HPC in half a day

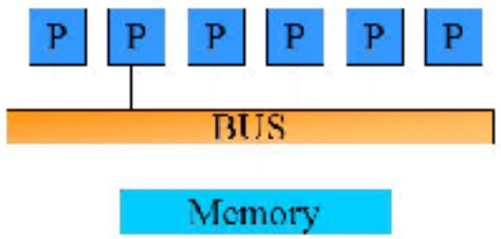
Andres Diaz-Gil

Software Carpentry Workshop IFT 2017

HPC systems

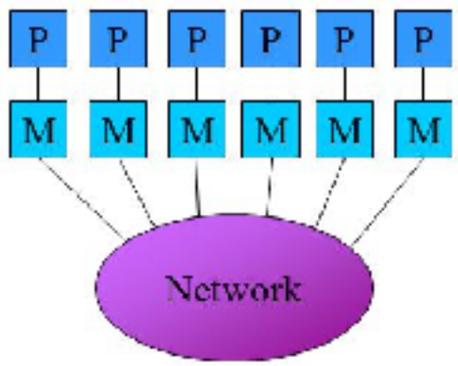
Mostly every HPC system falls in one of this two classifications:

Shared vs. Distributed Memory



Shared memory - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

Distributed memory - each processor has it's own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)



HPC clusters

HPC clusters

Distributed Memory Machines

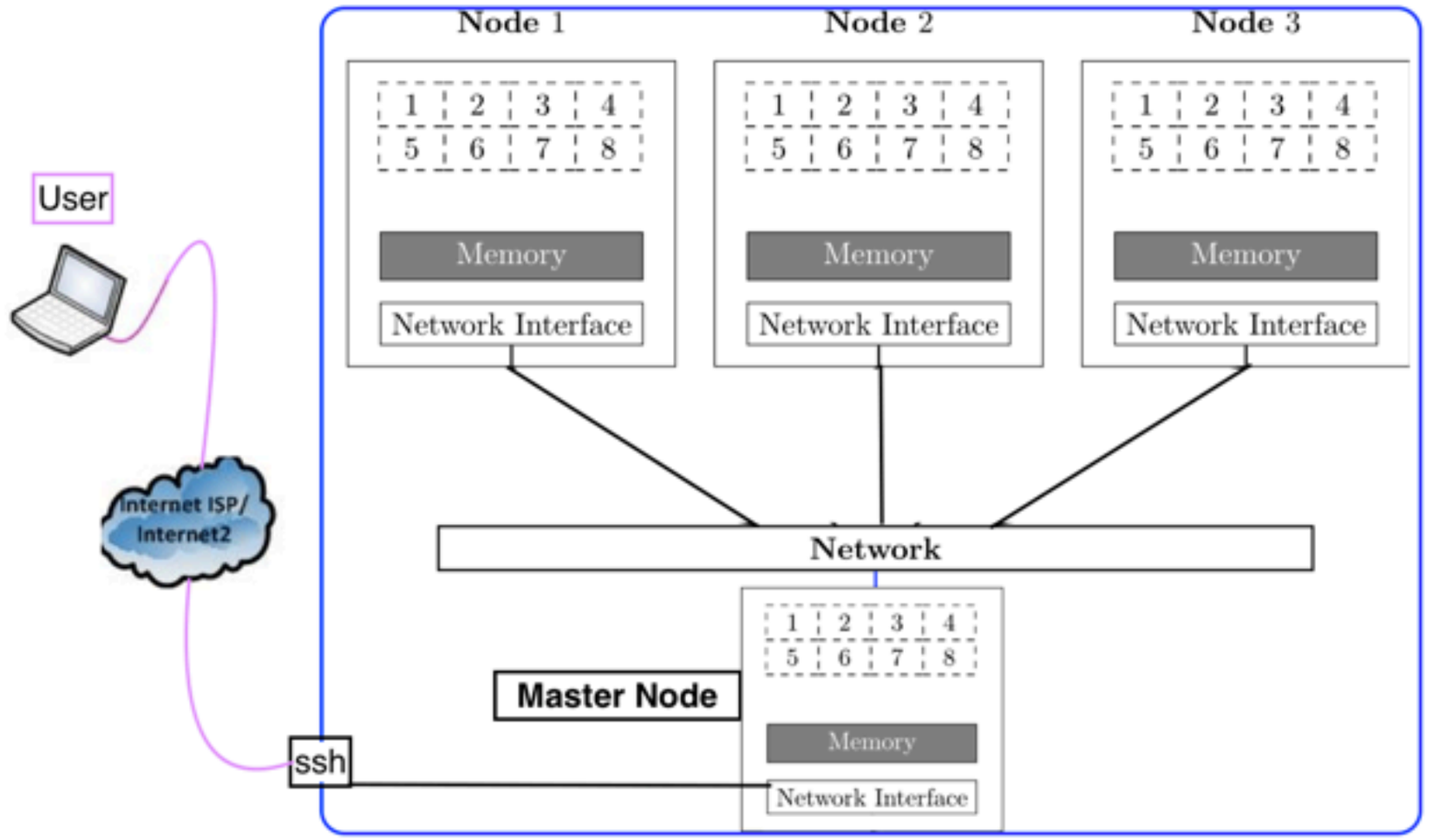
Several nodes & special network interconnection

Remote machines

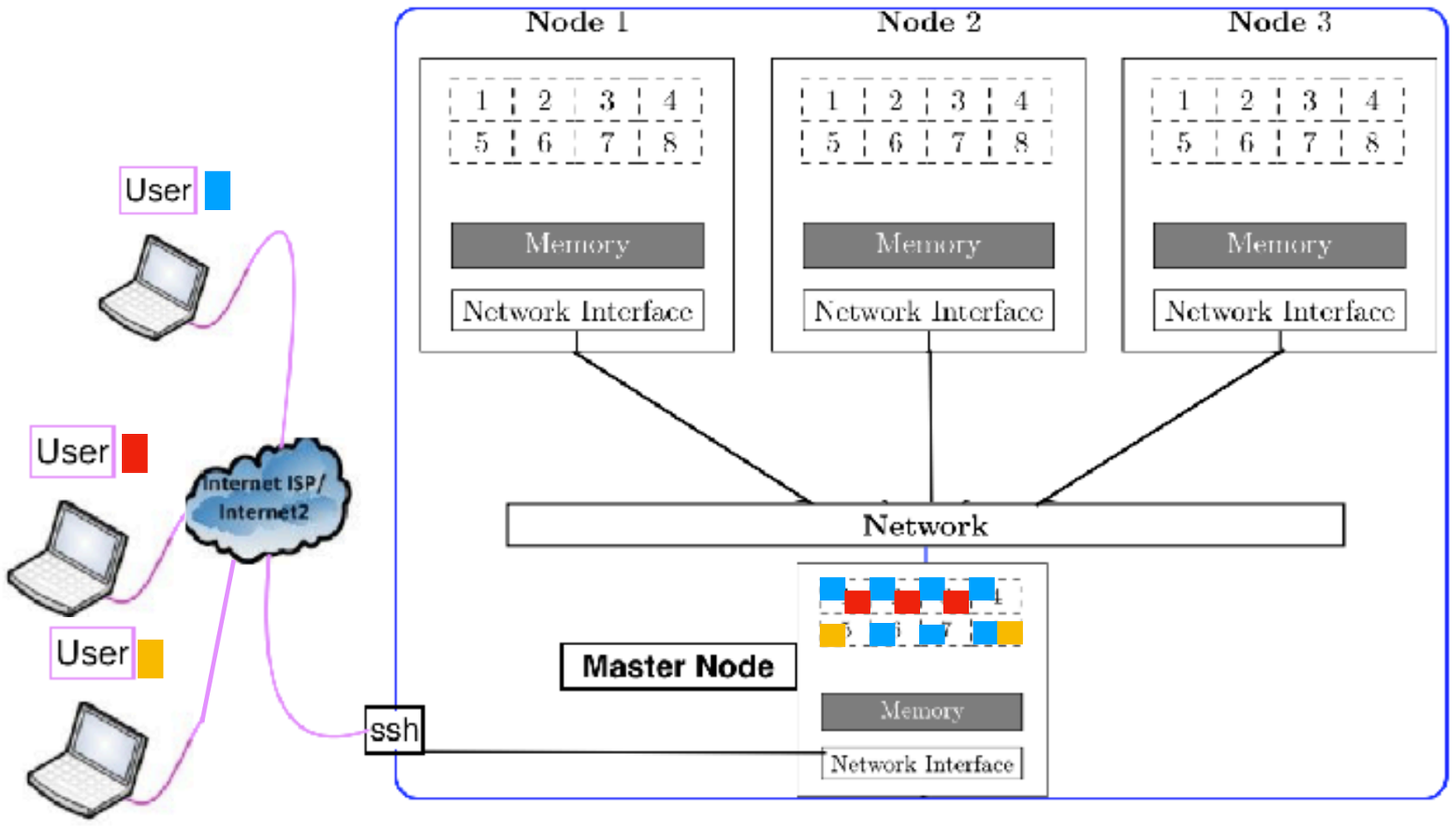
- **Access remote machines**
  - **ssh username@machinename**
- **Copy files in/out**
  - **scp**
    - **Single file**
    - **Multiple files/folder -r**
  - **rsync??**
- **Login out**
  - **logout**
  - **exit**

HPC Cluster (outside)

HPC Cluster (inside)



