Fundamentals of parallelization

Anthony Scemama

23-24/11/2021

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse (France)

Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizoon 2020 research and innovation programme under Grant Agreement **No. 952165.**

Outline

1. Supercomputers
2. Fundamentals of parallelization
3. Embarrassingly parallel computations
4. Inter-process communication
5. Message Passing Interface (MPI)
6. Multi-threading
7. OpenMP
8. Summary
9. Exercises

CC BY NC SA

Supercomputers

cc BY NC SA

Today (order of magnitude)

- 1 socket (x86 CPU @ 2.2-3.3 GHz, 4 cores, hyperthreading)
- ~ 4-16 GB RAM
- ~ 500GB SSD
- Graphics card : ATI Radeon, Nvidia GeForce
- Gigabit ethernet
- USB, Webcam, Sound card, etc
- ~ 500 euros

cc BY NC SA

Today (order of magnitude)

- 2 sockets (x86 CPU @ 2.2 GHz, 32 cores/socket, hyperthreading)
- 64-128 GB RAM
- Multiple SSD HDDs (RAID0)
- Gigabit ethernet
- Possibly an Accelerator (Nvidia Volta/Ampere)
- $\sim$ 5k euros

CC BY NC SA

- Many computers designed for computation
- Compact (1-2U in rack) / machine
- Network switches
- Login server
- Batch queuing system (SLURM / PBS / SGE / LFS)
- Cheap Cooling system
- Requires a lot of electrical power (~10kW/rack)
- Possibly a Low-latency / High-bandwidth network (Infiniband or 10Gb ethernet)
- >50k euros

Supercomputer

- Many computers designed for computation
- Very compact (<1U in rack) / machine
- Low-latency / High-bandwidth network (Infiniband or 10Gb ethernet)
- Network switches
- Parallel filesystem for scratch space (Lustre / BeeGFS / GPFS)
- Multiple login servers
- Batch queuing system (SLURM / PBS / SGE / LFS)
- Highly efficient cooling system (water)
- Requires a lot of electrical power (>100kW)

cc BY NC SA

Definitions

Top500	Rank of the 500 fastest supercomputers
Flop	Floating point operation
Flops	Flop/s, Number of Floating point operations per second
RPeak	Peak performance, max possible number of Flops
RMax	Real performance on the Linpack benchmark (dense eigenproblem)
SP	Single precision (32-bit floats)
DP	Double precision (64-bit floats)
FPU	Floating Point Unit
FMA	Fused multiply-add ($a \times x + b$ in 1 instruction)

CC BY NC SA

Example

RPeak of Intel Xeon Gold 6140 Processor :

- 18 cores
- 2.3 GHz
- 2 FPUs
- 8 FMA (DP)/FPU/cycle

$18 \times 2.3\,10^9 \times 2 \times 8 \times 2 = 1.3$ TFlops (DP)

Number of hours 730/month, 8760/year

Units Kilo (K), Mega (M), Giga (G), Tera (T), Peta (P), Exa (E), ...

cc BY NC SA

Top500 (1996)

Rank	System	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)	Power (kW)
1	CP-PACS/2048, Hitachi/Tsukuba Center for Computational Sciences, University of Tsukuba Japan	2,048	368.2	614.4	
2	Numerical Wind Tunnel, Fujitsu National Aerospace Laboratory of Japan Japan	167	229.0	281.3	498
3	SR2201/1024, Hitachi University of Tokyo Japan	1,024	220.4	307.2	
4	XP/S140, Intel Sandia National Laboratories United States	3,680	143.4	184.0	
5	XP/S-MP 150, Intel DOE/SC/Oak Ridge National Laboratory United States	3,072	127.1	154.0	

Top500 (2021)

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	7,630,848	442,010.0	537,212.0	29,899
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

https://www.top500.org/lists/top500/2021/11/

cc BY NC SA

Ranked 9th in 2012, 77 184 cores, 1.7 PFlops, 2.1 MW

Mare Nostrum (BSC, Spain)

Ranked 13th in 2017, 153 216 cores, 6.5 PFlops, 1.6 MW

cc BY NC SA

Supercomputer

Islands

Nodes

Sockets

Cores

Chassis (Front)

bullx blade chassis (B500 series)

LCD unit

Fans

9 blades = 18 nodes

7U chassis

CC BY NC SA

Chassis (Back)

bullx blade chassis (B500 series)

Chassis Management Board

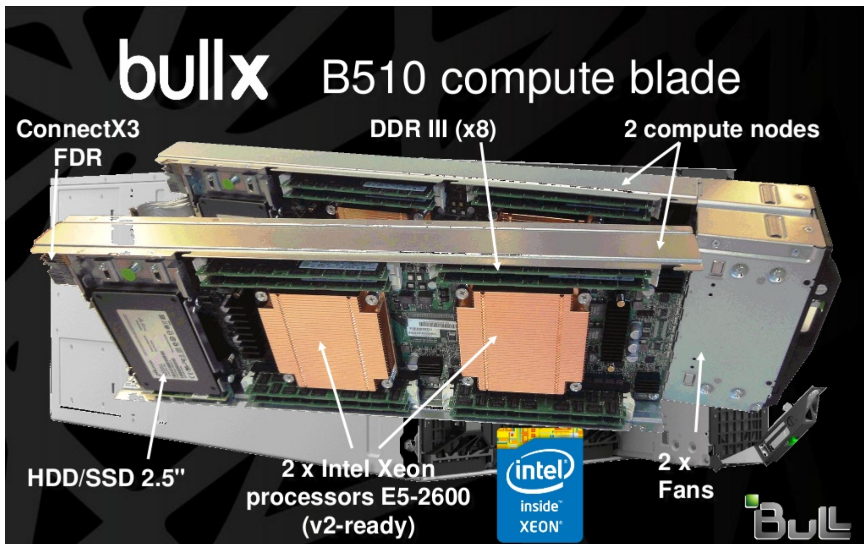
InfiniBand Switch board

Ultracapacitor

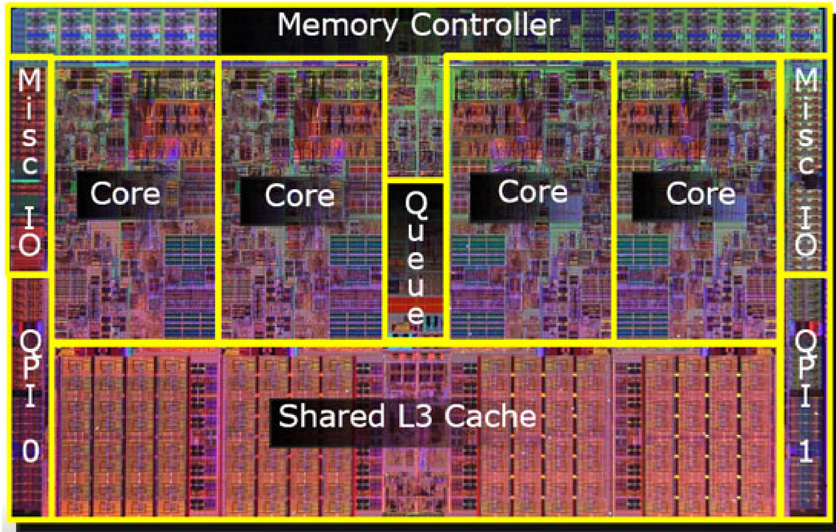
Hot- swappable PSU x 4

Ethernet Switch Board

CC BY NC SA


bullx B510 compute blade
ConnectX3 FDR
DDR III (x8)
2 compute nodes
HDD/SSD 2.5"
2 x Intel Xeon processors E5-2600 (v2-ready)
intel inside XEON
2 x Fans
BULL

cc BY NC SA


Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1

Moore's "Law"

- *The number of transistors doubles every two years.*
- Often interpreted as *the computational power doubles every two years.*
- Exponential law $\Longrightarrow$ will fail.
- 7~nm semiconductors in 2021: the atomic limit is approaching.

Main problem: energy

Dissipation : $D = F \times V^2 \times C$

- F : Frequency ($\sim$ 3 GHz)
- V : tension
- C : constant related to the size of semiconductors (nm)

For the processor to be stable, the tension needs to be linear with the frequency: so $D = \mathcal{O}(F^3) \implies$ The frequency has to be kept around 3 GHz.

