j IN REVERSE 2^i..2^(i+1)-1 LOOP
     a[j]:=max(a[2j],a[2j+1]);
   END LOOP;
END LOOP
```

- The final result is in `a[1]`



i	j	Indices comparés		Indice du résultat	$j-2^i$
3	15	30-31	->	15	7
	14	28-29	->	14	6
	13	26-27	->	13	5
	12	24-25	->	12	4
	11	22-23	->	11	3
	10	20-21	->	10	2
	9	18-19	->	9	1
	8	16-17	->	8	0
2	7	14-15	->	7	3
	6	12-13	->	6	2
	5	10-11	->	5	1
	4	8-9	->	4	0
1	3	6-7	->	3	1
	2	4-5	->	2	0
0	1	2-3	->	1	0

Remarks:

1. The outer loop (on i) is executed $m=\text{Log}_2(N)$ times.
2. The iterations of the inner loop (on j) are independent because they access different memory locations. This loop can be parallelized.

Implementation on a PRAM

- Constatation
  - The value $j-2^i$ can be used to identify the processing element that performs the comparison
    - we put $p=j-2^i$ where p is the processing element identifier.
  - thus
    - $j=p+2^i$
  - We can now implement this algorithm on a PRAM having N/2 processors

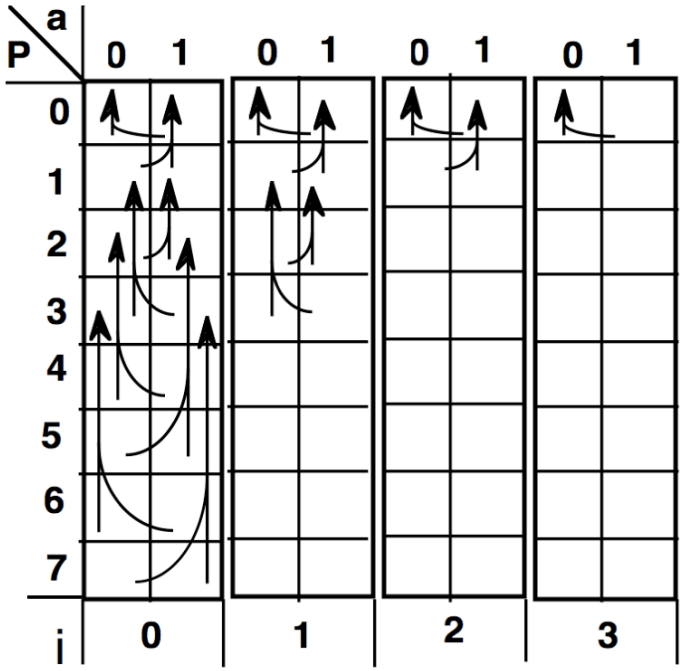
```
FOR i IN REVERSE 0..m-1 LOOP      // m = Log2(N)
   IF p<2^i THEN                   // p = id. of the current processing element
     IF a[2(p+2^i)]>a[2(p+2^i)+1]
     THEN a[p+2^i]=a[2(p+2^i)];
     ELSE a[p+2^i]=a[2(p+2^i)+1];
     ENDIF;
   ENDIF
 END LOOP   -- the final result is in a[1]
```

    - we can execute this program in O(log(N)) steps, each step i being executed in O(1) on 2^i processors.
    - Thus: $O(T_{//p})= \log(N)$
- Speedup S(N)
  - O(S(N)) = N/log(N)

Implementation on a BSP

- Additional difficulty
  - Data distribution
    - No common memory
  - Let's assume that initially each processor has in its local memory the two first numbers to add
    - The initial distribution is already done

```
▪FOR i IN 0..m-1 LOOP -- m = Log2(N)
   bsp_sstep;
   IF p<2m-i THEN
     IF a[0]>a[1]
     THEN v=a[0]; ELSE v=a[1];
     ENDIF;
     bsp_store(v,a[p mod 2],p div 2);
   bsp_sstep_end;
 END LOOP
```



  - The instruction `bsp_store(v,m,p)` stores the value `v` in the memory position `m` of the processor `p`.