NO Schedulers

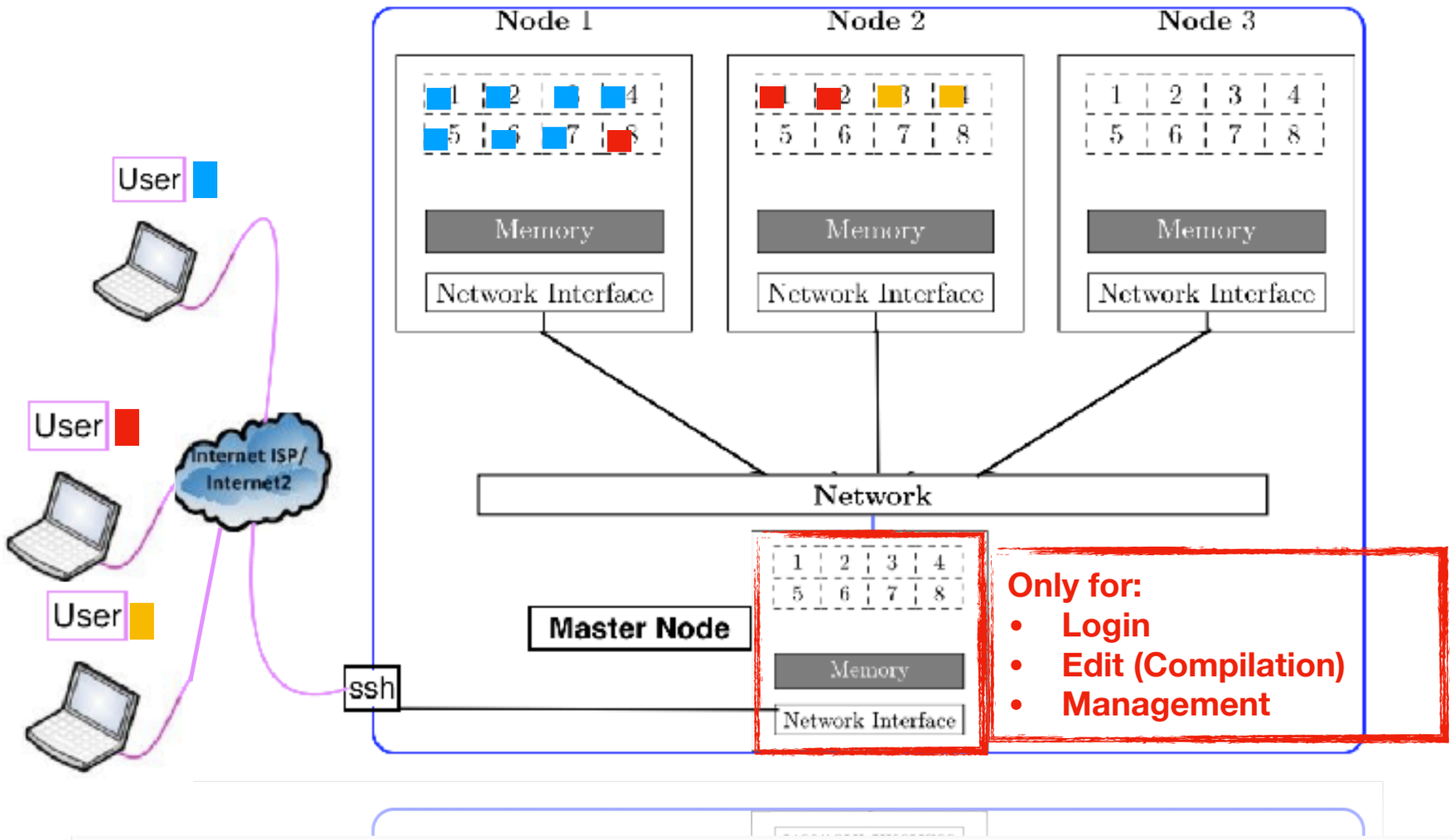


Interactive execution on master node:

- **./program.py**
- **python program.py**

Absolute disaster!!

Schedulers



Schedulers allocate resources for jobs in an ordered manner

- **queues (partitions)**

SLURM

Useful SLURM Commands:

- **squeue**
  - **squeue -p <queue>**
  - **squeue -u username**
  - **squeue -t RUNNING**
  - **squeue -t PENDING**

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

ST:

```
PD (pending)
R (running)
CD (completed)
CG (completing)
F (failed)
```

REASON:

```
JobHeldAdmin
PartitionNodeLimit
Priority
Resources
```

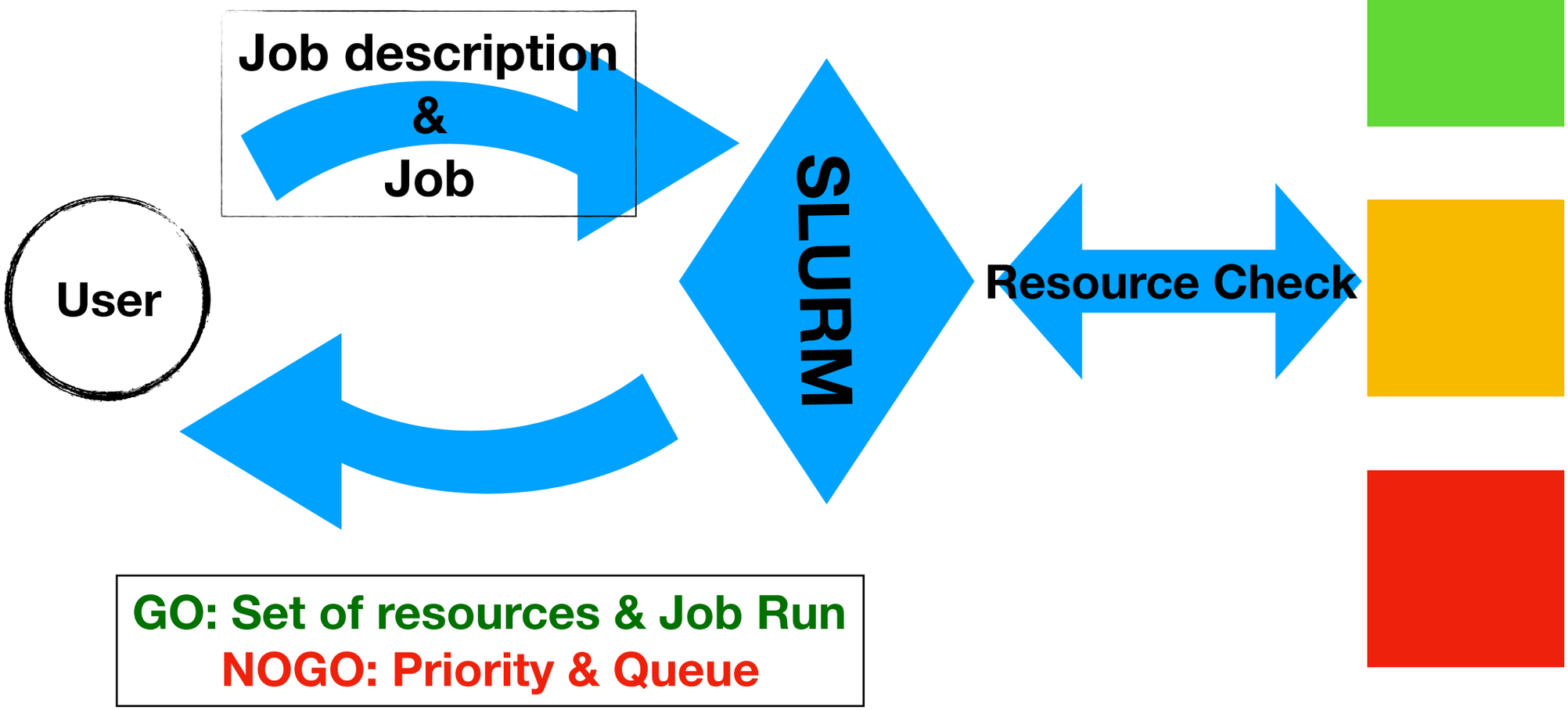
- **sinfo**
  - **sinfo -p <queue>**
  - **sinfo -Nl**

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
```

down
alloc
idle
mixed

Submit a Job

Universal procedure. No matter the nature of the job



GO: Set of resources & Job Run
NOGO: Priority & Queue

Submit a Job bis.

User —> describe job to the scheduler

Description usually includes:

- Number of tasks/cores needed
- Queue or partition required
- Walltime
- Executable/Instructions to run
- Input/enviroment needed

How this description is given depends on the running mode:

Interactive mode **Batch mode**

Interactive mode

Useful for debugging and testing

Pros:
It is like running in local.
You can see output or modify input in realtime
Cons:
The user must interact with the scheduler on every step to run the job

First example:
salloc
hostname
srun hostname
squeue -u <username>

```
[adgdt@master ~]$ salloc
salloc: Granted job allocation 2660
[adgdt@master ~]$ hostname
master.kerbero
[adgdt@master ~]$ srun hostname
nodo1.kerbero
[adgdt@master ~]$ exit
exit
salloc: Relinquishing job allocation 2660
```

Batch mode

Useful for production running

Pros:

Batch execution

Reusability

No action for the user is required afterwards

Cons:

You can not interact with job's IO

A batch script is needed

First example:
Open editor, create batch script
sbatch multiple_commands.sh
squeue -u <username>

```
[adgdt@master class]$ cat multiple_commands.sh
#!/bin/bash
hotname
date
echo –n "Number of Cores: “
cat /proc/cpuinfo | grep processor | wc –l
```

```
[adgdt@master class]$ sbatch multiple_commands.sh
Submitted batch job 2664
[adgdt@master class]$ squeue –u adgdt
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

Batch mode cont.

Exercise:
“sleep N”, is a linux command that basically waits N seconds.
Modify batch script to make it to take longer than 30s to complete.
What is now the output of squeue?