1. Double the number of Flops = double the number of processors.
2. Use accelerators (GPUs, FPGA, TPUs, ...)

CC BY NC SA

Main problem: energy

Dissipation : $D = F \times V^2 \times C$

- F : Frequency (~ 3 GHz)
- V : tension
- C : constant related to the size of semiconductors (nm)

For the processor to be stable, the tension needs to be linear with the frequency: so $D = \mathcal{O}(F^3) \implies$ The frequency has to be kept around 3 GHz.

1. Double the number of Flops = double the number of processors.
2. Use accelerators (GPUs, FPGA, TPUs, ...)

Consequence

This requires to re-think programming $\implies$ Parallel programming is unavoidable.

CC BY NC SA

1. Energy, physical limits
2. Interconnect technology
3. Increase of computational power is faster than memory capacity :

reduction of memory per CPU core

1. Latencies can't be reduced much more : moving data becomes very expensive
2. File systems are extremely slow
3. Supercomputers are well tuned for benchmarks (dense linear algebra), but not that much for general scientific applications
4. Fault-tolerance

CC BY NC SA

GPUs

Use less intelligent CPUs, but in a much larger number $\Longrightarrow$ Better flops/watt rate

But...

- Transfer from main memory to GPU as expensive as a network communication
- More computational power and less RAM than a CPU node
- Not adapted to all algorithms
- Requires re-thinking and rewriting of codes: loss of years of debugging
- Software stack is not as mature as the CPU stack $\Longrightarrow$ needs hacking to get performance

cc BY NC SA

Fundamentals of parallelization

Definitions

Concurrency Running multiple computations at the same time.

Parallelism Running multiple computations on different execution units.

Multiple levels

Distributed	Multiple machines
Shared memory	Single machine, multiple CPU cores
Hybrid	With accelerators (GPUs, . . .)
Instruction-level	Superscalar processors
Bit-level	Vectorization

All levels of parallelism can be exploited in the same code, but every problem is not parallelizable at all levels.

cc BY NC SA

Definition

If P is the proportion of a program that can be made parallel, the maximum speedup that can be achieved is:

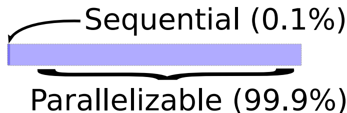
$$S_{\max} = \frac{1}{(1-P) + P/n}$$

S : Speedup

P : Proportion of a program that can be made parallel

n : Number of cores

Example



- Max speedup : $100\times$
- Perfect scaling needs all the program to be parallelized (difficult)

1. Human parallel machine
2. Compute a PES with GNU parallel (Bash)
3. Compute π with a Monte Carlo algorithm (Python, sockets)
4. Compute π with a deterministic algorithm (Python, MPI)
5. Matrix multiplication (Fortran, OpenMP)

cc BY NC SA

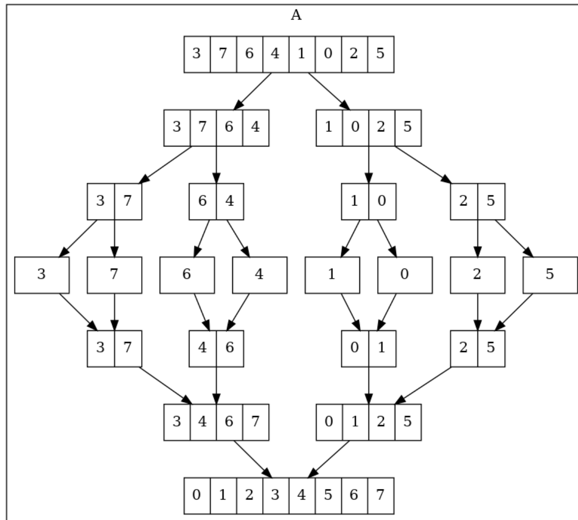
The Human Parallel Machine

- Sort a deck of cards
- Do it as fast as you can
- I did it alone in 5'25'' (325 seconds)
- Each one of you is a *human compute node*
- How fast can all of you sort the same deck?

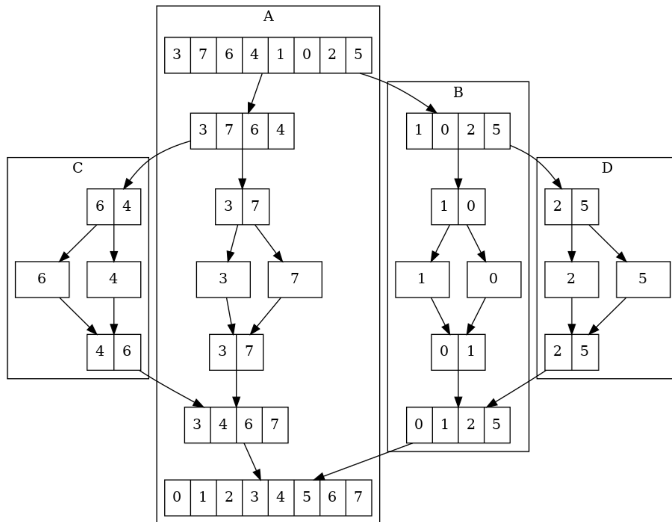
CC BY NC SA

You were many more people than me, but you didn't go many times faster !

$\Longrightarrow$ Try the Merge sort



cc BY NC SA


A
3 7 6 4 1 0 2 5
3 7 6 4
3 7
3
7
3 7
3 4 6 7
0 1 2 3 4 5 6 7
B
1 0 2 5
1 0
1
0
0 1
0 1 2 5
C
6 4
6
4
4 6
D
2 5
2
5
2 5

- Moving the cards around the room takes time (communication)
- Sorting the sub-piles is super-fast (computation)

Algorithm is bounded by communication : Difficult to scale

If the assignment was

- Consider the function $f(a,b,x) = a\,x^5 - \frac{b\,x^3}{a}$
- Each card has a value for a, b and x
- Evaluate $f(a,b,x)$ for each card and write the result on the card
- Sort the results

Same communication pattern, but more computational effort $\Longrightarrow$ better scaling.

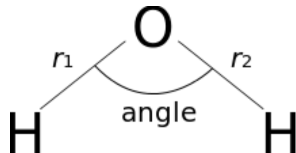
Take-home messages

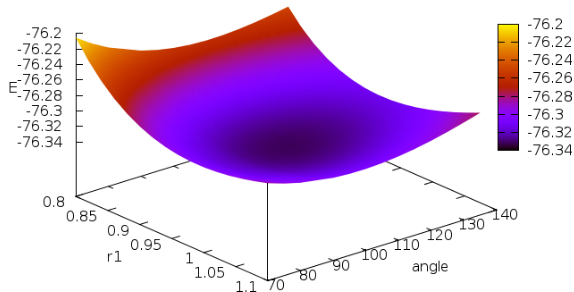
- Parallel workloads may require different algorithms than scalar workloads
- Difficulty is *data movement* (communication), not *computation*

Embarrassingly parallel computations

CC BY NC SA

We want to create the CCSD(T) potential energy surface of the water molecule.





cc BY NC SA

Constraints

- We want to compute $25 \times 25 \times 25 = 15\,625$ points
- We are allowed to use 100 CPU cores simultaneously
- We like to use GAU$$IAN to calculate the CCSD(T) energy

But:

- The grid points are completely independent
- Any CPU core can calculate any point

CC BY NC SA

Optimal solution: *work stealing*

- One grid point is $E(r_1, r_2, \theta)$
- Dress the list of all the arguments (r_1, r_2, θ): `[ (0.8,0.8,70.), ..., (1.1,1.1,140.) ]` (the queue)
- Each CPU core, when idle, pops out the head of the queue and computes $E(r_1, r_2, \theta)$
- All the results are stored in a single file
- The results are sorted for plotting

CC BY NC SA

GNU Parallel executes Linux commands in parallel and can guarantee that the output is the same as if the commands were executed sequentially.

```
$ parallel echo ::: A B C
A
B
C
```

is equivalent to:

```
$ echo A ; echo B ; echo C
```

Multiple input sources can be given:

```
$ parallel echo ::: A B ::: C D
A C
A D
B C
B D
```

