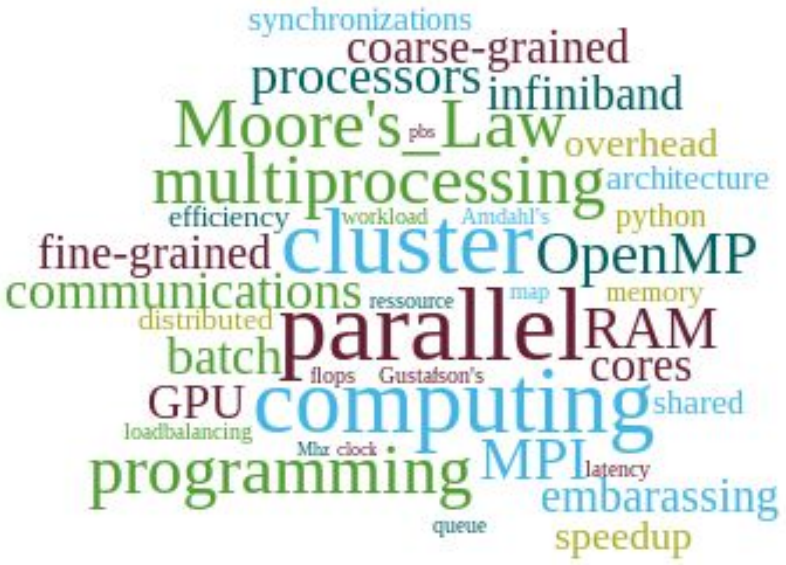
# Parallel computing at LAM

Jean-Charles Lambert
Sergey Rodionov

**Online document : https://goo.gl/n23DPx**

# Outline

I. General presentation
II. Theoretical considerations
III. Practical work

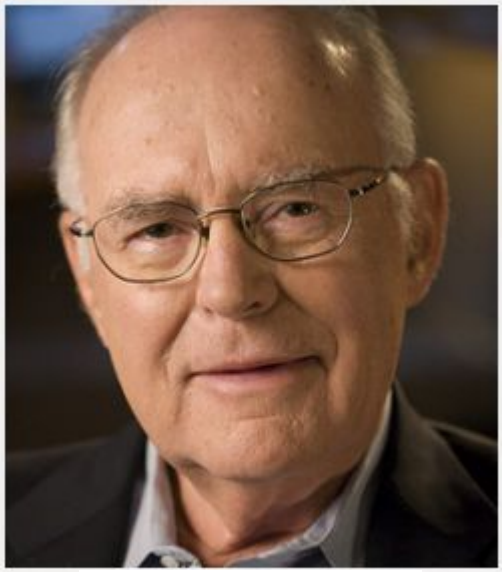


**PART I :**

# GENERAL PRESENTATION

# HISTORY

# Moore's LAW

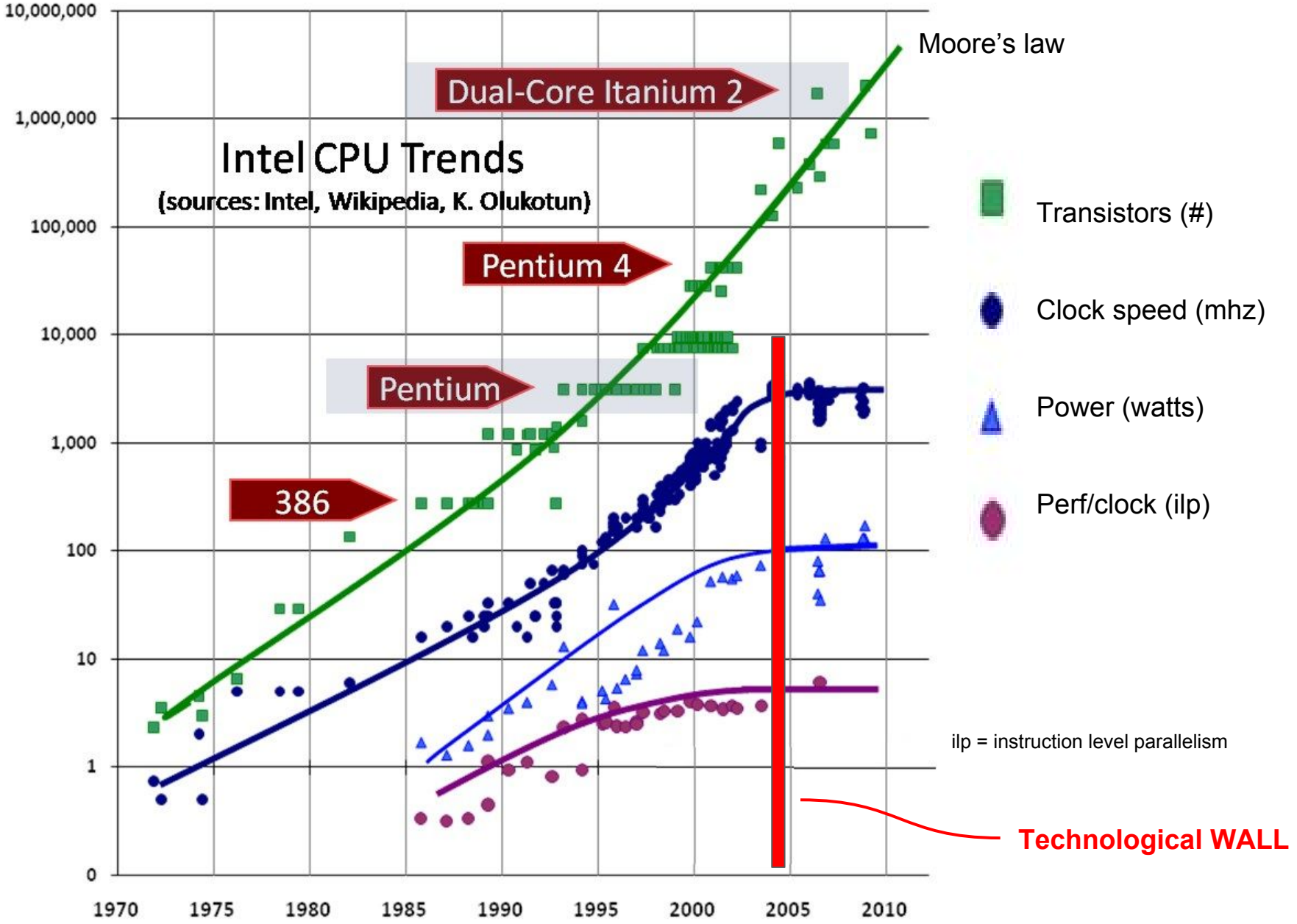


Gordon E. Moore

Moore's Law is a computing term which originated around 1970:

the simplified version of this law states that processor speeds, or overall processing power for computers will double every two years.