Controlling jobs

scancel & scontrol

To cancel one job: **scancel <jobid>**

To cancel all the jobs for a user: **scancel -u <username>**

To cancel all the pending jobs for a user: **scancel -t PENDING -u <username>**

To cancel one or more jobs by name: **scancel --name myJobName**

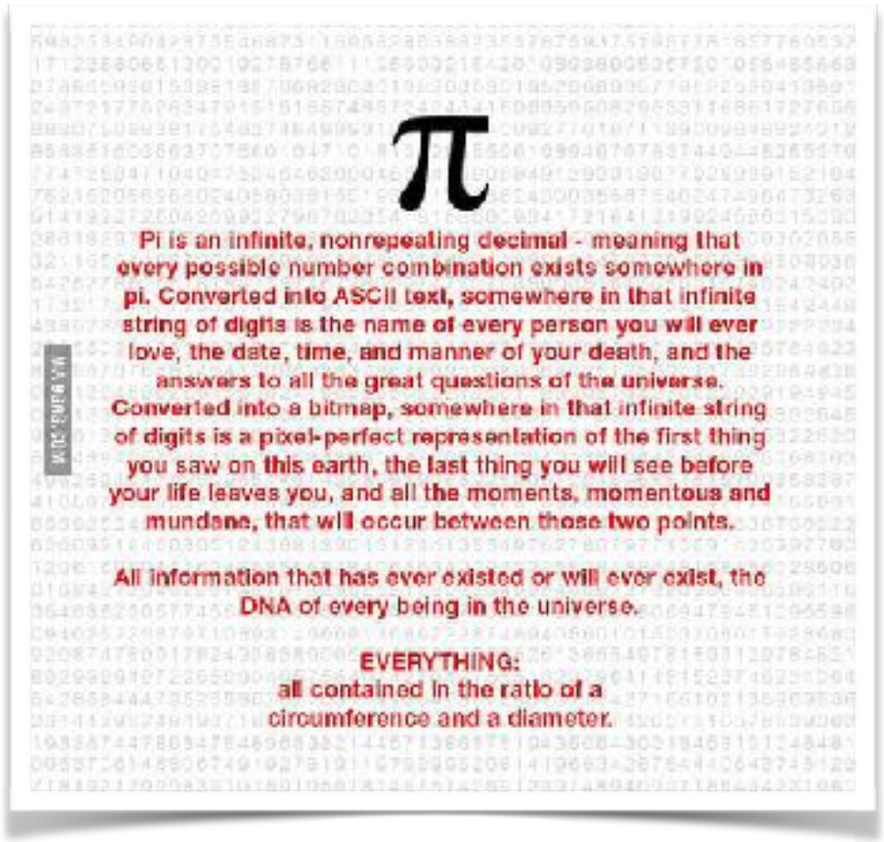
To pause a particular job: **scontrol hold <jobid>**

To resume a particular job: **scontrol resume <jobid>**

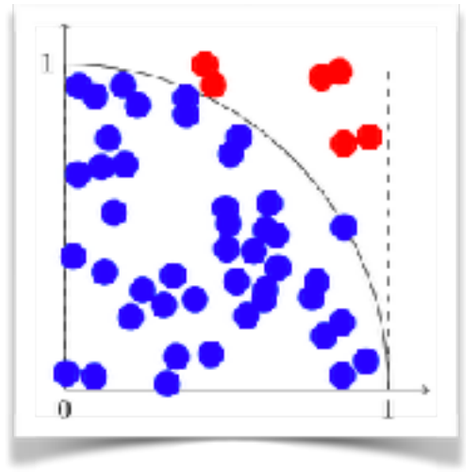
To requeue (cancel and rerun) a particular job: **scontrol requeue <jobid>**

To gather information about a job: **scontrol show job <jobid>**

The secret of Pi



Buffon's algorithm



n random points in the ([0,1),[0,1)) plane
m inside the circle of radius 1
Pi ~ 4m/n

Let's give it a try in Python…

Why Python in HPC?

- **Python is free**
- **Fast program development**
- **Simple syntax**
- **Easy to write well readable code**
- **Large standard library**
- **Lots of third party libraries**
  - **Numpy, Scipy**
  - **Mpi4py**
  - **…**

Python is not the most performant
But
Python can increase the performance of programmer drastically
And
Cython —> Source code translated to optimised C
Numba —> Compiler for Numpy arrays

“The slowest program is the one you never write”

Exercise

Open pi_serial.py with editor
Fill in the gaps to make Buffon's algorithm work
Run program in Master node:
$ python *pi_serial.py*

What is wrong??

```
Traceback (most recent call last):
  File "./pi_serial.py", line 8, in <module>
    import numpy as np
ImportError: No module named numpy
```

Modules

HPC clusters are complex systems
One user is doing python and the next is doing C or C++
One user uses GNU compiler, other Python 3, other Intel compiler etc…

All possibilities could not possible being loaded by default.
Conflicts and messed paths will arise….

Solution: Modules (lmod)
Each user prepares the system as she pleases, by loading/unloading preconfigured pieces of software
lmod takes care of the dependencies and avoids conflicts

lmod modules act pretty like import in Python, but at OS level

Working with modules

$ module list : shows currently loaded modules
$ module avail : shows wide system available modules
$ module spider/module spider *module_name* **:**
shows all possible modules and details

$ module load *module_name*
$ module unload *module_name*

module load *numpy*

```
module load numpy
Lmod has detected the following error: Cannot load module
numpy/1.10.2". At least one of these module(s) must be loaded:
   openblas


While processing the following module(s):

Module fullname  Module Filename
---------------  ---------------
numpy/1.10.2     /opt/ohpc/pub/moduledeps/gnu/numpy/1.10.2
```

module load *openblas*

Exercise

Run the program using SLURM batch mode

Remember!: To run any job in batch mode you must create a batch script!!

Name the script submit.sh

What is wrong this time???

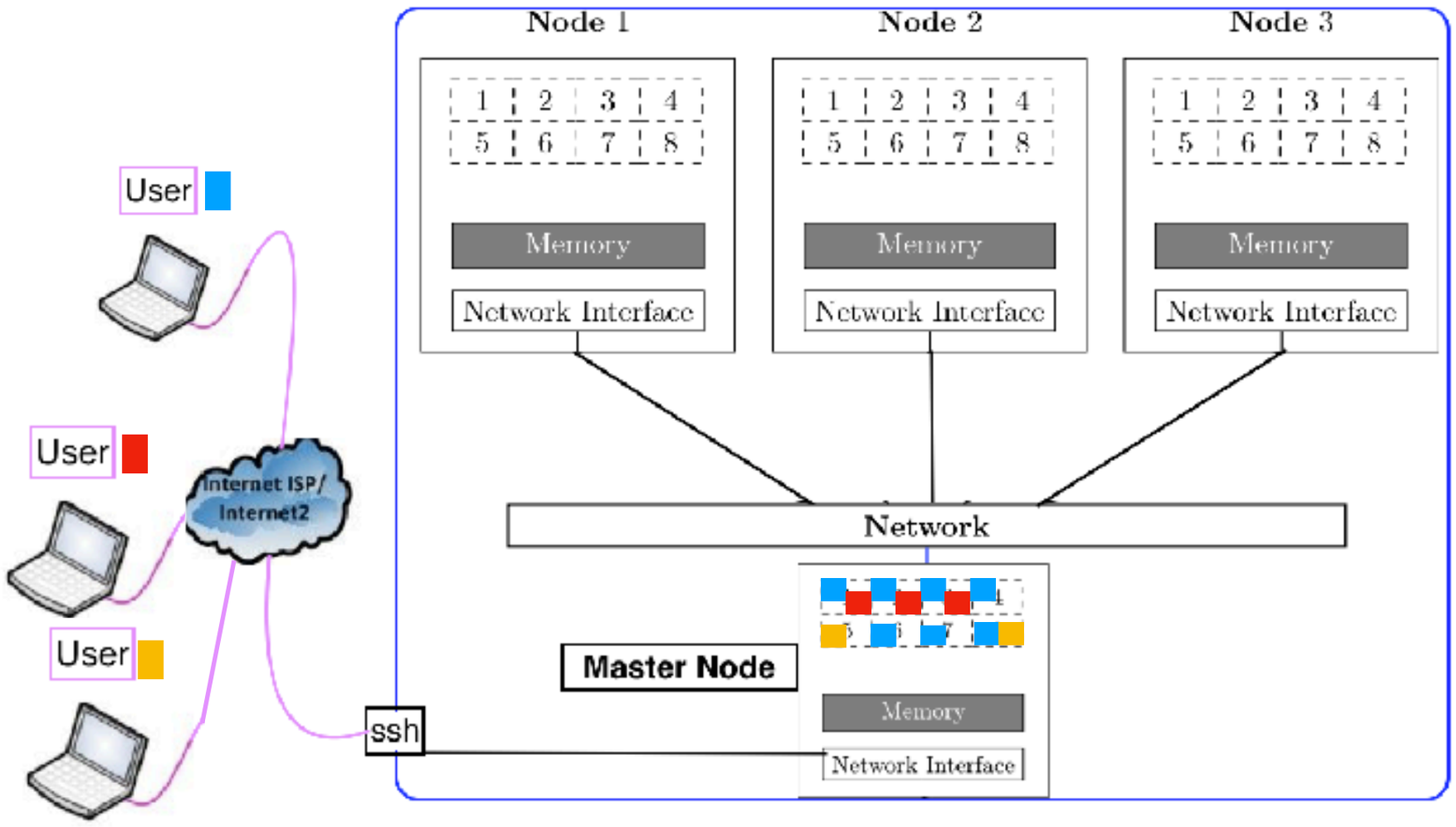
```
more slurm-2668.out
Traceback (most recent call last):
  File "pi_serial.py", line 8, in <module>
    import numpy as np
ImportError: No module named numpy
```

Try in the master node:

$ python *pi_serial.py*

```
[serial version] required memory 0.000 MB
[serial version] pi is 3.150000 from 10000 samples
```