If no command is given after parallel the arguments are treated as commands:

```
$ parallel ::: pwd hostname "echo $TMPDIR"
/home/scemama
lpqdh82.ups-tlse.fr
/tmp
```

Jobs can be run on remote servers:

```
$ parallel ::: "echo hello" hostname
lpqdh82.ups-tlse.fr
hello
$ parallel -S lpqlx139.ups-tlse.fr ::: "echo hello" hostname
hello
lpqlx139.ups-tlse.fr
```


CC BY NC SA

File can be transfered to the remote hosts:

```
1  $ echo Hello > input
2  $ parallel -S lpqlx139.ups-tlse.fr cat ::: input
3  cat: input: No such file or directory
4
5  $ echo Hello > input
6  $ parallel -S lpqlx139.ups-tlse.fr --transfer --cleanup cat ::: input
7  Hello
```

Convert thousands of images from .gif to .jpg

```
1  $ ls
2  img1000.gif  img241.gif  img394.gif  img546.gif  img699.gif  img850.gif
3  img1001.gif  img242.gif  img395.gif  img547.gif  img69.gif   img851.gif
4  [...]
5  img23.gif    img392.gif  img544.gif  img697.gif  img849.gif
6  img240.gif   img393.gif  img545.gif  img698.gif  img84.gif
```

To convert one .gif into .jpg format:

```
1  $ time convert img1.gif img1.jpg
2  real   0m0.008s
3  user   0m0.000s
4  sys    0m0.000s
```


CC BY NC SA

Sequential execution

```
1  $ time for i in {1..1011}
2  > do
3  > convert img${i}.gif img${i}.jpg
4  > done
5  real   0m7.936s
6  user   0m0.210s
7  sys    0m0.270s
```

Parallel execution on a quad-core:

```
1  $ time parallel convert {.}.gif {.}.jpg ::: *.gif
2  real   0m2.051s
3  user   0m1.000s
4  sys    0m0.540s
```


cc BY NC SA

1. Fetch the energy in an output file

Running a CCSD(T) calculation with GAU$$IAN gives the energy somewhere in the output::

```
CCSD(T)= -0.76329294074D+02
```

To get only the energy in the output, we can use the following command:

```
1  $ g09 < input | grep "^ CCSD(T)=" | cut -d "=" -f 2
2  -0.76329294074D+02
```


CC BY NC SA

2. Script `run_h2o.sh` that takes r_1, r_2 and θ as arguments

```
#!/bin/bash

r1=$1 ;  r2=$2 ; angle=$3

# Create Gaussian input file,  pipe it to Gaussian,
# and get the CCSD(T) energy
cat << EOF | g09 | grep "^ CCSD(T)=" | cut -d "=" -f 2
# CCSD(T)/cc-pVTZ

Water molecule r1=${r1} r2=${r2} angle=${angle}

0 1
h
o 1 ${r1}
h 2 ${r2} 1 ${angle}


EOF
```

Example:

```
$ ./run_h2o.sh 1.08 1.08 104.
-0.76310788178D+02

$ ./run_h2o.sh 0.98 1.0 100.
-0.76330291742D+02
```

3. Files containing the parameters

```
$ cat r1_file
  0.75
  0.80
  0.85
  0.90
  0.95
  1.00
```

```
$ cat nodefile
  2//usr/bin/ssh  compute-0-10.local
  2//usr/bin/ssh  compute-0-6.local
  16//usr/bin/ssh compute-0-12.local
  16//usr/bin/ssh compute-0-5.local
  16//usr/bin/ssh compute-0-7.local
  6//usr/bin/ssh  compute-0-1.local
  2//usr/bin/ssh  compute-0-13.local
  4//usr/bin/ssh  compute-0-8.local
```

```
$ cat angle_file
  100.
  101.
  102.
  103.
  104.
  105.
  106.
```

`nodefile` contains the names of the machines and the number of CPUs.

4. Run with GNU parallel

On a single CPU:

```
1  $ time parallel  -a r1_file  -a r1_file  -a angle_file \
2                   --keep-order  --tag  -j 1  $PWD/run_h2o.sh
3  0.75 0.75 100.        -0.76185942070D+02
4  0.75 0.75 101.        -0.76186697072D+02
5  0.75 0.75 102.        -0.76187387594D+02
6  [...]
7  0.80 1.00 106.        -0.76294078963D+02
8  0.85 0.75 100.        -0.76243282762D+02
9  0.85 0.75 101.        -0.76243869316D+02
10 [...]
11 1.00 1.00 105.        -0.76329165017D+02
12 1.00 1.00 106.        -0.76328988177D+02
13
14 real   15m5.293s
15 user   11m25.679s
16 sys    2m20.194s
```


cc BY NC SA

4. RUn with GNU parallel

On 64 CPUs: 39× faster!

```
1   $ time parallel  -a r1_file  -a r1_file  -a angle_file \
2                    --keep-order  --tag  --sshloginfile nodefile  $PWD/run_h2o.sh
3   0.75 0.75 100.       -0.76185942070D+02
4   0.75 0.75 101.       -0.76186697072D+02
5   0.75 0.75 102.       -0.76187387594D+02
6   [...]
7   0.80 1.00 106.       -0.76294078963D+02
8   0.85 0.75 100.       -0.76243282762D+02
9   0.85 0.75 101.       -0.76243869316D+02
10  [...]
11  1.00 1.00 105.       -0.76329165017D+02
12  1.00 1.00 106.       -0.76328988177D+02
13
14  real   0m23.848s
15  user   0m3.359s
16  sys    0m3.172s
```


CC BY NC SA

Inter-process communication

CC BY NC SA

Process

- Has its own memory address space
- Context switching between processes is slow
- Processes interact only through system-provided communication mechanisms
- Fork: creates a copy of the current process
- Exec: switches to running another binary executable
- Spawn: `Fork`, `exec` the child and `wait` for its termination

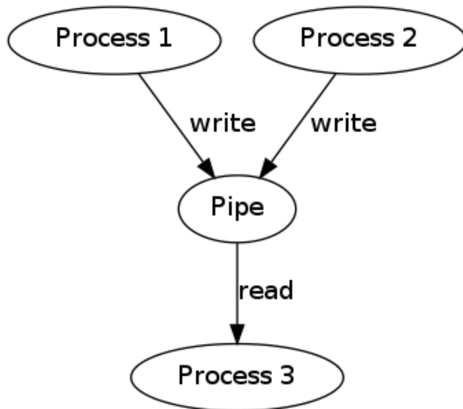
Thread

- Exist as subsets of a process
- Context switching between threads is fast
- Share the same memory address space : interact via shared memory

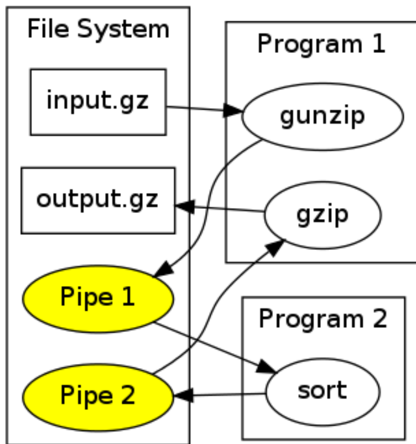
CC BY NC SA

- Processes exchange data via read/write on *File descriptors*
- These file descriptors can point to:
  - Files on disk: Simplest choice
  - Named Pipes: Same program as with files
  - Pipes: Same behavior as files
  - Network sockets

A named pipe is a virtual file which is read by a process and written by other processes.
It allows processes to communicate using standard I/O operations:



cc BY NC SA



- Unzip `input.gz`
- Sort the unzipped file
- Zip the result into `output.gz`

Same as

```
1  $ gunzip --to-stdout input.gz \
2        | sort \
3        | gzip > output.gz
```

```bash
#!/bin/bash

# Create two pipes using the mkfifo command
mkfifo /tmp/pipe /tmp/pipe2

# Unzip the input file and write the result
# in the 1st pipe
echo "Run gunzip"
gunzip --to-stdout input.gz > /tmp/pipe

# Zip what comes from the second pipe
echo "Run gzip"
gzip < /tmp/pipe2 > output.gz

# Clear the pipes in the filesystem
rm /tmp/pipe /tmp/pipe2
```

```bash
#!/bin/bash

# Read the 1st pipe, sort the result and write
# in the 2nd pipe
echo "Run sort"
sort < /tmp/pipe > /tmp/pipe2
```