The "exact" version says that the number of transistors on an affordable CPU would double every two years


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Moore's law
Dual-Core Itanium 2
Pentium 4
Pentium
386
Transistors (#)
Clock speed (mhz)
Power (watts)
Perf/clock (ilp)
ilp = instruction level parallelism
Technological WALL
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010

**Processor speed ⟹ technological WALL**

Technological WALL for increasing operating processor frequency

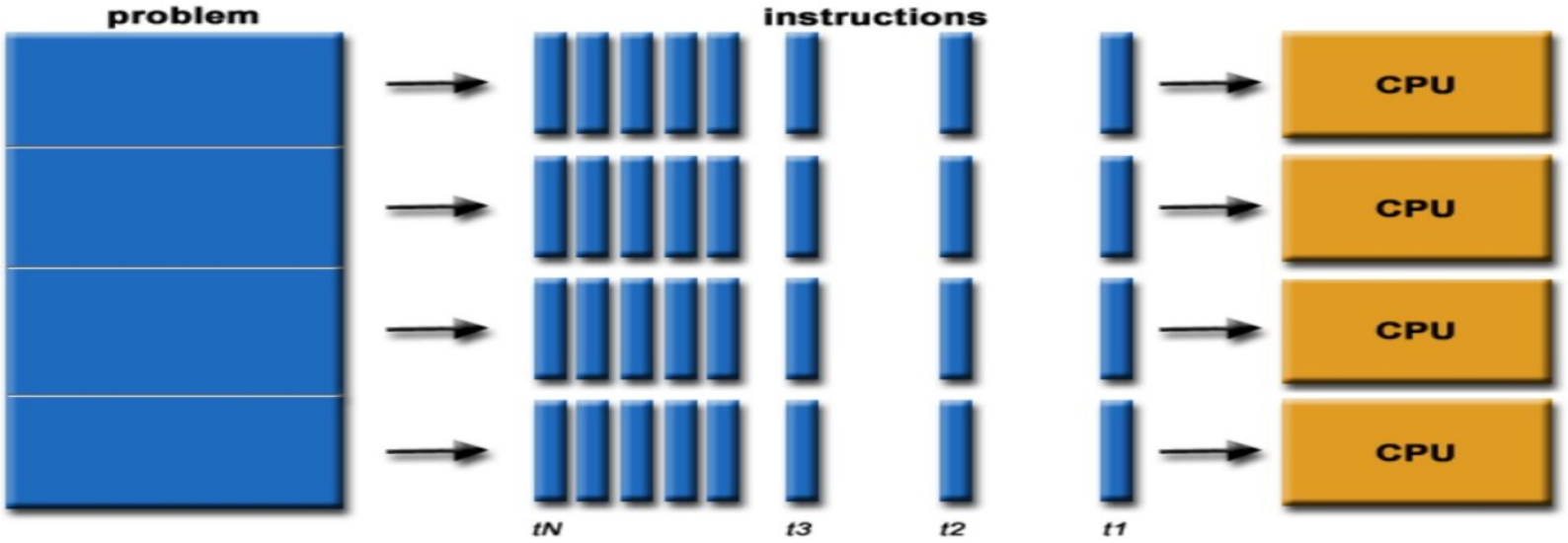
- **Memory Wall** : gap between processor speed vs memory speed
- **ILP (Instruction Level Parallelism) Wall** : difficulty to find enough ILP
- **Power Wall** : exponential increase of power vs factorial increase of speed

**This is why, we MUST use multiple cores to speed up computations.**

**Here comes parallel computing !**

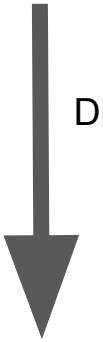
Definition :