NO Schedulers

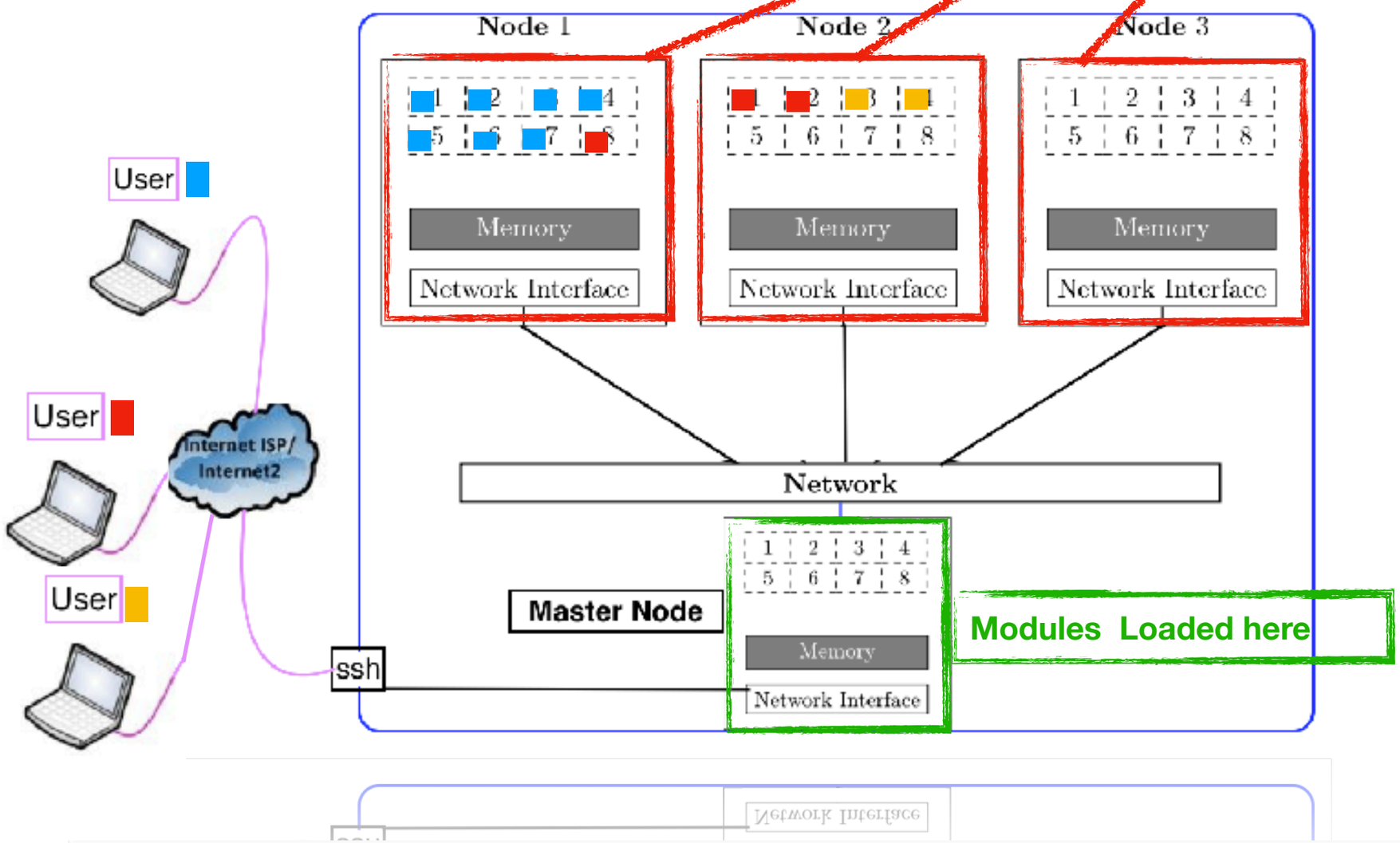


Interactive execution on master node:

- **./program.py**
- **python program.py**

```
[serial version] required memory 0.000 MB
[serial version] pi is 3.150000 from 10000 samples
```

Schedulers



Schedulers allocate resources for jobs in an ordered manner

- **sbatch** *script.sh*

```
more slurm-2668.out
Traceback (most recent call last):
  File "pi_serial.py", line 8, in <module>
    import numpy as np
ImportError: No module named numpy
```

Exercise

BATCH scripts are only BASH (or shell) scripts.
They contain special expressions to give specific orders to SLURM

SLURM stuff goes first:

- Starts under #!/bin/bash
- Every line begins with #SBATCH

BASH stuff goes second:

- Starts when all PBS finishes
- Made of regular BASH commands

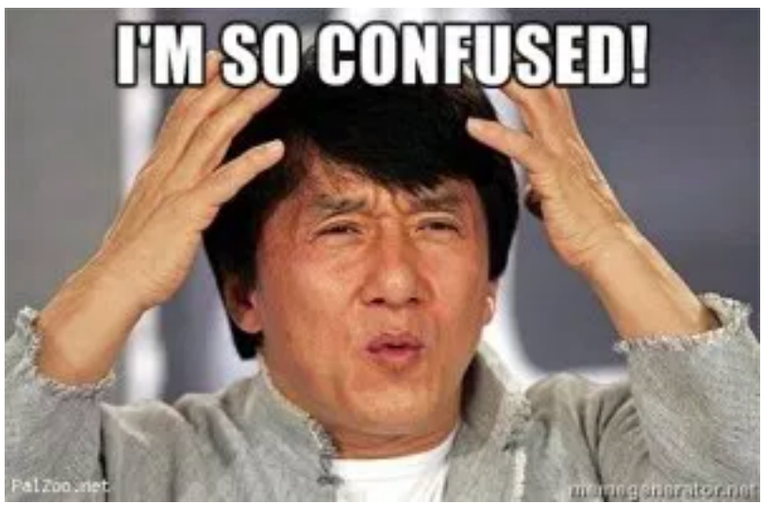
Refine our batch script to make it:

1) Load mandatory modules in the nodes

2) Change the name of the output file:

Use #SBATCH -o *filename.out*

Review



Too many scripts...

pi_serial.py : python script, aka python program.
All python stuff is in it.
Must be submitted to nodes
Otherwise overwhelms Master

BASH script: A script in shell language, with shell stuff. Calls to commands, loops, pipes and so on...
Seen long time ago... (Inigo's lesson first day)

submit.sh : a BATCH script
BATCH scripts contains both SLURM and BASH expressions:
SLURM instructions give descriptions to SLURM
BASH instructions run commands

What about #SBATCH, sbatch ????

- **sbatch : command to submit a BATCH script**
- **#SBATCH : keyword to start a special line in a a BATCH script, intended to describe something to SLURM**

A matter of time....

HPC is all about time (also about memory) but mostly about time

How long does it take to run the program?

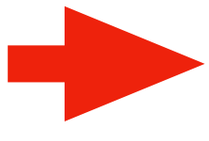
Let's modify submit.sh to see how long it takes
Lets run 100.000.000 samples

```
more pi_serial_10to8_2692.out
[serial version] required memory 1144.000 MB
[serial version] pi is 3.141606 from 100000000 samples

real 0m8.218s
user 0m6.388s
sys  0m1.160s
```

Understanding time output

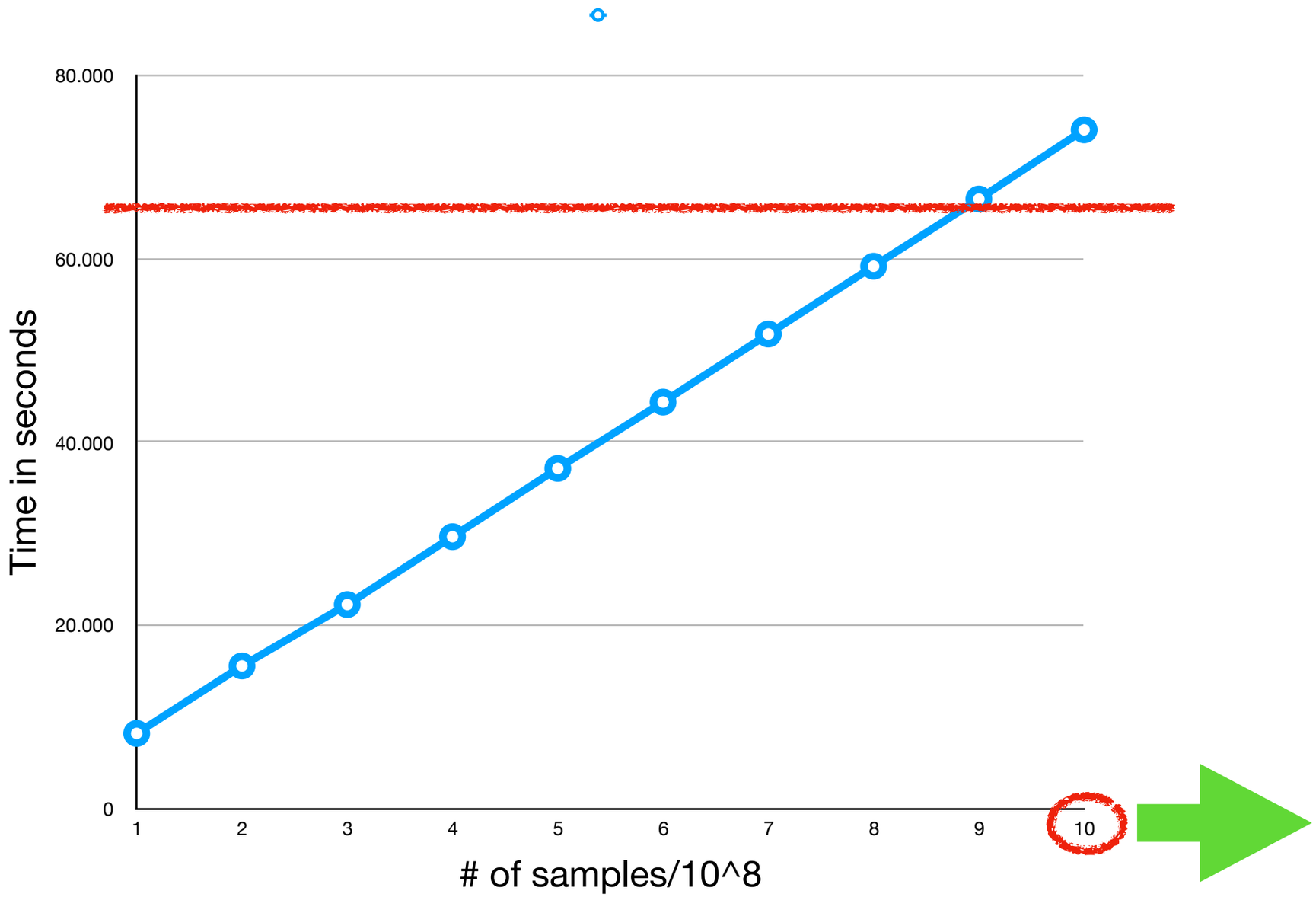
- `real` that denotes the time that has passed during our program as if you would have used a stop watch
- `user` this is accumulated amount of CPU seconds (so seconds that the CPU was active) spent in code by the user (you)
- `sys` this is accumulated amount spent while executing system code that was necessary to run your program (memory management, display drivers if needed, interactions with the disk, etc.)

```
more pi_serial_10to8_2692.out
[serial version] required memory 1144.000 MB
[serial version] pi is 3.141606 from 100000000 samples

real 0m8.218s
user 0m6.388s
sys  0m1.160s
```



user + sys < real **Why???**

Time Scaling



What is taking time?