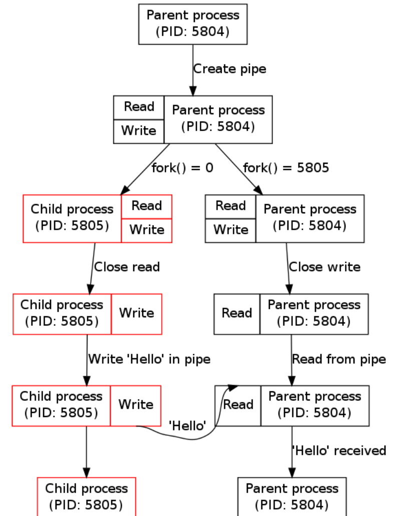
Execution

```
$ ./p1.sh &
Run gunzip
$ ./p2.sh
Run sort
Run gzip
[1]+ Done
./p1.sh
```

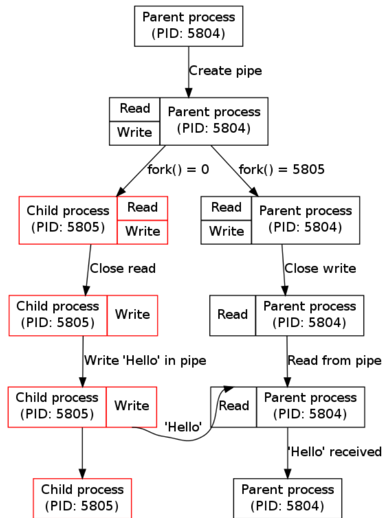

CC BY NC SA

Fork : Copy the current process
Pipes don't have an entry on the file system, and are opened/closed in the programs.

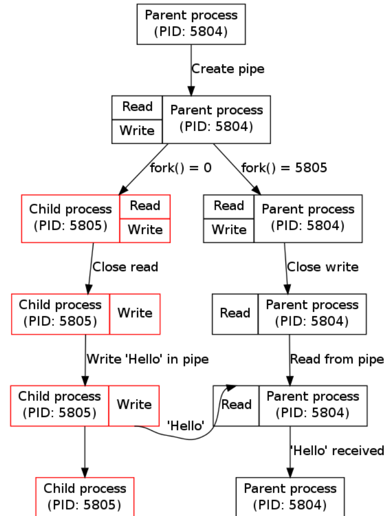
1. Create the pipe
2. Fork in parent/child processes
3. Exchange data between parent and child



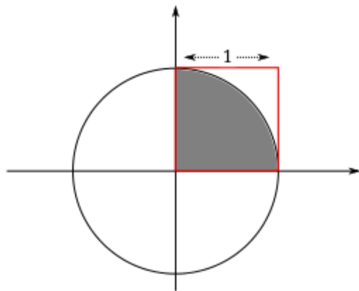
```python
#!/usr/bin/env python
import sys,os

def main():
  print("Process ID: %d" % (os.getpid()))
  r, w = os.pipe()
  new_pid = os.fork()

  if new_pid != 0:  # This is the parent process
      print("I am the parent, my PID is %d"%(os.getpid()))
      print("and the PID of my child is %d"%(new_pid))

      # Close write and open read file descriptors
      os.close(w)
      r = os.fdopen(r,'r')

      # Read data from the child
      print("Reading from the child")
      s = r.read()
      print("Read '%s' from the child"%(s))
      r.close() ;  os.wait()
```



```
1     else:   # This is the child process
2         print("  I am the child, my PID is %d"%(os.getpid()))
3
4         # Close read and open write file descriptors
5         os.close(r)
6         w = os.fdopen(w,'w')
7
8         # Send 'Hello' to the parent
9         print("  Sending 'Hello' to the parent")
10        w.write( "Hello!" )
11        w.close()
12
13        print("  Sent 'Hello'")
14
15  if __name__ == "__main__":
16      main()
```

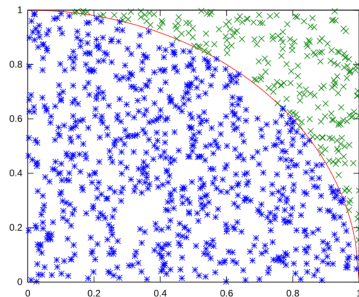


cc BY NC SA



- The surface of the circle is $\pi r^2 \implies$ For a unit circle, the surface is π
- The function in the red square is $y = \sqrt{1 - x^2}$ (the circle is $\sqrt{x^2 + y^2} = 1$)
- The surface in grey corresponds to

$$\int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4}$$

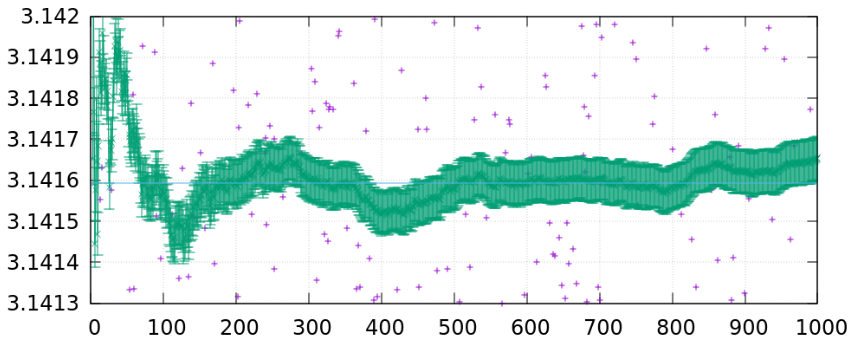


- Points (x, y) are drawn randomly in the unit square
- Count how many times the points are inside the circle

$$\frac{N_{\text{in}}}{N_{\text{in}} + N_{\text{out}}} = \frac{\pi}{4}$$

BY NC SA

- Each core $1 \leq i \leq M$ computes its own average X_i
- All M results are independent $\Longrightarrow$ Gaussian-distributed random variables (central-limit theorem)



```python
#!/usr/bin/env python
import os, sys
from random import random, seed
from math import sqrt

NMAX = 10000000              # Nb of MC steps/process
error_threshold = 1.0e-4     # Stopping criterion
NPROC=4                      # Use 4 processes

def compute_pi():
   """Local Monte Carlo calculation of pi"""
   seed(None)  # Initialize random number generator

   result = 0.
   for i in range(NMAX):          # Loop NMAX times
      x,y = random(), random()    # Draw 2 random numbers x and y
      if x*x + y*y <= 1.:         # Check if (x,y) is in the circle
         result += 1
   return 4.* float(result)/float(NMAX) # Estimate of pi
```

Computation of π with pipes in Python

```
 1  def main():
 2      r = [None]*NPROC       # Reading edges of the pipes
 3      pid = [None]*NPROC     # Running processes
 4
 5      for i in range(NPROC):
 6          r[i], w = os.pipe()     # Create the pipe
 7          pid[i] = os.fork()      # Fork and save the PIDs
 8
 9          if pid[i] != 0:         # This is the parent process
10              os.close(w)
11              r[i] = os.fdopen(r[i],'r')
12          else:                    # This is the child process
13              os.close(r[i])
14              w = os.fdopen(w,'w')
15              while True:         # Compute pi on this process
16                  X = compute_pi()
17                  try:
18                      w.write("%f\n"%(X))  # Write the result in the pipe
19                      w.flush()
20                  except IOError:        # Child process exits here
21                      sys.exit(0)
```