**Parallel computing** is the use of two or more processors (cores, computers) in combination to solve a single problem



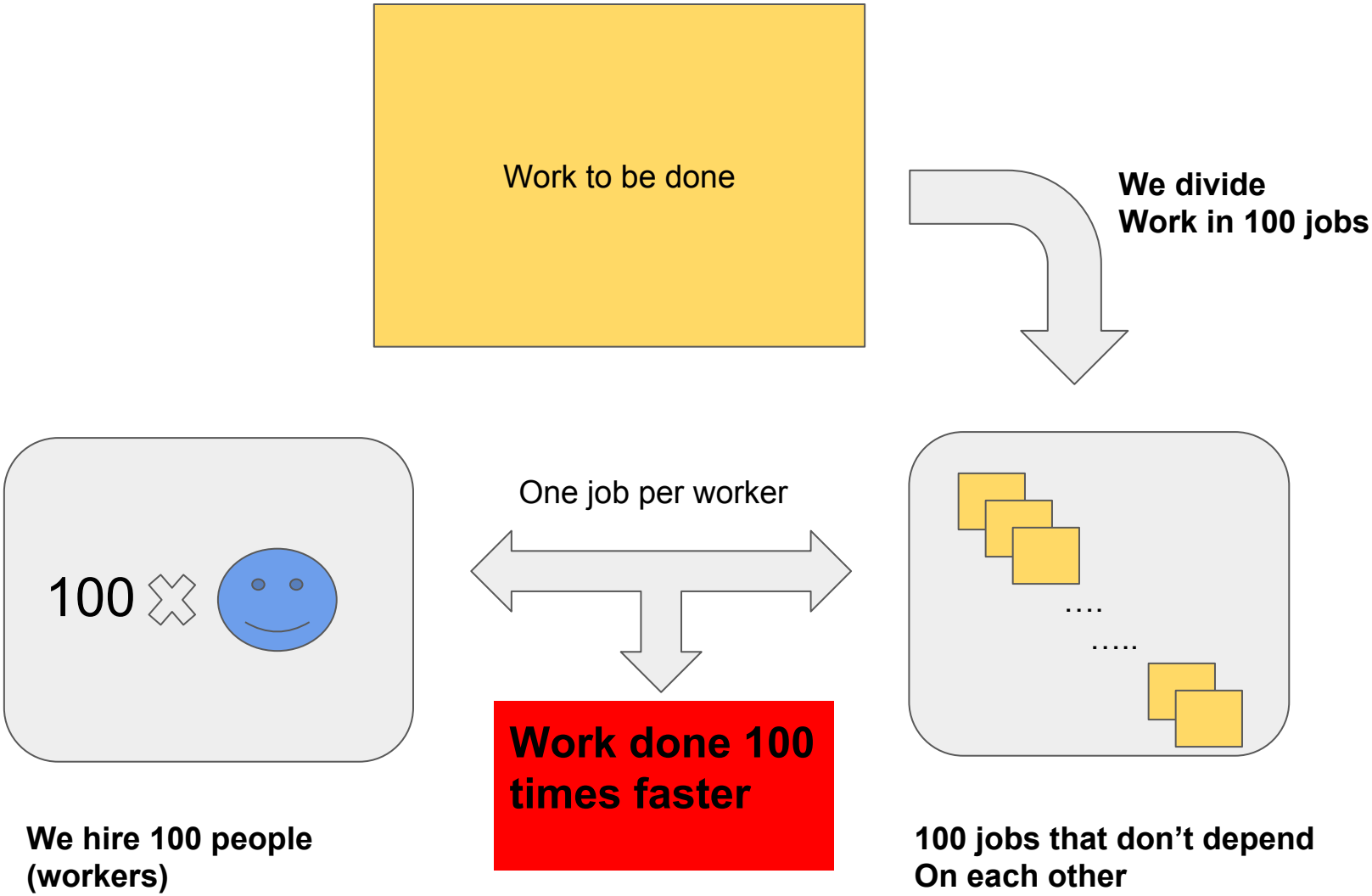
# Three (mains) types of parallelism

- Embarrassing
- Coarse-grained
- Fine-grained



Difficulty to implement

# Embarrassingly parallel workload



# Coarse-grained -> ex: Simple queue multiple server

**queue (data)**

**checkout (processors)**

fnac Centre bourse

# Fine-Grained : Concurrency, synchronisation and communication

**Building a house can be divided in many jobs**

- Fondations
- Plumbing
- Electricity
- Roof

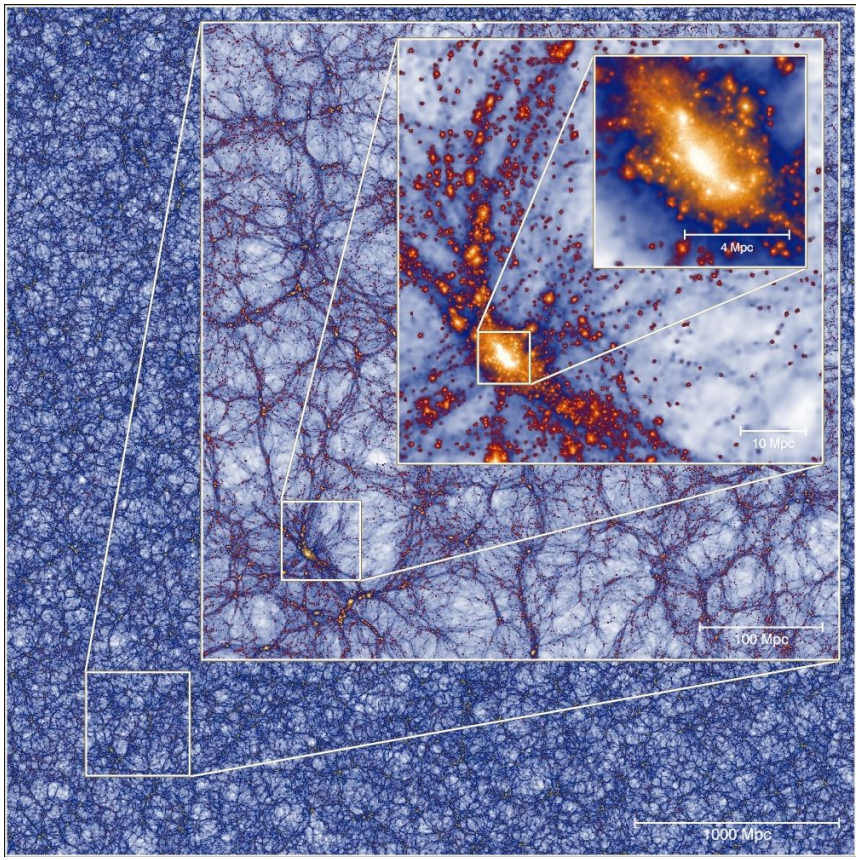
➔ Many jobs can be run in parallel
➔ Some have specific orderings
  ◆ fondations before electricity

Some workers have to wait….

# Why do parallel computing ?

- To get results faster
- Solve problems that don’t fit on a single CPU’s memory space
- Solve problems that can’t be solved in a reasonable time
- Solve larger problems faster
- Run multiple test cases / analysis at the same time

# Millennium-XXL project Project : simulating the Galaxy Population in Dark Energy Universes

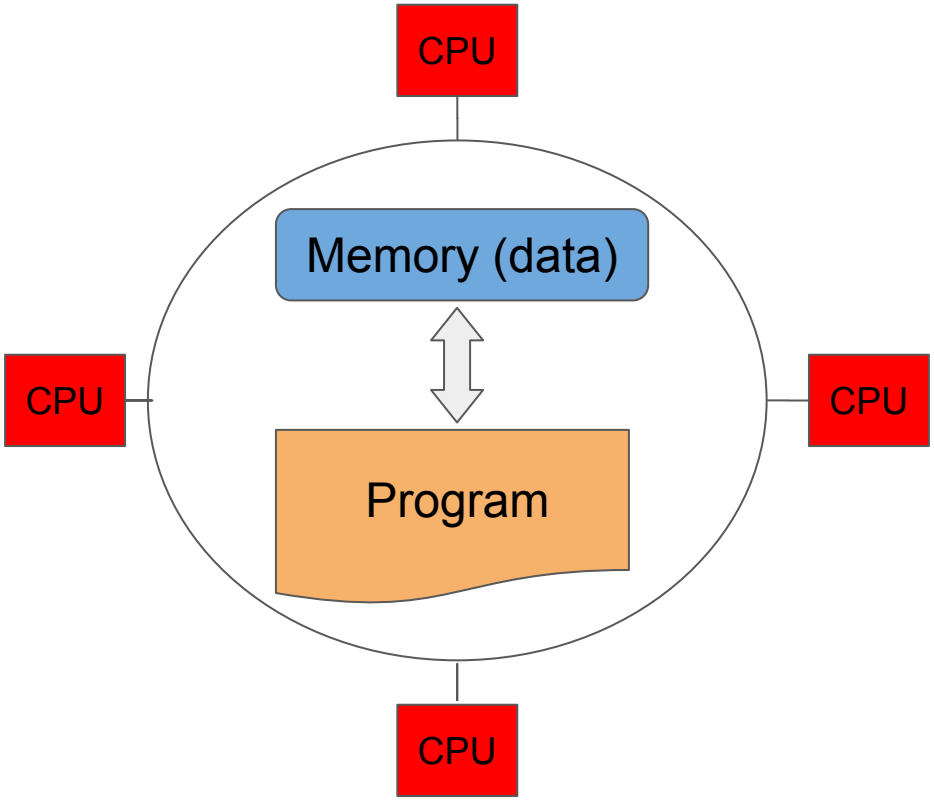



- 300 billions particles
- 12000 cpus
- 30 TB RAM
- 100 TB of data products

Equivalent of 300 years of CPU time !!!