Programs do a lot of stuff inside
Which one is taking the most?
Parts taking more time than others are called “HOT SPOTS”

Profiler: line_profiler/kernprof

Add @profile just before definition of any function you want to profile

Make it last 2~3 secs
pi_serial.py must be executable

kernprof -l ./pi_serial.py 50000000

python -m line_profiler pi_serial_profiled.py.lprof

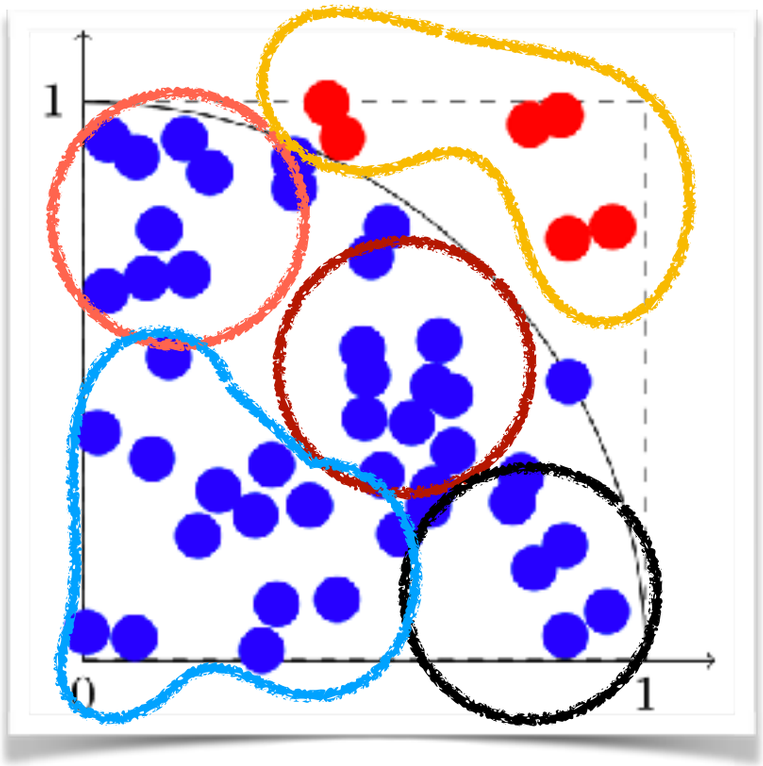
Profiler

```
python -m line_profiler pi_serial.py.lprof
Timer unit: 1e-06 s

Total time: 3.69153 s
File: ./pi_serial.py
Function: inside_circle at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           @profile
    13                                           def inside_circle(total_count):
    14
    15         1      1376097 1376097.0     37.3      x = np.float32(np.random.uniform(size=total_count))
    16         1      1376745 1376745.0     37.3      y = np.float32(np.random.uniform(size=total_count))
    17
    18         1       429997 429997.0      11.6      radii = np.sqrt(x*x + y*y)
    19
    20         1       508689 508689.0      13.8      count = len(radii[np.where(radii<=1.0)])
    21
    22         1            4      4.0       0.0      return count
```

75% of the time is taking by just 2 lines!!
That is a HUGE Hot spot

Parallelising

- **Divide programs in independent parts that can be made at the same time by different agents.**
- **Is very often a good way to save time, but can we?**



Parallelisation V1

Shared Memory, in-node or MP parallelisation

Level: Easy

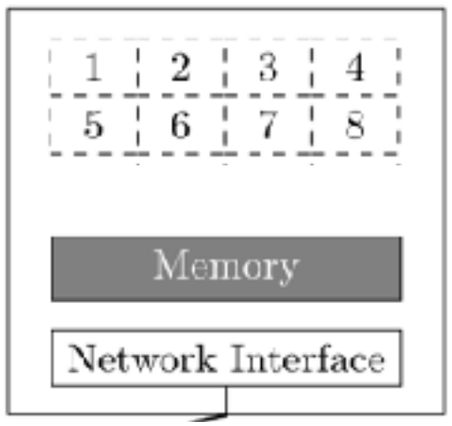
Module Multiprocessing: Process-based “threading” interface

The most basic example of data parallelism is Pool and its method pool.map

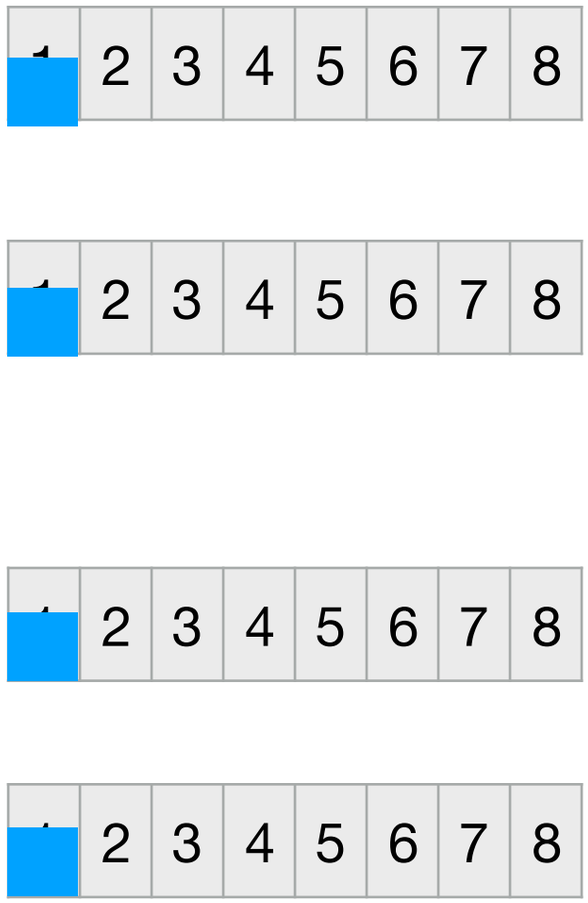
```
from multiprocessing import Pool, cpu_count
```

Serial vs Multiprocess

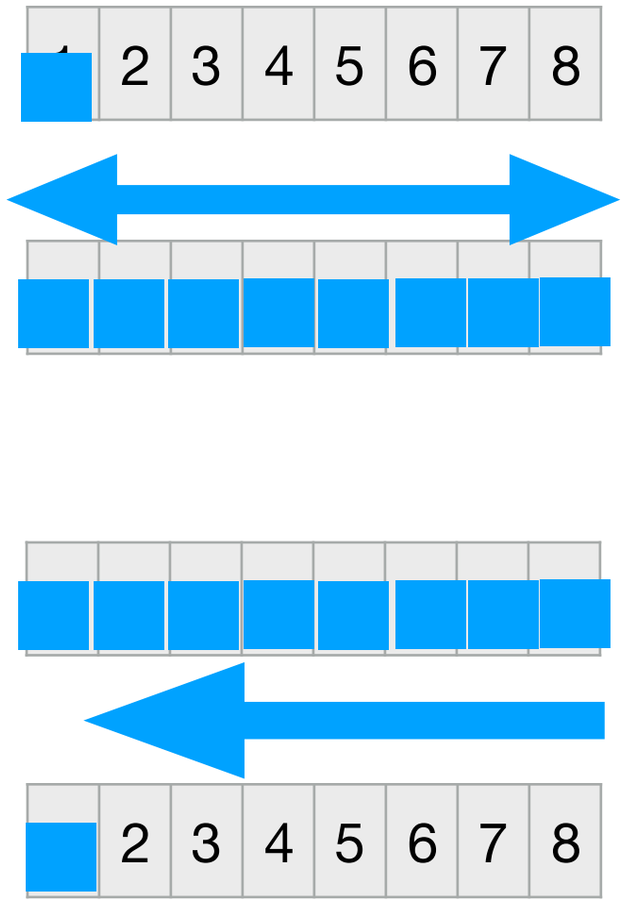
Node X



./pi_serial.py



Multiprocess pi



Time

Output

Output

Pool.map()

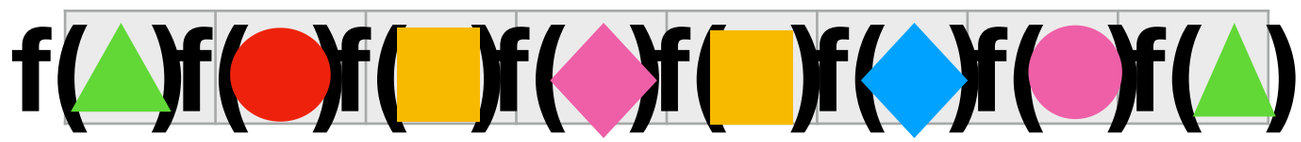
Structured data (a python list)

f(x)

```
def f(x):
  ....
```

1	2	3	4	5	6	7	8

Some cores



Returns a list with the return of the function on every data element

Exercise

Thinking parallel

Open pi_serial.py
Which function is suitable for pool.map() method?
Which is the list of data to pass to it?
What would be the returned data list?