CC BY NC SA

```
1       data = []
2       while True:
3           for i in range(NPROC): # Read in the pipe of each process
4               data.append( float(r[i].readline()) )
5               N = len(data)
6               average = sum(data)/N                 # Compute average
7               if N > 2:                             # Compute variance
8                   l = [ (x-average)*(x-average) for x in data ]
9                   variance = sum(l)/(N-1.)
10              else:
11                  variance = 0.
12              error = sqrt(variance)/sqrt(N)        # Compute error
13              print("%f +/- %f %d"%(average, error, N))
14
15              if N > 2 and error < error_threshold:  # Stopping condition
16                  for i in range(NPROC):             # Kill all children
17                      try: os.kill(pid[i],9)
18                      except: pass
19                  sys.exit(0)
20
21  if __name__ == '__main__':  main()
```


cc BY NC SA

Computation of π with pipes in Python

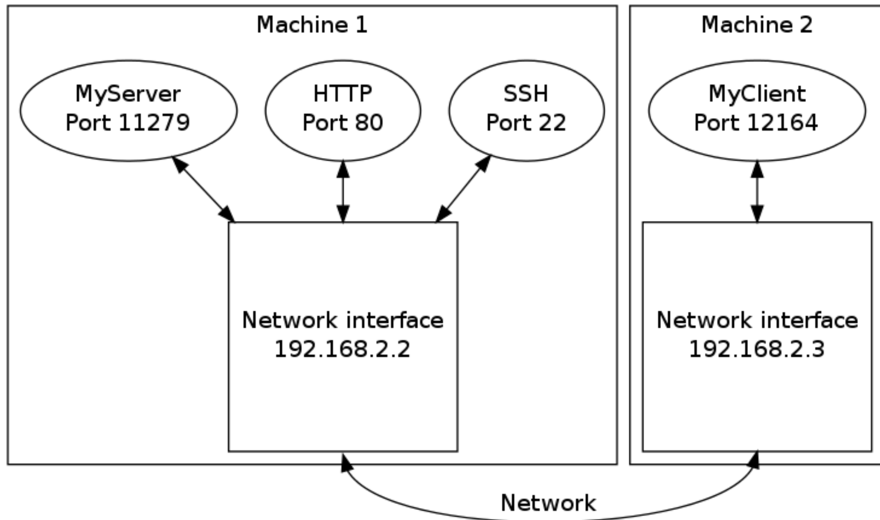
```
1   $ python pi_python.py
2   3.141974 +/- 0.000000 1
3   3.142569 +/- 0.000000 2
4   3.142168 +/- 0.000528 3
5   3.141938 +/- 0.000439 4
6   3.141947 +/- 0.000340 5
7   [...]
8   3.141625 +/- 0.000107 33
9   3.141614 +/- 0.000104 34
10  3.141617 +/- 0.000101 35
11  3.141606 +/- 0.000099 36
```


cc BY NC SA

Sockets are analogous to pipes, but they allow both ends of the pipe to be on different machines connected by a network interface. An Internet socket is characterized by a unique combination of:

- A transport protocol: TCP, UDP, raw IP, ...
- A local socket address: Local IP address and port number, for example 192.168.2.2:22
- A remote socket address: Optional (TCP)

CC BY NC SA


Machine 1
Machine 2
MyServer
Port 11279
HTTP
Port 80
SSH
Port 22
MyClient
Port 12164
Network interface
192.168.2.2
Network interface
192.168.2.3
Network

```python
#!/usr/bin/env python

import sys, os, socket, datetime
now = datetime.datetime.now

def main():
    HOSTNAME = socket.gethostname()
    PORT     = 11279
    print(now(), "I am the server : %s:%d"%(HOSTNAME,PORT))

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create an INET socket
    s.bind( (HOSTNAME, PORT) )                   # Bind the socket to the address and port
    s.listen(5)                                  # Wait for incoming connections
    conn, addr = s.accept()                      # Accept connection
    print(now(), "Connected by", addr)
```


CC BY NC SA

```python
1       data = ""
2       while True:                                    # Buffered read of the socket
3           message = conn.recv(8).decode('utf-8')
4           print(now(), "Buffer : "+message)
5           data += message
6           if len(message) < 8:  break
7       print(now(), "Received data : ", data)
8
9       print(now(), "Sending thank you...")
10      conn.send("Thank you".encode())
11      conn.close()
12
13  if __name__ == "__main__":
14       main()
```


CC BY NC SA

```python
#!/usr/bin/env python
import sys, os, socket, datetime
now = datetime.datetime.now

def main():
    HOSTNAME = sys.argv[1]          # Get host name from command line
    PORT     = int(sys.argv[2])     # Get port from command line
    print(now(), "The target server is : %s:%d"%(HOSTNAME,PORT))

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create an INET socket
    s.connect( (HOSTNAME, PORT) )  # Connect the socket to the address and port
    # of the server
    message = "Hello, world!!!!!!!"
    print(now(), "Sending : "+message)
    s.send(message.encode())  # Send the data

    data = s.recv(1024)  # Read the reply of the server
    s.close()
    print(now(), 'Received: ', data.decode('utf-8'))

if __name__ == "__main__": main()
```


CC BY NC SA

Sockets: Execution

```
1   $ python Codes/socket_client.py lpqdh82 11279
2   2021-11-20 21:20:32.258632 The target server is : lpqdh82:11279
3   2021-11-20 21:20:32.258959 Sending : Hello, world!!!!!!!
4   2021-11-20 21:20:32.259042 Connected by ('127.0.0.1', 36798)
5   2021-11-20 21:20:32.259068 Buffer : Hello, w
6   2021-11-20 21:20:32.259076 Buffer : orld!!!!
7   2021-11-20 21:20:32.259083 Buffer : !!!
8   2021-11-20 21:20:32.259088 Received data :  Hello, world!!!!!!!
9   2021-11-20 21:20:32.259093 Sending thank you...
10  2021-11-20 21:20:32.259133 Received:  Thank you
```

Note that the client and server can be written in different languages.

cc BY NC SA

```python
#!/usr/bin/env python
import sys, os,  socket
from math import sqrt
PORT      = 1666
error_threshold = 1.e-4   # Stopping criterion

def main():
   data = []
   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create an INET socket
   s.bind( (socket.gethostname(), PORT) )  # Bind the socket to the address and port
   while True:
      s.listen(5)                      # Wait for incoming connections
      conn, addr = s.accept()          # Accept connection
      X = ""
      while True:                      # Buffered read of the socket
         message = conn.recv(128)
         X += message.decode('utf-8')
         if len(message) < 128: break
      data.append( float(X) )
      N = len(data)
```

```
1          average = sum(data)/N        # Compute average
2          if N > 2:                    # Compute variance
3             l = [ (x-average)*(x-average) for x in data ]
4             variance = sum(l)/(N-1.)
5          else:
6             variance = 0.
7          error = sqrt(variance)/sqrt(N)     # Compute error
8
9          print('%f +/- %f'%(average,error))
10
11         # Stopping condition
12         if N > 2 and error < error_threshold:
13            conn.send("STOP".encode())
14            break
15         else:
16            conn.send("OK".encode())
17      conn.close()
18
19   if __name__ == "__main__":
20      main()
```

```python
#!/usr/bin/env python
import os, sys, socket
from random import random, seed
from math import sqrt
HOSTNAME = "localhost"
PORT     = 1666
NMAX     = 10000000         # Nb of MC steps/process
error_threshold = 1.0e-4    # Stopping criterion
NPROC=4                     # Use 4 processes

def compute_pi():
    """Local Monte Carlo calculation of pi"""
    seed(None)  # Initialize random number generator

    result = 0.
    for i in range(NMAX):             # Loop NMAX times
        x,y = random(), random()      # Draw 2 random numbers x and y
        if x*x + y*y <= 1.:           # Check if (x,y) is in the circle
            result += 1
    return 4.* float(result)/float(NMAX) # Estimate of pi
```

```python
def main():

    while True:
        X = compute_pi()
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # Create an INET socket
        try:     # Connect the socket to the address and port of the server
            s.connect( (HOSTAME, PORT) )
        except socket.error:
            break
        message = str(X)
        s.send(message.encode())                       # Send the data
        reply = s.recv(128).decode('utf-8')    # Read the reply of the server
        s.close()

        if reply == "STOP": break

if __name__ == '__main__':
  main()
```

Execution of π with sockets in Python

```
1  $ python pi_server_python.py &
2  > for i in {1..8} ; do
3  >     python pi_client_python.py &
4  > done ; wait
5
6  3.142136 +/- 0.000000
7  3.141783 +/- 0.000000
8  3.141992 +/- 0.000291
9  3.141804 +/- 0.000279
10 [...]
11 3.141687 +/- 0.000104
12 3.141666 +/- 0.000102
13 3.141651 +/- 0.000098
```

Message Passing Interface (MPI)

- Application Programming Interface for inter-process communication
- Takes advantage of HPC hardware:
  - TCP/IP: 50 μs latency
  - Remote Direct Memory Access (RDMA): <2 μs (low-latency network)
- Portable
- Each vendor has its own implementation adapted to the hardware
- Standard in HPC
- Initially designed for fixed number of processes:
  - No problem for the discovery of peers
  - Fast collective communications
- Single Program Multiple Data (SPMD) paradigm

CC BY NC SA

- Group of processes that can communicate together
- Each process has an ID in the communicator: no need for IP adresses and port numbers
- `MPI_COMM_WORLD`: Global communicator, default
- `size`: number of processes in the communicator
- `rank`: ID of the process in the communicator

Python

- Send: `comm.send(data, dest, tag)`
- Receive: `comm.recv(source, tag)`

Fortran

- Send: `MPI_SEND(buffer, count, datatype, destination, tag, communicator, ierror)`
- Receive: `MPI_RECV(buffer, count, datatype, source, tag, communicator, status, ierror)`

`MPI_Ssend` Blocking synchronous send. Returns upon notification that the message has been received (safe).