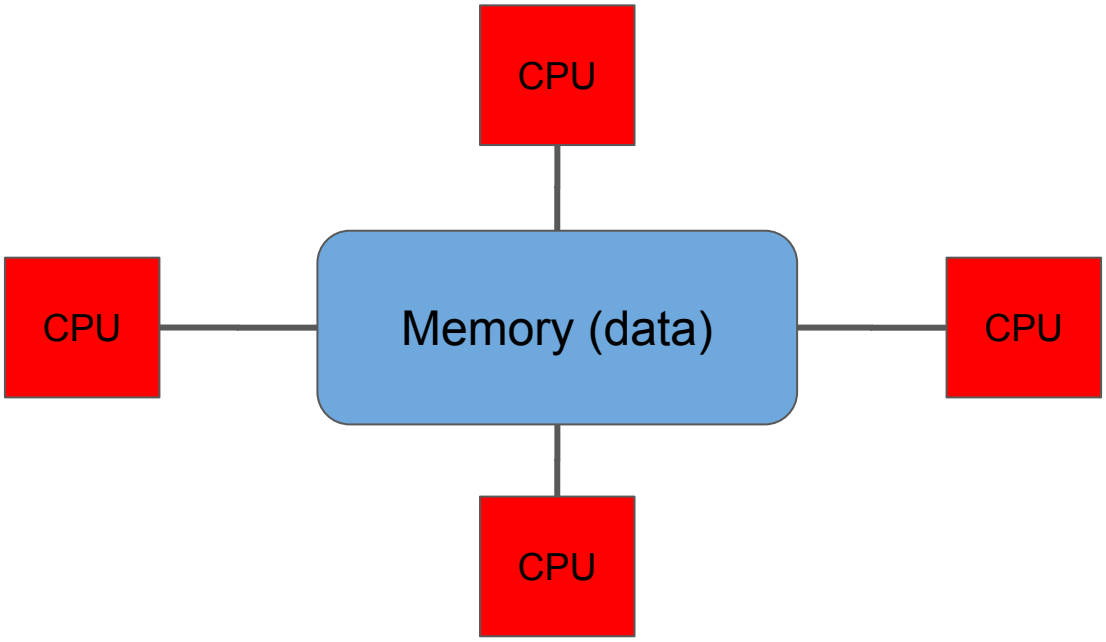
# Memory and communication

## Fine grain parallel program design



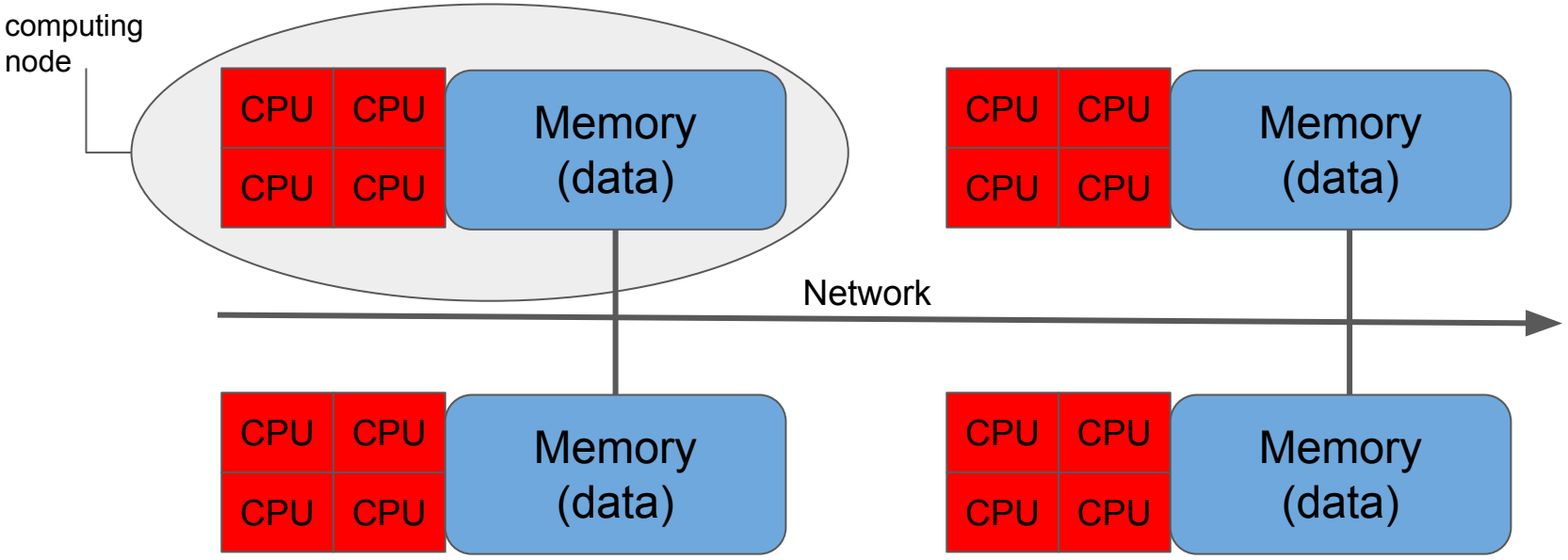

- same program runs on many CPU
- get data from memory space
- exchange data

## Shared memory




On your laptop/desktop or in one node of a computing cluster, memory is shared between all processors/cores

## Distributed shared memory




On a computing cluster, memory is distributed between nodes, along a fast network with low latency, and shared inside nodes

# Parallel API (Application Program Interface)

## OpenMP (Open Multi-Processing)

- C,C++ and Fortran language
- Implicit parallelism
- set of compiler directives
- quite easy to implement
  - no big modifications of the code
  - Same workflow
- restricted to shared memory

## MPI (Message Passing Interface)

- C,C++,Fortran language, python
- explicit parallelism
- explicit calls (send and receive data)
- hard to implement
  - code must be re written
  - explicit data moving
- shared and distributed memory

# OpenMP vs MPI : a simple example

```
Array A[N]

For i=1 to N
      A [ i] = i * 2
```

Objective :

- split “**for**” loop
- distribute **A[ ] = i *2** computation among processors

## Sequential version

```
#define NMAX 100000
int main(int argc, char **argv)
{
    int a[NMAX];


    for (int i = 0; i < NMAX; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

# OpenMP version

```
#define NMAX 100000
int main(int argc, char **argv)
{
    int a[NMAX];

    #pragma omp parallel for
    for (int i = 0; i < NMAX; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

**`#pragma omp parallel for`**

- create as many threads as there are processors
- split up loop iterations among these threads

# MPI version

```
#include <mpi.h>   // PROVIDES THE BASIC MPI DEFINITION AND TYPES
#include <stdio.h>
#include <string.h>
#define NMAX 24
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int main(int argc, char **argv) {
  int i, my_rank, partner, size,a[NMAX],chunk,istart,istop;
  MPI_Status stat;

  MPI_Init(&argc, &argv);                  // START MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // DETERMINE RANK OF THIS PROCESSOR
  MPI_Comm_size(MPI_COMM_WORLD, &size);    // DETERMINE TOTAL NUMBER OF PROCESSORS

  chunk=NMAX/size;  istart=(chunk*my_rank);  istop=MIN(istart+chunk,NMAX);

  for (i=istart; i<istop; i++)  // EVERY PROCESSOR COMPUTE ONE CHUNK OF THE ARRAY
    a[i] = 2 * i;

  if (my_rank == 0) { // master GATHER ALL RESULTS
    for (partner = 1; partner < size; partner++){
      istart=(chunk*partner); istop=MIN(istart+chunk,NMAX);
      MPI_Recv(a + istart ,istop-istart, MPI_INT, partner, 1, MPI_COMM_WORLD, &stat);
    }
    for (i=0; i<NMAX; i++)
      fprintf(stderr,"a[%5d] = %8d\n",i,a[i]);
  }
  else {             // ALL processors except the master
    MPI_Send(a+istart,istop-istart , MPI_INT, 0,1,MPI_COMM_WORLD);
  }
  MPI_Finalize();  // EXIT MPI
}
```

# Parallel computing architecture

## Where to get parallel computers ?

- Smartphone ( 2 to 8 cores)
- Laptop
  - Macbook pro (4 cores i7 2.5Ghz)
  - HP zbook linux (4 cores i7/xeon 2.7Ghz)
- Desktop workstations
  - 10 to 96 cores
- Computing cluster
  - 100 to thousand cores
  - LAM ( ~ 400 cores)
- GPU (Graphic Processor units)
  - 500 to 4000 shader units (dedicated cores)


TOP of the line Intel processor :

Intel Xeon Processor E7-8890 V4

Lithography : 14 nm
# of cores : **24**
Recommend customer price : 7200 $

# TOP 500 supercomputers

| Rank | Site | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|---|---|---|---|---|---|---|
| 1 | National Supercomputing Center in Wuxi<br>China | **Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway<br>NRCPC | 10,649,600 | 93,014.6 | 125,435.9 | 15,371 |
| 2 | National Super Computer Center in Guangzhou<br>China | **Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5<br>Ex<br>NU | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 3 | DOE/SC/Oak Ridge National Laboratory<br>United States | Ti<br>16<br>in<br>Cr | | | | 8,209 |
| 4 | DOE/NNSA/LLNL<br>United States | Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom<br>IBM | | | | 7,890 |
| 5 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect<br>Fujitsu | 705,024 | 10,510.0 | 11,280.4 | 12,660 |
| 6 | DOE/SC/Argonne National Laboratory<br>United States | **Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM | 786,432 | 8,586.6 | 10,066.3 | 3,945 |
| 7 | DOE/NNSA/LANL/SNL<br>United States | **Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect | 301,056 | 8,100.9 | 11,078.9 | |

**FLOPS** = Floating Operations Per Second

10^17 FLOPS = 100 PetaFlops !!!