Pool.map() applied to Pi

[n1pairs, n2pairs, n3pairs ....,n8pairs]

Structured data (a python list)

inside_circle(x)

```
def inside_circle(x):
  ....
```

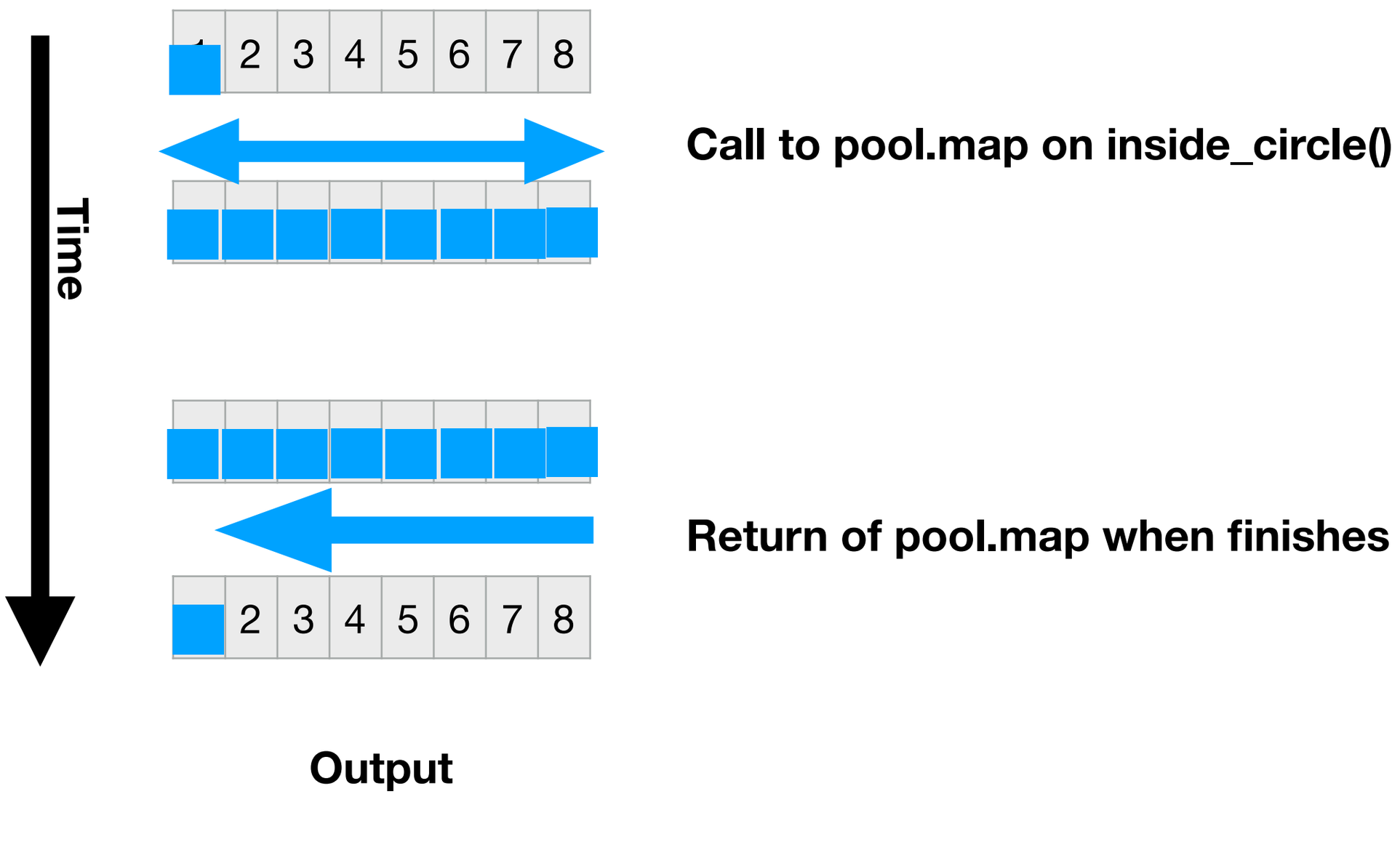
1	2	3	4	5	6	7	8

Some cores

[count1, count2, count3 ....,count8]

Returns a list with the return of the function on every data element

Multiprocess Pi



Exercise

Open pi_parallel.py
Fill in the gaps to make it multiprocessing
Modify BATCH script to submit it with 10^8 samples
And collect the time
Launch it to the queue but with this option:
sbatch --exclusive submit.sh

```
more pi_parallel_v1_2880.out
[parallel version] required memory 1144.000 MB
[using   8 cores ] pi is 3.141757 from 100000000 samples

real  0m2.535s
user  0m9.043s
sys0m2.665s
```

Time comparison

```
more pi_parallel_v1_2880.out
[parallel version] required memory 1144.000 MB
[using   8 cores ] pi is 3.141757 from 100000000 samples

real   0m2.535s
user   0m9.043s
sys    0m2.665s
```

Parallel

3,2x speedup

```
more pi_serial_10to8_2692.out
[serial version] required memory 1144.000 MB
[serial version] pi is 3.141606 from 100000000 samples

real  0m8.218s
user  0m6.388s
sys   0m1.160s
```

Serial

A longer run

```
cat pi_parallel_v1_2.10to9_2896.out
[parallel version] required memory 22888.000 MB
[using   8 cores ] pi is 3.141661 from 2000000000 samples

real    0m30.151s
user    3m1.794s
sys     0m51.711s
```

Parallel

5x speedup!

Remarkable things:

1) **user & sys time are longer in parallel**
2) **Memory size is always the same**
3) **Why not 8x???**

```
more pi_serial_2.10to9_2897.out
[serial version] required memory 22888.000 MB
[serial version] pi is 3.141555 from 2000000000 samples

real  2m30.141s
user  2m6.979s
sys   0m22.447s
```

Serial

Exercise: Something is terrible wrong in the program

Could you tell me what??

Going beyond the node

What if??

a) Time bounded problem:
I need twice the samples in about the same time?
b) Memory bounded problem:
Memory required is greater than available

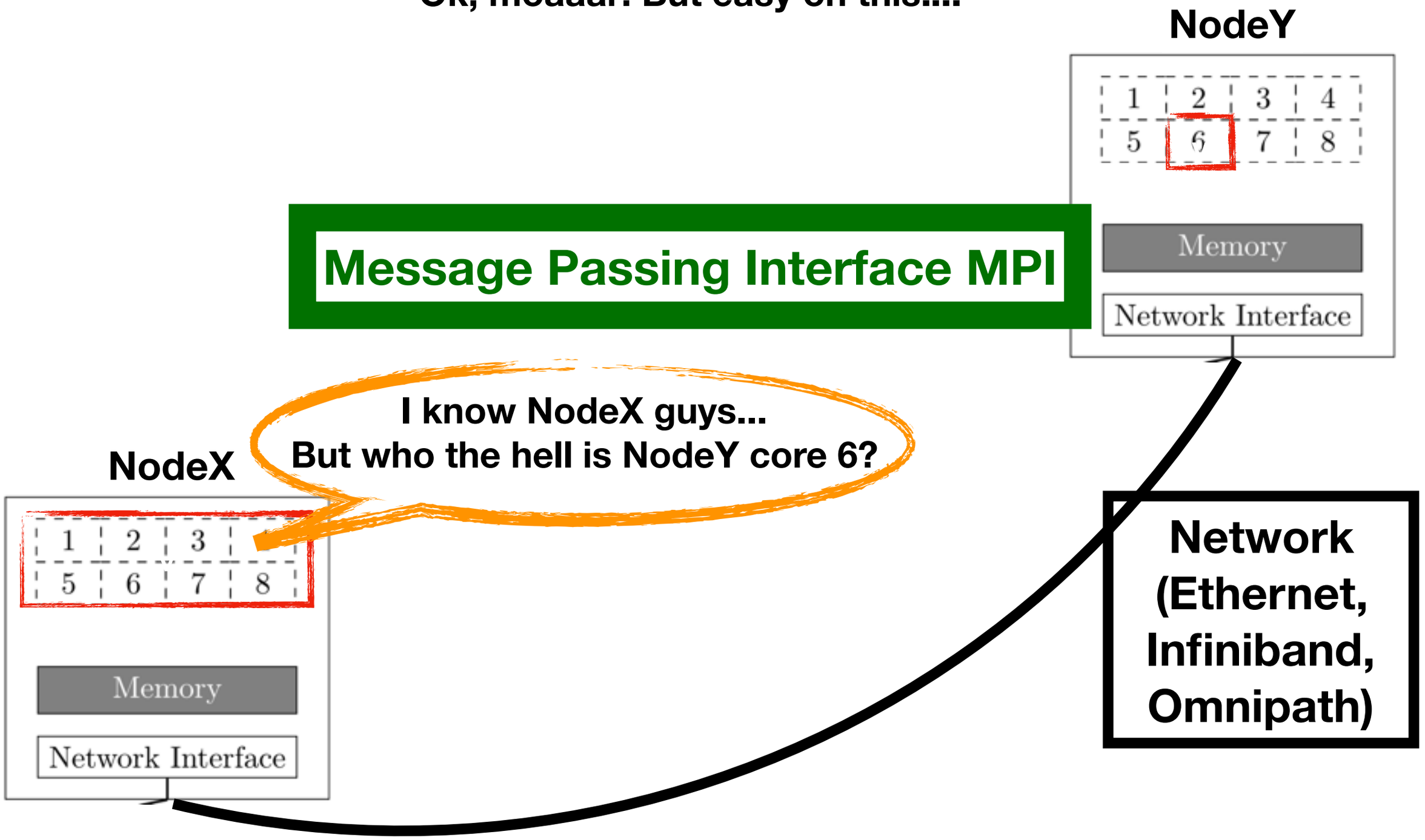
One node is limited to a certain number of cores and a certain memory

Solution:
Use more nodes!!!

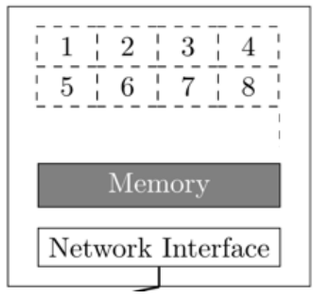
Distributed computing

Ok, moaaar! But easy on this....