`MPI_Isend` Non-blocking send. Returns immediately (dangerous).

`MPI_Send` Returns when the send buffer is ready for use. Non-deterministic: depends on MPI implementation, size of the data, number of ranks, ... (dangerous)

Important

Writing the program and Debugging:

- *Always* use `MPI_Ssend`
- Use `MPI_Isend` only when `MPI_Ssend` is irrelevant

Once the code is well checked

- You can replace `MPI_Ssend` by `MPI_Send` as an optimization.
- If `MPI_Send` is still too slow, consider rewriting for `MPI_Isend`

CC BY NC SA

```python
from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        data = 42
        print("Before: Rank: %d    Size: %d    Data: %d"%(rank, size, data))
        comm.ssend(data, dest=1, tag=11)
        print("After : Rank: %d    Size: %d    Data: %d"%(rank, size, data))
    elif rank == 1:
        data = 0
        print("Before: Rank: %d    Size: %d    Data: %d"%(rank, size, data))
        data = comm.recv(source=0, tag=11)
        print("After : Rank: %d    Size: %d    Data: %d"%(rank, size, data))

if __name__ == "__main__": main()
```

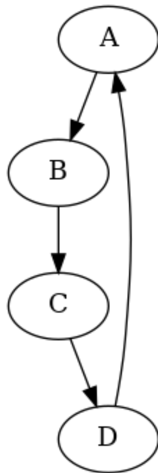
```
1  $ mpiexec -n 4 python mpi_rank.py
2  Before: Rank: 0    Size: 4    Data: 42
3  Before: Rank: 1    Size: 4    Data: 0
4  After : Rank: 0    Size: 4    Data: 42
5  After : Rank: 1    Size: 4    Data: 42
```

In Fortran, compile using `mpif90` and execute using `mpiexec` (or `mpirun`).

CC BY NC SA

```fortran
program test_rank
      use mpi
      implicit none
      integer :: rank, size, data, ierr, status(mpi_status_size)

      call MPI_INIT(ierr)       ! Initialize library (required)
      if (ierr /= MPI_SUCCESS) then
         call MPI_ABORT(MPI_COMM_WORLD, 1, ierr)
      end if

      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      if (ierr /= MPI_SUCCESS) then
         call MPI_ABORT(MPI_COMM_WORLD, 2, ierr)
      end if

      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      if (ierr /= MPI_SUCCESS) then
         call MPI_ABORT(MPI_COMM_WORLD, 3, ierr)
      end if
```

```
1      if (rank == 0) then
2          data = 42
3          print *, "Before: Rank:", rank, "Size:", size, "Data: ", data
4          call MPI_SSEND(data, 1, MPI_INTEGER, 1, 11, MPI_COMM_WORLD, ierr)
5          print *, "After : Rank:", rank, "Size:", size, "Data: ", data
6
7      else if (rank == 1) then
8          data = 0
9          print *, "Before: Rank:", rank, "Size:", size, "Data: ", data
10         call MPI_RECV(data, 1, MPI_INTEGER, 0, 11, MPI_COMM_WORLD, &
11                       status, ierr)
12         print *, "After : Rank:", rank, "Size:", size, "Data: ", data
13
14     end if
15     call MPI_FINALIZE(ierr)       ! De-initialize library (required)
16 end program
```



Deadlock

Each process waits for a message coming from another process

Example: round-robin

```fortran
program deadlock
  use mpi
  implicit none
  integer ::  rank, size, value, source, destination, ierr
  integer, parameter :: tag=100

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

  source      = mod(size+rank-1, size)
  destination = mod(rank+1     , size)

  call MPI_SSEND(rank+10, 1, MPI_INTEGER, destination, tag, MPI_COMM_WORLD, ierr)
  call MPI_RECV(value, 1, MPI_INTEGER, source, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)

  print *, rank, 'received', value, 'from', source
  call MPI_FINALIZE(ierr)
end program
```

The `MPI_Sendrecv` function sends and receives a message *simultaneously*. It can avoid deadlocks.

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,
             RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)

<type>     SENDBUF(*), RECVBUF(*)
INTEGER :: SENDCOUNT, SENDTYPE, DEST, SENDTAG
INTEGER :: RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM
INTEGER :: STATUS(MPI_STATUS_SIZE), IERROR
```

```fortran
program sendrecv
  use mpi
  implicit none
  integer ::  rank, size, value, source, destination, ierr
  integer, parameter :: tag=100

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

  source      = mod(size+rank-1, size)
  destination = mod(rank+1     , size)

  call MPI_SENDRECV(rank+10, 1, MPI_INTEGER, destination, tag, value, &
                    1, MPI_INTEGER, source, tag, MPI_COMM_WORLD,      &
                    MPI_STATUS_IGNORE, ierr)

  print *, rank, 'received', value, 'from', source

  call MPI_FINALIZE(ierr)
end program
```


cc BY NC SA

One-to-all

`MPI_Bcast` Broadcast the same data to all

`MPI_Scatter` distribute an array

All-to-one

`MPI_Reduce` Sum/product/... of data coming from all ranks

`Gather` collect a distributed array

All-to-all

`MPI_Barrier` Global synchronization

`MPI_AllReduce` Reduce and broadcast the result

`MPI_AllGather` Gather and broadcast the result

`MPI_Alltoall` Gather and scatter the result

CC BY NC SA

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx \sim 4 \sum_{i=1}^{M} \sqrt{1 - x_i^2}\, \Delta x$$

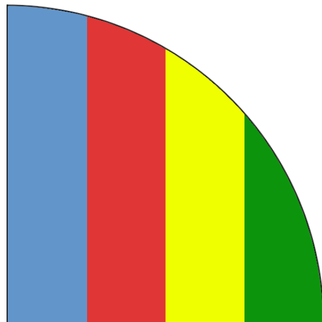
with $0 \leq x_i \leq 1$ and $\Delta x = x_{i+1} - x_i$

- Define a grid of M points
- Split the grid on N processes
- Each process computes part of the sum
- The partial sums are reduced on the master process

```
1  #!/usr/bin/env python
2  from mpi4py import MPI
3  import sys
4  from math import sqrt
5
6  def main():
7      comm = MPI.COMM_WORLD
8      rank = comm.Get_rank()
9      size = comm.Get_size()
10
11     M = int(sys.argv[1])                 # Total Number of grid points
12     M_local = (M-1) // size + 1          # Number of grid points to compute locally
13     istart = rank * M_local              # Beginning of interval
14     iend = min(istart + M_local, M)      # End of interval
15     dx = 1./float(M)                     # Delta x
16
17     sum = 0.
18     for i in range(istart, iend):
19         x = (i+0.5)*dx
20         sum += sqrt(1.-x*x)
```

```
1         result = 4. * dx * sum
2         pi = comm.reduce(result, MPI.SUM)
3         print("%10f -> %10f : %10f"%(istart*dx, iend*dx, result))
4         if rank == 0:
5             print("Result = ", pi)
6
7     if __name__ == "__main__": main()
```