First french supercomputer private (ranked 11)

| | | | | | | |
|---|---|---|---|---|---|---|
| 11 | Total Exploration Production<br>France | **Pangea** - SGI ICE X, Xeon Xeon E5-2670/ E5-2680v3 12C 2.5GHz, Infiniband FDR<br>SGI | 220,800 | 5,283.1 | 6,712.3 | 4,150 |

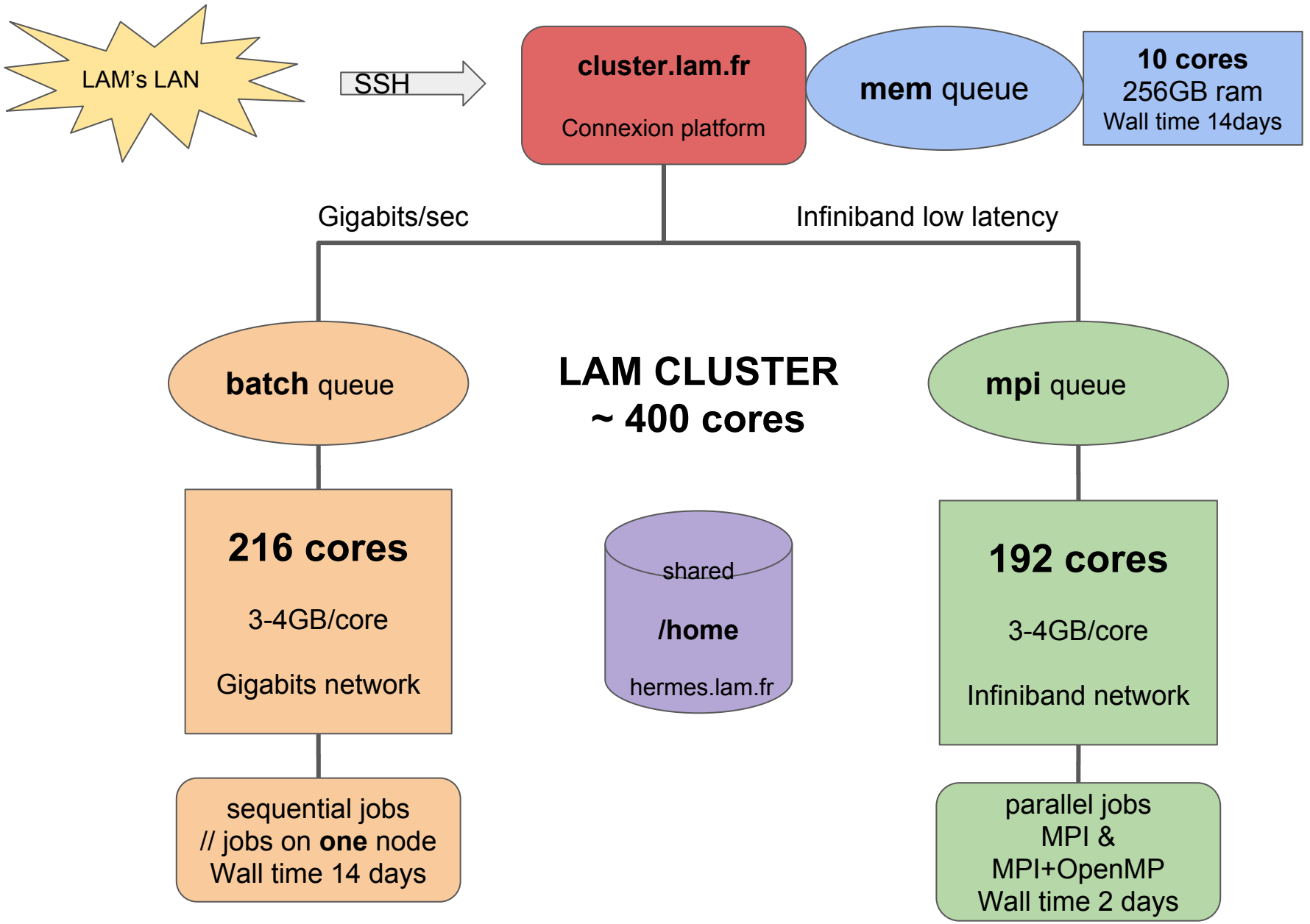
First french supercomputer public [CINES] (ranked 53)

| | | | | | | |
|---|---|---|---|---|---|---|
| 53 | Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES)<br>France | **Occigen** - bullx DLC, Xeon E5-2690v3 12C 2.6GHz, Infiniband FDR<br>Bull, Atos Group | 50,544 | 1,628.8 | 2,102.6 | 935 |

[CEA/TGCC] (ranked 62)

| | | | | | | |
|---|---|---|---|---|---|---|
| 62 | CEA/TGCC-GENCI<br>France | **Curie thin nodes** - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR<br>Bull, Atos Group | 77,184 | 1,359.0 | 1,667.2 | 2,251 |

# LAM’s cluster


LAM's LAN
SSH
cluster.lam.fr
Connexion platform
mem queue
10 cores
256GB ram
Wall time 14days
Gigabits/sec
Infiniband low latency
batch queue
LAM CLUSTER
~ 400 cores
mpi queue
216 cores
3-4GB/core
Gigabits network
shared
/home
hermes.lam.fr
192 cores
3-4GB/core
Infiniband network
sequential jobs
// jobs on one node
Wall time 14 days
parallel jobs
MPI &
MPI+OpenMP
Wall time 2 days

# Job scheduling

LAM cluster is managed by **PBS torque** (distributed resource manager)