    - $O(T_{//p}) = \log N$

Implementation on a LogP

- Below is the pseudo code of the implementation in the LogP model
  - We assume
    - -FIFO channels
    - -Times to execute line 1 = t1, line 2 = t2, line 3 = t3
    - -size of a integer b bytes

```
1. FOR i IN 0..m-1 LOOP    -- m = Log2(N)
2.   IF p<2^(m-i) THEN
3.     IF a[0]>a[1] v=a[0] ELSE v=a[1]; ENDIF;
4.    SEND v TO p div 2;
5. END LOOP
```

- Total execution time: ?????? -> exercice

Parallel complexity

- The study of the complexity of algorithms is necessary when developing performant algorithms.
- In the case of parallel algorithms, the notion of “performance” is often related to the performance of a sequential reference algorithm that solves the same problem
- Notations:
  - T_1 = execution time of the sequential reference algorithm.
  - $T_{//}$ = execution time of the parallel algorithm (with a non specified number of processing elements).
  - $T_{//p}$ = execution time of the parallel algorithm using p processors.

Parallel complexity: definitions

- Definition

  - The time $T_{//}(A_N)$ is the parallel execution time of the **algorithm A** with a **data of size N**. This time is defined as the numbers of steps used by the PRAM machine (or any suitable model) to execute the algorithm

  - The surface $H(A_N)$ used by the algorithm A with a data size of N is defined as the maximal number of processors necessary to the PRAM machine to execute the algorithm.

  - The work $W(A_N)$ of the algorithm A with a data size of N is defined by the relation:
    - $W(A_N)=T_{//}(A_N) \cdot H(A_N)$.

Parallel algorithms efficient and optimal

- A parallel algorithm is said **efficient** if and only if:
  - $O(T_{//}(A_N))=Log^k(N)$
  - In other words, the parallel time complexity is polylogarithmic. This means that it is better than any sequential algorithm
  - => $O(W(A_n)) = O(T_1 \cdot Log^k(N))$

- A parallel algorithm is said **optimal** if and only if:
  - $O(T_{//}(A_N))=Log^k(N)$ and
  - $O(W(A_N))=O(T_1(A_N))$
  - A parallel algorithm is optimal if its time complexity is polylogarithmic AND if it performs the same work than the sequential algorithm

The algorithm for the maximum

- To run the algorithm we need N/2 processing elements
  - Time complexity: $O(T_p(A_N)) = Log(N)$
  - Work complexity: $O(W(A_N)) = N \cdot Log(N)$

- BUT the time complexity of the sequential algorithm *(which is equal to the work complexity because H=p=1)* is $O(T_1) =$ **N**.

  - This algorithm is efficient but NOT optimal !

    -$O(W(A_N)) > O(T_1)$

Optimality

- Can we make this algorithm optimal ?
  - It is not optimal because its work complexity is bigger than the work complexity of the sequential algorithm
    - $O(W(A_N)) = N \cdot Log(N) > O(T_1(A_N)) = N$
  - There is a Log(N) factor between these two complexities.
    The problem comes from the surface which uses N/2 processors: $O(H(A_n))=N$
    - It is during the first step that we need the maximal number of processors: N/2.
    - Lets make a first step using (N/2)/(Log(N)/2)=N/Log(N) processors where each processor computes sequentially the maximum of N/(N/Log(N))=Log(N) values
      - The time for the first step is now: Log(N), after the step 1 we still have N/Log(N) values to compare (there are enough p.e.)
      - The number of steps is O(log(N/Log(N)) = O(Log(N)-Log(Log(N))) = O(Log(N)) (not changed)
    - Surface: $O(H_{//}(A_N))=N/Log(N)$
    - Time: $O(T_{//}(A_N)) = O(Log(N) + Log(N)) = O(2 \cdot Log(N))$ = **Log(N)**
    - Work: $O(W(A_N)) = O((N/Log(N)) \cdot Log(N)$ = **N**
  - This algorithm is optimal !… *but we have a factor 2 for $T_{//} = 2 \cdot log(N)$*

Questions... ?