```
1   $ mpiexec -n 4 python mpi_pi.py 100000000
2     0.000000 ->   0.250000 :   0.989483
3   Result =  3.1415926535902408
4     0.250000 ->   0.500000 :   0.923740
5     0.500000 ->   0.750000 :   0.775058
6     0.750000 ->   1.000000 :   0.453312
```



cc BY NC SA

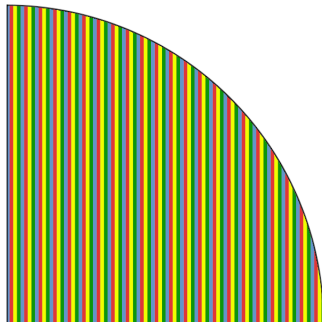
Alternate Computation of π with MPI (Python)

```
1  #!/usr/bin/env python
2  from mpi4py import MPI
3  import sys
4  from math import sqrt
5
6  def main():
7      comm = MPI.COMM_WORLD
8      rank = comm.Get_rank()
9      size = comm.Get_size()
10
11     M = int(sys.argv[1])                    # Total Number of grid points
12     dx = 1./float(M)                        # Delta x
13
14     sum = 0.
15     for i in range(rank, M, size):
16         x = (i+0.5)*dx
17         sum += sqrt(1.-x*x)
```

Computation of π with MPI (Python)

```python
    result = 4. * dx * sum
    pi = comm.reduce(result, MPI.SUM)
    print(rank, result)
    if rank == 0:
        print("Result = ", pi)

if __name__ == "__main__": main()
```

```
$ mpiexec -n 4 python mpi_pi_v2.py 100000000
0 0.7853981783959749
Result =  3.1415926535901777
2 0.7853981583983196
3 0.7853981483981102
1 0.7853981683977732
```



cc BY NC SA

Multi-threading

BY NC SA

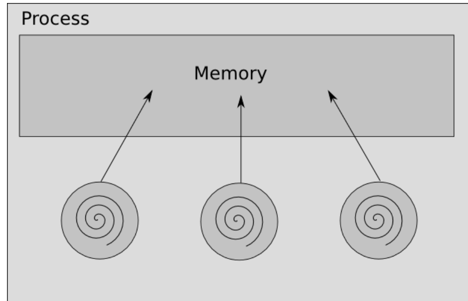
Process

- Has its own memory address space
- Context switching between processes is slow
- Processes interact only through system-provided communication mechanisms
- Fork: creates a copy of the current process
- Exec: switches to running another binary executable
- Spawn: `Fork`, then `exec` the child

Thread

- Exist as subsets of a process
- Context switching between threads is fast
- Share the same memory address space : interact via shared memory

cc BY NC SA



- Concurrent programming
- Graphical user interfaces (progress bars, . . .)
- Asynchronous I/O
- Standard library: POSIX threads (`pthreads`)

Communication time

- Low latency network latency : $\sim$ 1.2 microsecond
- Random memory access : $\sim$ 0.1 microsecond

CC BY NC SA

```python
#!/usr/bin/env python
import threading
import time

class test:
    def __init__(self, Nthreads):
        self.Nthreads = Nthreads
        self.data = [ i for i in range(Nthreads) ]

    def run_thread(self, j):
        self.data[j] = 0
        time.sleep(j)
        self.data[j] = j
```

```python
    def run(self):
        thread = [ None ] * self.Nthreads
        t0 = time.time()
        print(self.data)
        for i in range(self.Nthreads):
            thread[i] = threading.Thread( target=self.run_thread, args=(i,)  )
            thread[i].start()
        for i in range(self.Nthreads):
            thread[i].join()
            print(time.time()-t0, "seconds. ", self.data)

if __name__ == '__main__':
   t = test(4)
   t.run()
```

```
$ python thread_python.py
[0, 1, 2, 3]
0.0009775161743164062 seconds.  [0, 0, 0, 0]
1.0018701553344727 seconds.  [0, 1, 0, 0]
2.003377676010132 seconds.  [0, 1, 2, 0]
3.004056930541992 seconds.  [0, 1, 2, 3]
```

Computation of π with threads in Python

```
1  #!/usr/bin/env python
2  import os, sys, threading
3  from random import random, seed
4  from math import sqrt
5
6  NMAX = 10000000              # Nb of MC steps/process
7  error_threshold = 1.0e-4     # Stopping criterion
8
9  class pi_calculator:
10     def __init__(self, Nthreads):
11         self.Nthreads= Nthreads
12         self.results = []
13         self.lock = threading.Lock()
14
15     def compute_pi(self):
16         result = 0.
17         for i in range(NMAX):          # Loop NMAX times
18             x,y = random(), random()    # Draw 2 random numbers x and y
19             if x*x + y*y <= 1.:         # Check if (x,y) is in the circle
20                 result += 1
21         with self.lock:
22             self.results.append(4.* float(result)/float(NMAX))
```

```
1       def run(self):
2           thread = [None] * self.Nthreads
3           for i in range(self.Nthreads):
4               thread[i] = threading.Thread( target=self.compute_pi, args=()  )
5               thread[i].start()
6           print("All threads started")
7
8           while True:
9               for i in range(self.Nthreads):
10                  thread[i].join()
11              N = len(self.results)
12              average = sum(self.results)/N             # Compute average
13              if N > 2:                                 # Compute variance
14                  l = [ (x-average)*(x-average) for x in self.results ]
15                  variance = sum(l)/(N-1.)
16              else:
17                  variance = 0.
18              error = sqrt(variance)/sqrt(N)    # Compute error
19              print("%f +/- %f %d"%(average, error, N))
```

```
1             if N > 2 and error < error_threshold:   # Stopping condition
2                 return
3
4             for i in range(self.Nthreads):
5                 thread[i] = threading.Thread( target=self.compute_pi, args=()  )
6                 thread[i].start()
7
8     if __name__ == '__main__':
9         calc = pi_calculator(4)
10        calc.run()
```

Note: Inefficient in Python because of the Global Interpreter Lock (GIL), but you got the idea.

OpenMP

CC BY NC SA

- OpenMP is an extension of programming languages that enable the use of multi-threading to parallelize the code using directives.
- The OpenMP library may be implemented using `pthreads`
- Extensions in OpenMP 5.0 to offload code execution to GPUs
- The same source code can be executed with/without OpenMP

```
1    !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i)
2    !$OMP DO
3    do i=1,n
4       A(i) = B(i) + C(i)
5    end do
6    !$OMP END DO
7    !$OMP END PARALLEL
```

- `!$OMP PARALLEL` starts a new multi-threaded section. Everything inside this block is executed by *all* the threads
- `!$OMP DO` tells the compiler to split the loop among the different threads (by changing the loop boundaries for instance)
- `!$OMP END DO` marks the end of the parallel loop. It contains an implicit synchronization. After this line, all the threads have finished executing the loop.
- `!$OMP END PARALLEL` marks the end of the parallel section. Contains also an implicit barrier.
- `DEFAULT(SHARED)` : all the variables (A,B,C) are in shared memory by default
- `PRIVATE(i)` : the variable i is private to every thread

OpenMP directives (Fortran)

- `!$OMP CRITICAL ... !$OMP END CRITICAL` : all the statements in this block are protected by a lock
- `!$OMP TASK ... !$OMP END TASK` : define a new task to execute
- `!$OMP BARRIER` : synchronization barrier
- `!$OMP SINGLE ... !$OMP END SINGLE` : all the statements in this block are executed by a single thread
- `!$OMP MASTER ... !$OMP END MASTER` : all the statements in this block are executed by the master thread

cc BY NC SA

Functions

- `omp_get_thread_num()` : returns the ID of the current running thread (like `MPI_Rank`)
- `omp_get_num_threads()` : returns the total number of running threads (like `MPI_Size`)
- `OMP_NUM_THREADS` : Environment variable (shell) that fixes the number of threads to run

Important

- Multiple threads can read at the same memory address
- Multiple threads must not write at the same address

CC BY NC SA

Matrix multiplication

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```
1      do j=1,N
2        do i=1,N
3          C(i,j) = 0.d0
4          do k=1,N
5            C(i,j) = C(i,j) + A(i,k) * B(k,j)
6          end do
7        end do
8      end do
```


cc BY NC SA

Matrix multiplication with OpenMP

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```fortran
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE (i,j,k)
do j=1,N
  do i=1,N
    C(i,j) = 0.d0
    do k=1,N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end do
  end do
end do
!$OMP END PARALLEL DO
```

- Loop is parallelized over j
- Writing in `C(i,j)` is OK

CC BY NC SA

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```
1  !$OMP PARALLEL DEFAULT(SHARED) PRIVATE (i,j,k)
2  !$OMP DO COLLAPSE(2)
3  do j=1,N
4    do i=1,N
5      C(i,j) = 0.d0
6      do k=1,N
7        C(i,j) = C(i,j) + A(i,k) * B(k,j)
8      end do
9    end do
10 end do
11 !$OMP END DO
12 !$OMP END PARALLEL
```