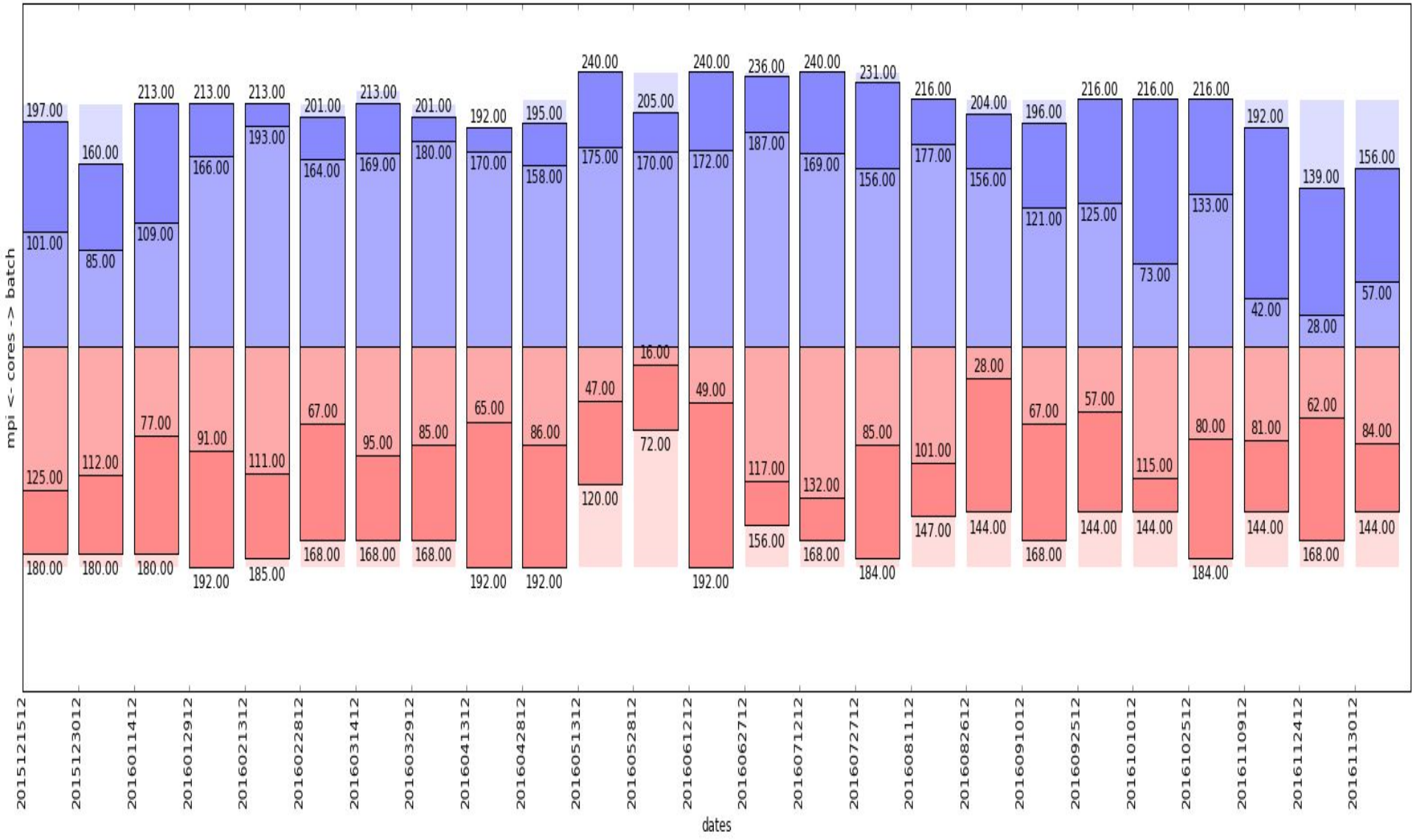
- based on simple shell script
- allocate computational task among available resources
  - rules for requested cpu,ram,walltime
- Submission via **qsub** command on specific queue
  - qsub -q batch myjob.sh
  - qsub -q mpi myjob.sh
  - qsub -q mem myjob.sh

Example : **myjob.sh**

```
#!/bin/sh
#
#PBS -N MYJOBname
#PBS -l nodes=1:ppn=1     # request 1 node and 1 core
#PBS -l walltime=12:00:00 # 12 hours wall clock time
#PBS -M jean-charles.lambert@lam.fr # mail send to this adress after run completion
#PBS -o "run-log.txt"   # standard output redirection
#PBS -e "run-err.txt"   # standard error redirection

# change to submission jobs directory
cd $PBS_O_WORKDIR
my_program  # start my_program
```

# LAM's cluster cores occupation (YEAR 2016)



# Parallel programs running at LAM

- **Lenstools** : gravitational lensing code (OpenMP)
- **Gadget3** : nbody simulation program (MPI) massively parallel
- **Ramses** : nbody simulation program (MPI) massively parallel
- **Proto planetary formation** : Hybrid OpenMP + MPI program
- **Cosmic rays** : OpenMP program
- **GInemo2** : 3D visualisation program (GPU shaders)

# People behind LAM's cluster

**System administration and software management**
- Thomas Fenouillet
- Jean-Charles Lambert

**Parallels code development and advisories**
- Sergey Rodionov
- Jean-Charles Lambert

**Hardware support**
- Vincent Herero

**File server management and network**
- Julien Lecubin
- Adrien Malgoyre

ANR AGENCE NATIONALE DE LA RECHERCHE

LAM's cluster
EXPANSION

- Increase overall computing power
- Easy to maintain
- Easy to expand

**PART II :**

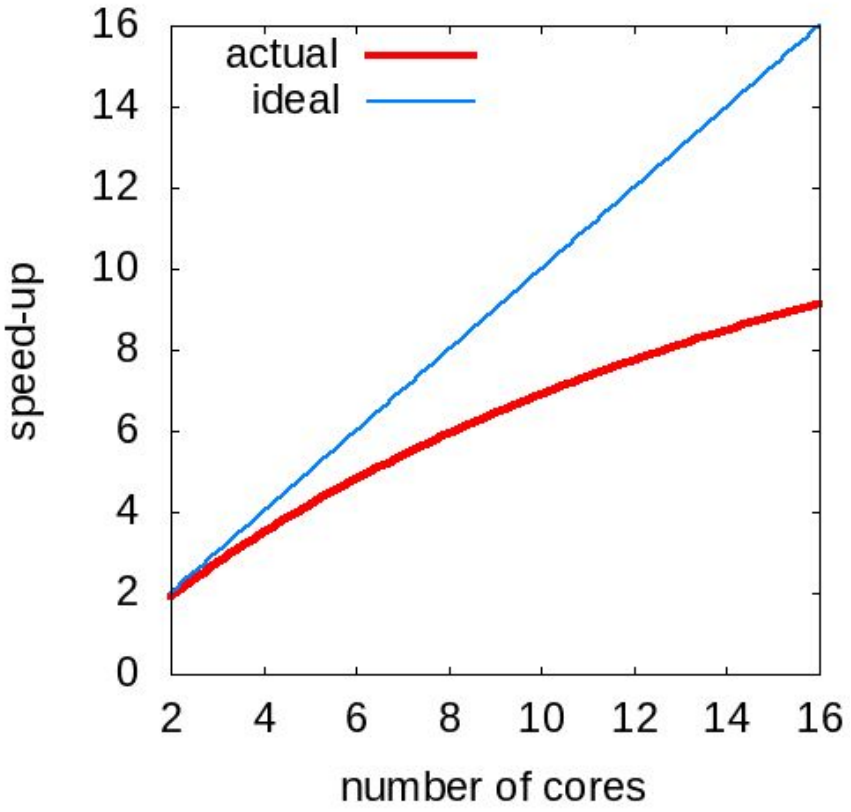
# THEORETICALS CONSIDERATIONS

# Parallelization is a brute force “optimization”

- Optimize first!
- Consider using faster algorithm.
  - N-body simulations: directs summation O(N^2) vs tree-code O(N * log(N))
- Use the fastest libraries.
  - matrix multiplication: MKL vs naive function
- Python: Code critical part in C/C++.
- Do profiling to find bottleneck.