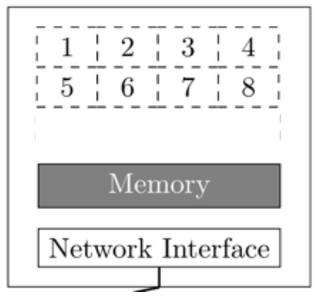
Message Passing Interface MPI



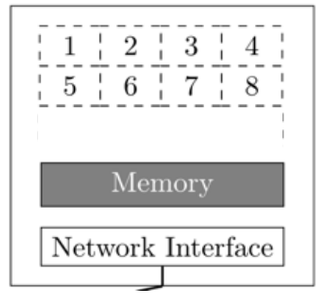
Serial vs MP vs MPI

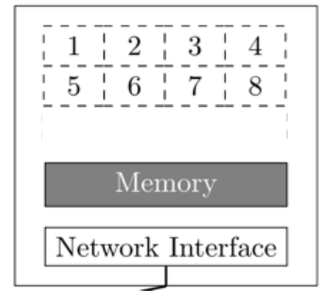


Serial

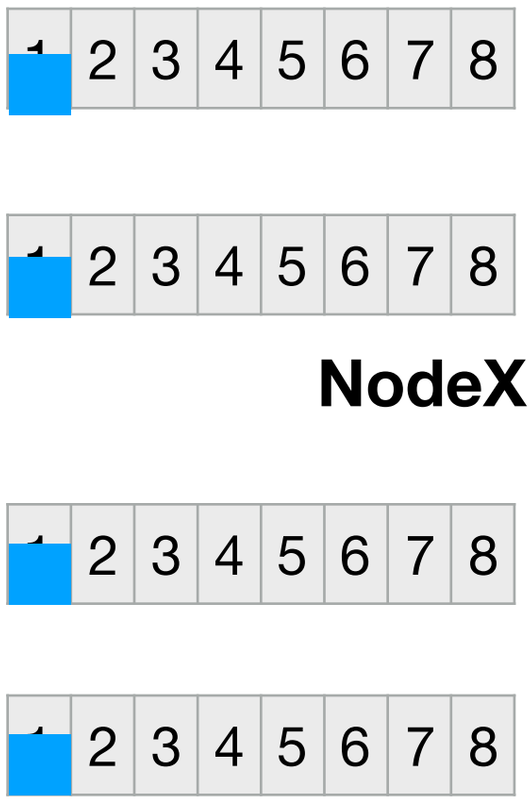


Multiprocess

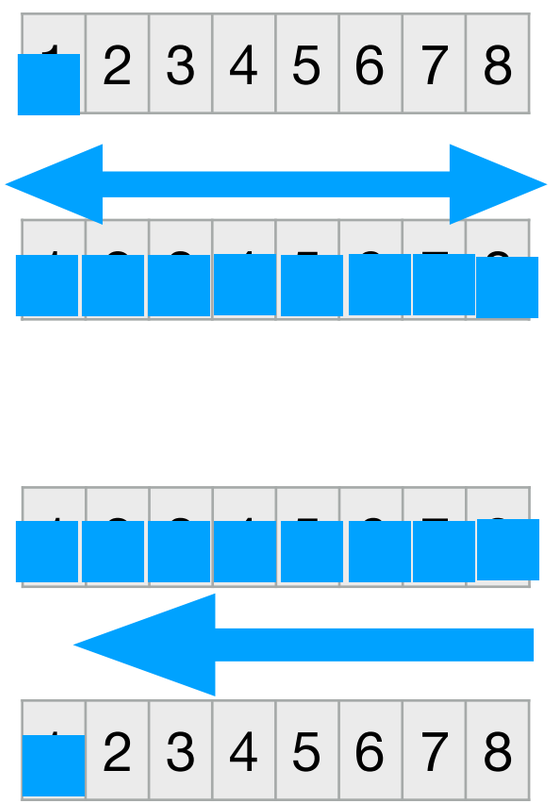




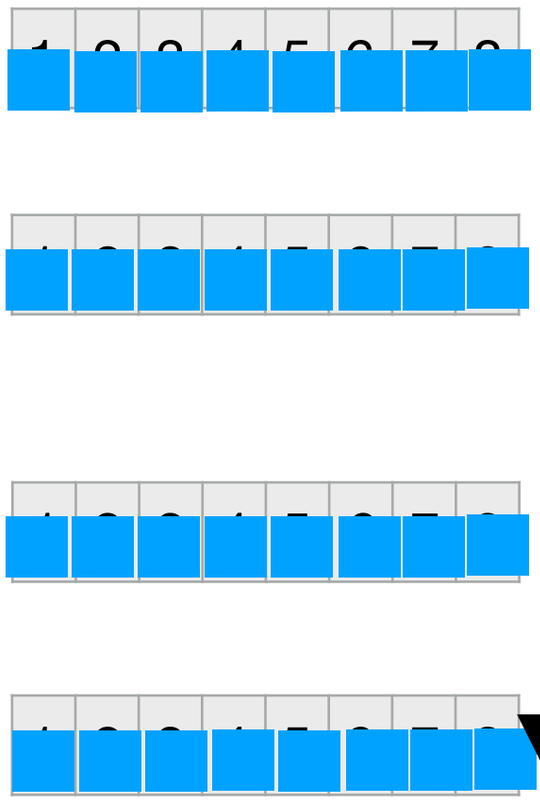
MPI

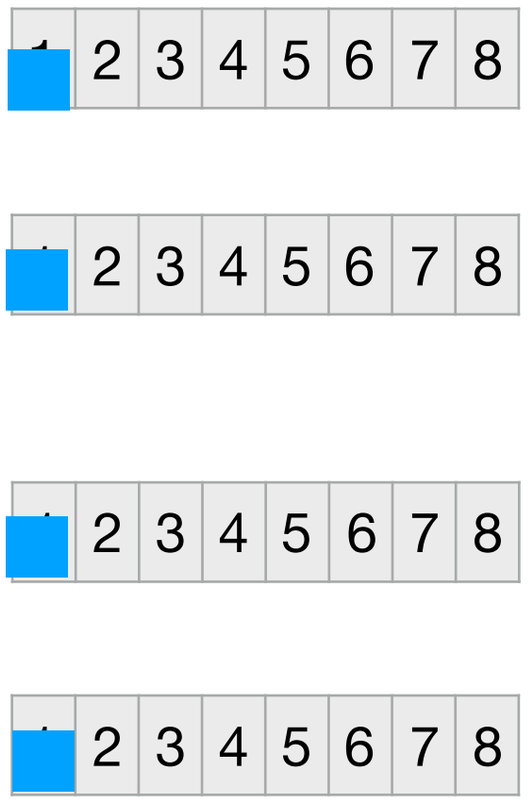


Output



Output



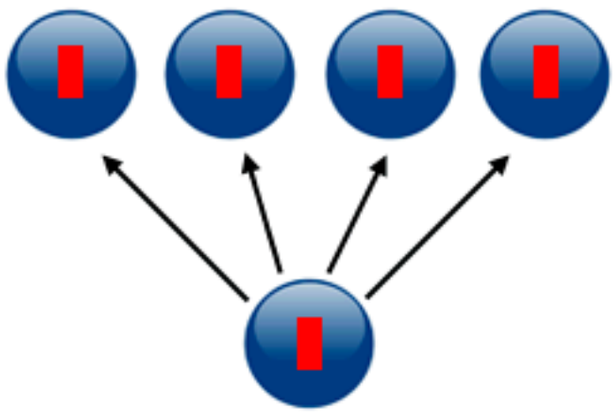


Output

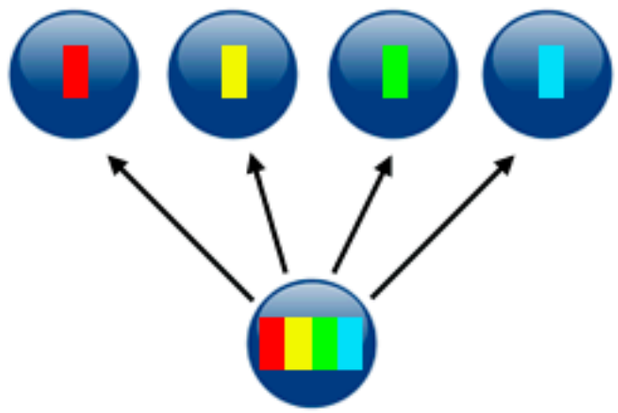
MPI in a nutshell

- When being said SLURM prepares nodes for you
- Each core has a copy of the program
- Each core has a rank number that identifies it
- First assigned is rank = 0
- Each core executes the program from beginning to end
- Do operations corresponding to its rank
- Communications are made through special functions

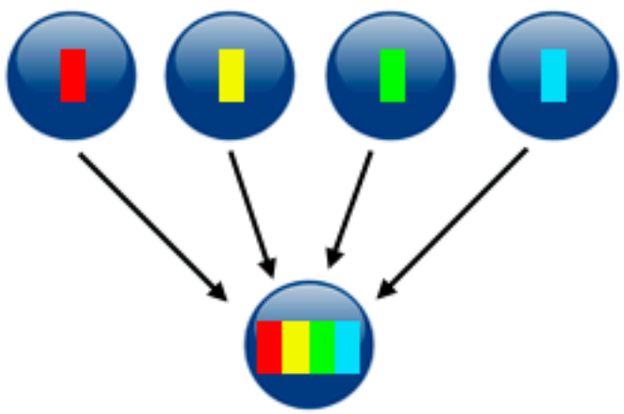
MPI collective communication functions



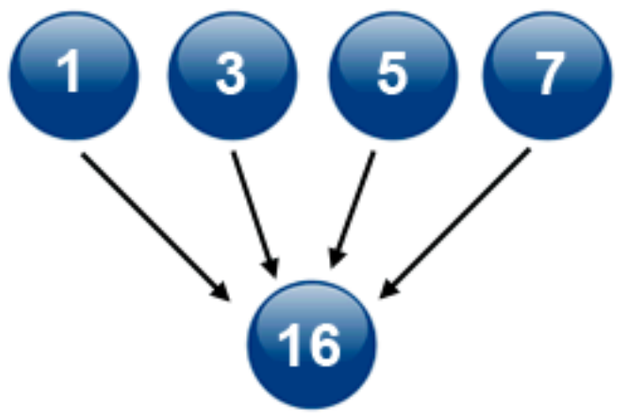
broadcast



scatter



gather



reduction

Distributed Python

mpi4py

mpi4py provides Python interface to MPI:

- **Object-oriented interface similar to standard C++**
- **Communication of arbitrary (serializable) Python objects**
- **Communication of contiguous NumPy arrays at nearly C-speed**

mpi4py Recipe

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
```

Write regular Python code with actions depending on the rank

Use communication functions if needed

Simulated cluster

Rank

Output

0

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   print (“I am leading rank”)
else:
   print (“I will wait for rank 0 to send me something”)
```

I am leading rank

1

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   print (“I am leading rank”)
else:
   print (“I can wait for rank 0 to send me something”)
```

I can wait for rank 0 to send me something

Requesting resources

To go beyond the node we have to ask SLURM for resources. This imply to modify BATCH script slightly:

cat submit_mpi.sh

```
#!/bin/bash
#SBATCH -o mpi_hostame_%j.out
#SBATCH -n 12
prun hostname
```

```
more mpi_hostname_2930.out
[prun] Master compute host = nodo1
[prun] Launch cmd = mpirun hostname
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo1.kerbero
nodo2.kerbero
nodo2.kerbero
nodo2.kerbero
nodo2.kerbero
```

MPI PI

Exercise

a) **Open file pi_mpi.py and try to fill it as much as you can! This is slightly more difficult one....**

b) **Write a BATCH script to submit it, based in your submit.sh:**

- **Ask SLURM for 16 parallel tasks**
- **Ask for 10^8 samples**
- **Hint: maybe you need to load something more....**

MPI speedup!

```
more pi_mpi_2.10to9_2926.out
[prun] Master compute host = nodo1
[prun] Launch cmd = mpirun python pi_mpi.py 2000000000
[MPI version] required memory 22888.000 MB
[MPI version. Ranks= 16 ] pi is 3.141636 from
2000000000 samples

real  0m16.680s
user  1m31.234s
sys   0m26.861s
```

MPI Parallel

9x speedup!
Compared with serial

```
cat pi_parallel_v1_2.10to9_2896.out
[parallel version] required memory 22888.000 MB
[using   8 cores ] pi is 3.141661 from 2000000000 samples

real    0m30.151s
user    3m1.794s
sys     0m51.711s
```

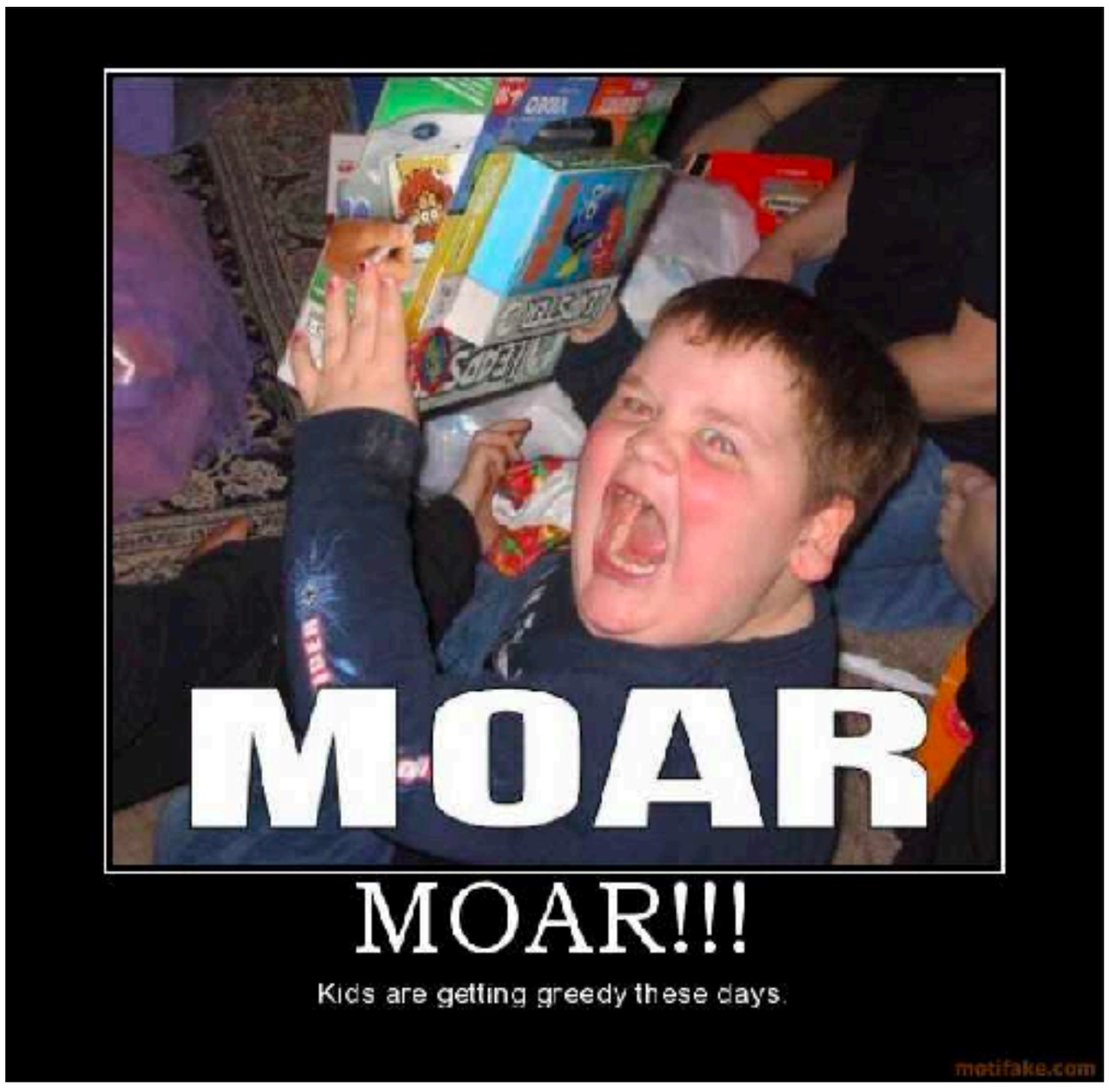
Parallel

5x speedup!
Compared with serial

```
more pi_serial_2.10to9_2897.out
[serial version] required memory 22888.000 MB
[serial version] pi is 3.141555 from 2000000000 samples

real  2m30.141s
user  2m6.979s
sys   0m22.447s
```

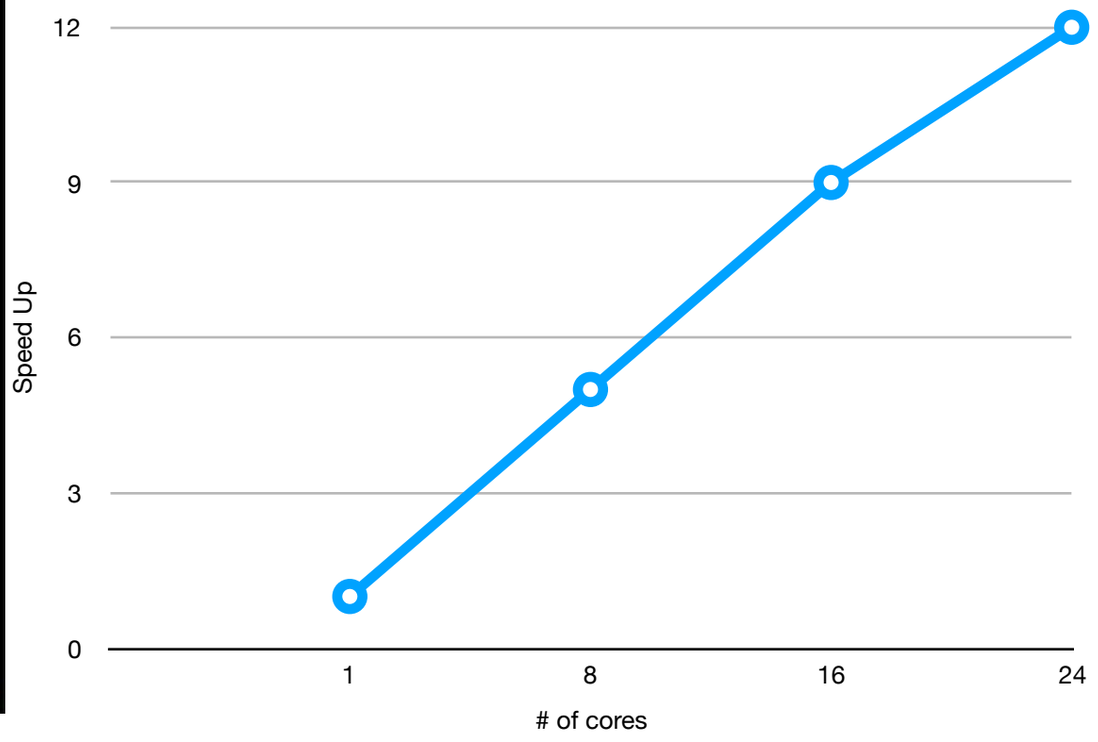
Serial

Mooooooaaaaaar!




```
more pi_mpi_all_2.10to9_2931.out
[prun] Master compute host = nodo1
[prun] Launch cmd = mpirun python pi_mpi.py
2000000000
[MPI version] required memory 22888.000 MB
[MPI version. Ranks= 24 ] pi is 3.141643 from
2000000000
 samples

real    0m12.513s
user    1m0.779s
sys     0m18.396s
```

12x speedup



Don’t be greedy!