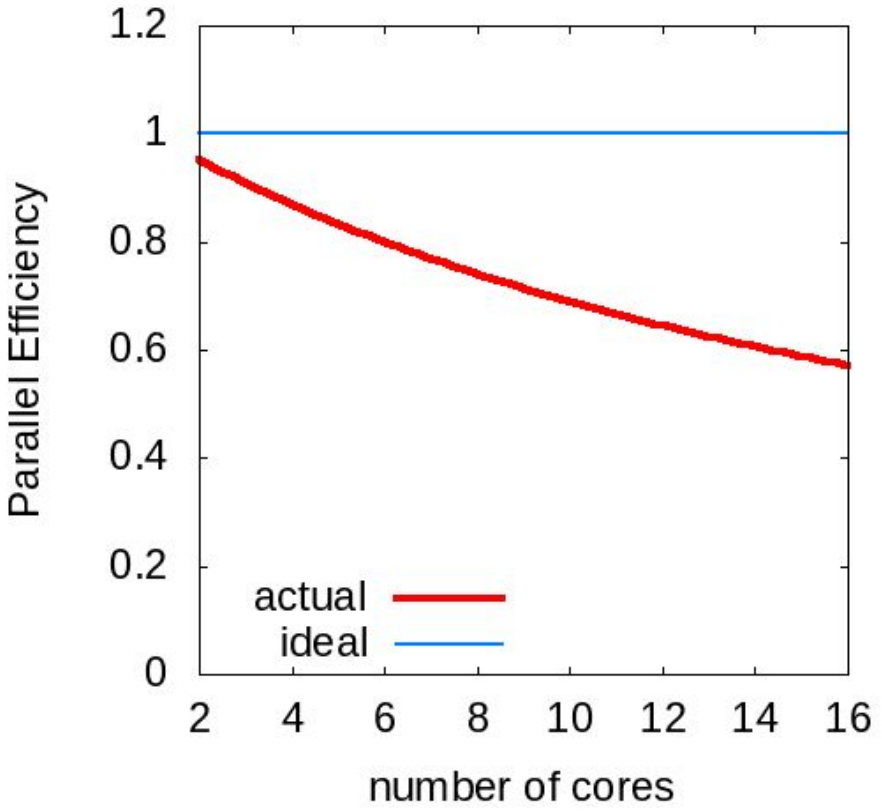
# Speed-up

T(1)/T(n) as function of n




# Efficiency

T(1)/T(n)/n as function of n




Always make speed-up analysis for your parallel program!

# Why real speedup is smaller than theoretical maximum?

1. **Parallel Overhead:** The amount of time required to coordinate parallel tasks, as opposed to doing useful work.
   - Synchronizations
   - Data communications
   - Software overhead imposed by parallel languages, libraries, operating system, etc.
2. **Serial components** of the program. Parts which have not been parallelized.

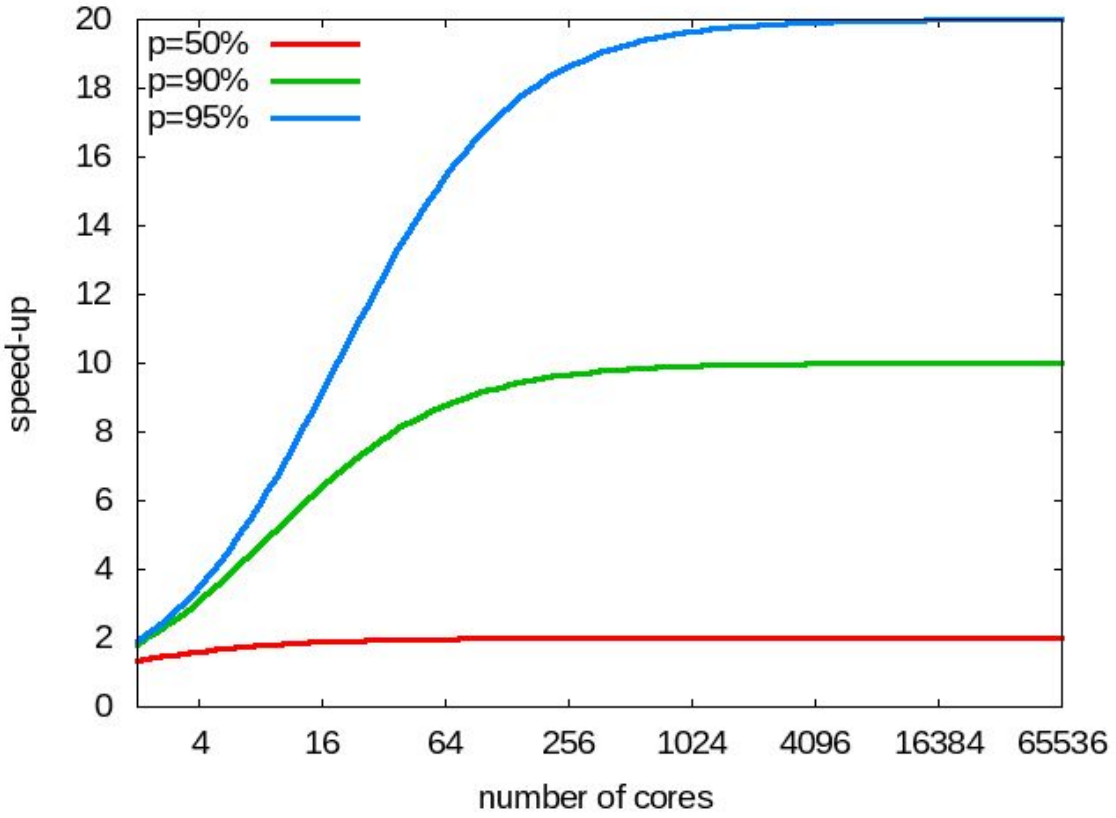
# Amdahl's law

p - fraction of execution time of part which we can parallelize

Part which we cannot parallelize


$$\frac{T(1)}{T(N)} = \frac{1}{1 - p + \frac{p}{n}}$$

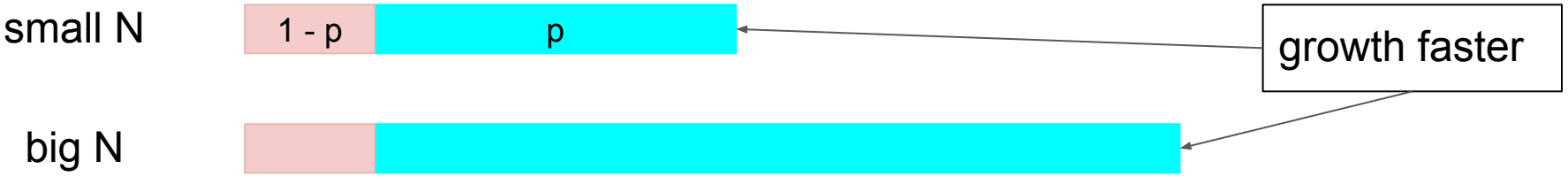
$$\frac{T(1)}{T(\infty)} = \frac{1}{1 - p}$$




1. A man can do no more than he can.
2. Are we doomed?

# Gustafson's law

Observation: when size of the problem growth “parallel part” usually growth faster than “sequential part”



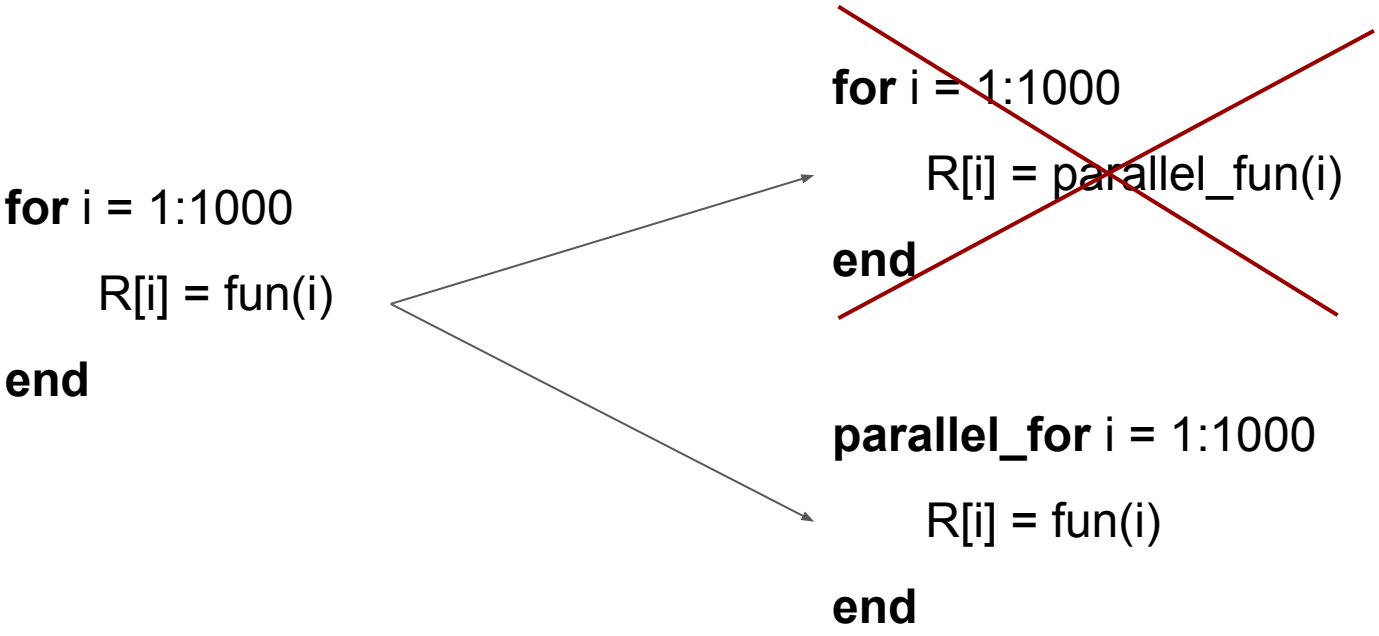

Speedup for fixed execution time: bigger number of cores --- bigger N

$$\frac{T_n(1)}{T_n(n)} = 1 - p + np \approx np$$

We **cannot** infinitely speedup a given program.

We **can** solve more and more complex problems in reasonable time.

# Make parallelization on as high level as possible.




Smaller number of bigger parallel blocks ==> smaller parallel overhead

Smaller sequential part ==> bigger speed-up

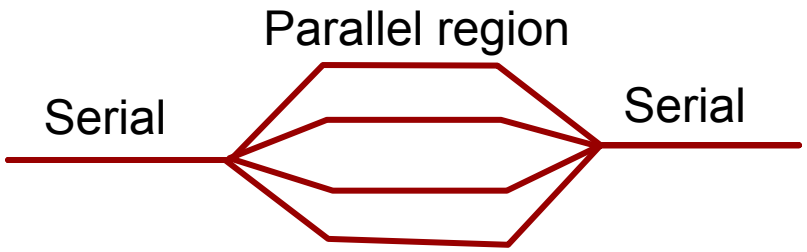
*But with smaller blocks might be easier achieve load-balancing*

# “Rules of Parallelization”

1. Do you need this program to work faster?
2. Do you need this particular program to work faster? Bottleneck of your project?
3. Optimize first. Use superiours algorithms and the fastest libraries
4. Make parallelization on as high level as possible.
5. You could violate rules 3 and 4 because of practical reasons.
6. Make speed-up plots.
7. Amdahl's law: speedup_max = 1/(1-p)
8. Gustafson's law: for bigger N speedup can be better

# Types of parallelization (from point of view of “processes”).

- Separate processes
  - without communication : gnu parallel, cluster batch
  - with communication : mpi




- Processes created with “fork” (expensive, but copy-on-write on linux)
  - python multiprocessing

- Threads (lightweight processes with shared memory)
  - We cannot use it in Python directly (GIL)
  - openmp for C++/C and FORTRAN

## PART III :

# PRACTICAL WORK

- Connect to lam's cluster
  - ssh -l **parwork cluster.lam.fr**
- Download exercises from gitlab (once connected to cluster)
  - cd WORKS
  - mkdir ***your_username***
  - cd ***your_username***
  - git clone https://gitlab.lam.fr/srodionov/parallel-computing-practice.git

  (There is a README.txt file with git command in WORKS directory)

# Cluster basic commands

## Query commands

- **showq** : display detailed jobs status (3 categories : running/waiting/blocked)
- **qstat** : display simplified jobs's status
- **qstat -u *jclamber*** : display ***jclamber***'s jobs status
- **qnodes** : display nodes's information
- **checkjob *jobid***: give ***jobid***'s info

## Action commands

- **qdel *jobid***: delete ***joib*** from the queue
- **qsub -q *queue mybatch.sh*** : submit ***mybatch.sh*** to the selected ***queue***
- **qsub -q *mpi mybatch.sh*** : submit ***mybatch.sh*** to ***mpi*** queue
- **qsub -q *batch mybatch.sh*** : submit ***mybatch.sh*** to ***batch*** queue
- **qsub -t *200* -q *batch mybatch.sh*** : submit ***200 mybatch.sh*** to ***batch*** queue

## Portable Batch System (PBS) Torque resource manager : **SCRIPT FILE**

Example : **myjob.sh**

```
#!/bin/sh
#
#PBS -N MYJOBname          # job's name seen from qstat command
#PBS -l nodes=1:ppn=1      # request 1 node and 1 core
#PBS -l walltime=12:00:00 # 12 hours wall clock time
#PBS -M jean-charles.lambert@lam.fr # mail send to this adress
#PBS -m abe  # sends a mail if job (a)aborted,(b)begins,(e)ends
#PBS -o "run-log.txt"    # standard output redirection
#PBS -e "run-err.txt"    # standard error redirection

# change to submission jobs directory
cd $PBS_O_WORKDIR
my_program  # start my_program
```

Portable Batch System (PBS) Torque resource manager : **ENV VARIABLE**

**Following variables are available within the submitted script**

- **PBS_O_WORKDIR** : specify directory from where script has been submitted
- **PBS_NUM_NODES** : #nodes requested
- **PBS_NUM_PPN** : #cores requested per node
- **PBS_JOBID** : job id number
- **PBS_NODEFILE** : name of the file containing list of HOSTS provided for the job
- **PBS_ARRAYID** : Array ID numbers for jobs submitted with the -t flag

# Gnu parallel

GNU **parallel** is a shell tool for executing jobs in parallel using one or more computers

- How to run
  ```
  cat list_of_jobs            | parallel
  cat list_of_parameters | parallel   program   {}
  ```

- “-j” Control number of workers
  ```
  seq 0 100 | parallel -j 10  "sleep 1;   echo {}"
  seq 0 100 | parallel -j 50% "sleep 1; echo {}"
  ```

- “-k” Keep sequence of output same as the order of input.
  ```
  seq 0 100 | parallel     "sleep 1; echo {}"
  seq 0 100 | parallel -k "sleep 1; echo {}"
  ```

# How to parallelize this code?

```
#matrix multiplication
def mult(A,B):
    n = len(A)
    C = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i,j] += A[i,k] * B[k,j]
    return C

N=5000
A = np.random.rand (N, N)
B = np.random.rand (N,N)
C = mult(A,B)
```

# Python multiprocessing

High level interface: pool of workers + map

map(f, [1,2,3]) ⇒ [f(1),f(2), f(3)] ]

```
from multiprocessing import Pool

def f(x):
    return x*x

p      = Pool()
rez  = p.map(f, [1,2,3])
```

# Python multiprocessing (low level)

Process(target=fun, args=args)  start new process and run fun(args) there.

Queue - create queue with put/get functions which are “process save”.

```
from multiprocessing import Process, Queue

def f(q):
    rez = 2*3*7
    q.put([rez, None, 'hello'])

q = Queue()
p = Process(target=f, args=(q,))
p.start()
print q.get()    # prints "[42, None, 'hello']"
p.join()
```