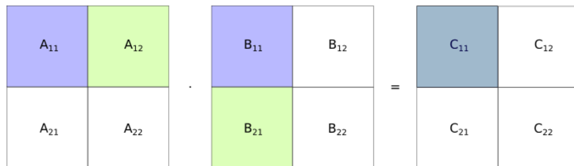
- Loop is parallelized over pairs `(i,j)`
- Writing in `C(i,j)` is OK

cc BY NC SA

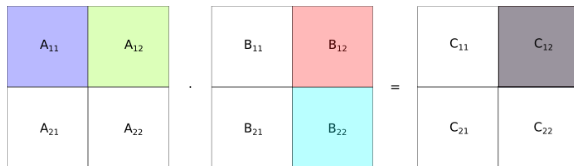
- Decompose the problem into similar subproblems
- Solve the smaller problems
- Compose their solutions to solve the given problem
- The subproblems are solved in the same way (recursion), unless the subproblems are simple enough to solve
- Example: merge sort

Divide and conquer example: matrix multiplication

$$A \cdot B = C$$



The $N \times N$ matrix multiplication can be performed by doing 8 smaller matrix multipications of size $N/2 \times N/2$, that can be done simultaneously.



$$\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{aligned}$$

cc BY NC SA

```fortran
step=sze/2
!$OMP DO COLLAPSE(2)
do i=1,sze,step
   do j=1,sze,step
      istart_C  = i  ;  iend_C = istart_C + step - 1
      istart_A  = i  ;  iend_A = istart_A + step - 1
      jstart_C  = j  ;  jend_C = jstart_C + step - 1
      jstart_B  = j  ;  iend_B = istart_B + step - 1
      do k=1,sze,step
         jstart_A = k  ; jend_A  = jstart_A + step - 1
         istart_B = k  ; iend_B  = istart_B + step - 1

         ! Compute the submatrix product
         call dgemm( 'N','N', 1+iend_C-istart_C, 1+jend_C-jstart_C, &
                     1+jend_A-jstart_A, 1.d0, A(istart_A,jstart_A), &
                     sze, B(istart_B,jstart_B), sze, 1.d0,            &
                     C(istart_C,jstart_C), sze )
      enddo
   enddo
enddo
!$OMP END DO
```

```fortran
recursive subroutine divideAndConquer(A, B, C, sze, ie1, je2)
[...]
  if ( (ie1 < 500).and.(je2 < 500) ) then
    call DGEMM     ! Call BLAS here because the matrix is small enough
  else

    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
    call divideAndConquer( & !  +-------+   +---+---+   +---+---+
        A(1,1),            & !  |   X   |   |   |   |   | X |   |
        B(1,1),            & !  +-------+ . + X |   + = +---+---+
        C(1,1),            & !  |       |   |   |   |   |   |   |
        sze,               & !  +-------+   +---+---+   +---+---+
        ie1/2,             & !      A           B           C
        je2/2)

    !$OMP END TASK
```

Recursive matrix multiplication (OpenMP)

```
1        !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
2        call divideAndConquer( & !  +-------+   +---+---+   +---+---+
3            A(1,1),            & !  |   X   |   |   |   |   |   | X |
4            B(1,1+je2/2),      & !  +-------+ . |   | X | = +---+---+
5            C(1,1+je2/2),      & !  |       |   |   |   |   |   |   |
6            sze,               & !  +-------+   +---+---+   +---+---+
7            ie1/2,             & !      A           B           C
8            je2-(je2/2))
9        !$OMP END TASK
10
11       !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
12       call divideAndConquer( & !  +-------+   +---+---+   +---+---+
13           A(1+ie1/2,1),      & !  |       |   |   |   |   |   |   |
14           B(1,1),            & !  +-------+ . | X |   | = +---+---+
15           C(1+ie1/2,1),      & !  |   X   |   |   |   |   | X |   |
16           sze,               & !  +-------+   +---+---+   +---+---+
17           ie1-(ie1/2),       & !      A           B           C
18           je2/2)
19       !$OMP END TASK
```


CC BY NC SA

```
1       !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
2       call divideAndConquer( & !  +-------+   +---+---+   +---+---+
3           A(1+ie1/2,1),      & !  |       |   |   |   |   |   |   |
4           B(1,1+je2/2),      & !  +-------+ . |   | X | = +---+---+
5           C(1+ie1/2,1+je2/2),& !  |   X   |   |   |   |   |   | X |
6           sze,               & !  +-------+   +---+---+   +---+---+
7           ie1-(ie1/2),       & !      A           B           C
8           je2-(je2/2))
9
10      !$OMP END TASK
11      !$OMP TASKWAIT
12    end if
13  end
14
15  subroutine mat_prod(A,B,C,LDA,m,n)
16    !$OMP PARALLEL DEFAULT(SHARED)
17    !$OMP SINGLE
18      call divideAndConquer(A,B,C,LDA,m,n)
19    !$OMP END SINGLE NOWAIT
20    !$OMP TASKWAIT
21    !$OMP END PARALLEL
22  end
```

Summary

- The architecture of supercomputers: why parallelism matters
- Amdahl's law (parallel speedup)
- Parallelizing shell commands (GNU Parallel)
- Inter-process communication: named pipes, pipes, sockets
- Message Passing Interface (MPI)
- Multi-threading (pthread)
- OpenMP: loops and tasks

Exercises

cc BY NC SA

Exercise 1: Distributed matrix-vector multiplication

```fortran
do i=1,m
   y(i) = 0.d0
   do j=1,n
      y(i) = y(i) + A(i,j) * x(j)
   end do
end do
```

$$y = A\,x$$

A $m \times n$ matrix

x vector of size n

y result: vector of size m

- Each y_i is a dot product
- All y_i are independent
- Parallel algorithm: Distribute the rows/columns of A on N_{proc} processes
- For testing, use $A_{ij} = \frac{1}{i+j-1}$ and $x_j = \frac{1}{j}$

cc BY NC SA

Variant 1

1. The rows of A are distributed: each process holds the sub-matrix $A_{\{i_a,i_b\}\{1,n\}}$
2. Broadcast the vector x with `MPI_Bcast`
3. Each process computes the sub-vector $y_{\{i_a,i_b\}}$
4. The result is transferred back to the master process (rank-0)

CC BY NC SA

Variant 2

1. The columns of A are distributed: each process holds the sub-matrix $A_{\{1,m\}\{j_a,j_b\}}$
2. Define a new MPI datatype for vectors of size $\frac{m-1}{N_{\text{proc}}} + 1$
3. Scatter the vector x with `MPI_Bcast` with the new data type for sub-vectors
4. Each process k computes a vector z_k
5. The result computes $\sum_k z_k$ on the master process (rank-0) using `MPI_Reduce`

1. Write a Fortran `double precision function compute_pi(M)` that computes π with the Monte Carlo algorithm using M samples
2. Call it using this main program:

```
program pi_mc
  implicit none
  integer                       :: M
  logical                       :: iterate
  double precision              :: sample
  double precision, external    :: compute_pi
  call random_seed()  ! Initialize random number generator
  open(unit=11, file='fortran_out.fifo', status='old', action='write', &
       access='stream',form='formatted')
  iterate = .True.
  do while (iterate)  ! Compute pi over N samples until 'iterate=.False.'
    sample = compute_pi()
    write(11,*) sample
  end do
end program pi_mc
```


CC BY NC SA

1. Write a Python server `pi_server.py` that receives samples of π in a socket and compute the running average of π. Its address and port number are written in a file `server.cfg`.
2. Write a Python script `pi_bridge.py` that reads samples of π in a named pipe `fortran_out.fifo` and sends the samples to the server. This script can read the the address and port number of the server from the file `server.cfg`.
3. When the convergence criterion is reached, the server informs the bridges that they can stop.

Running a simulation on multiple nodes

- Run a single server
- Run one bridge per compute node using the `mpiexec` command
- Run one Fortran process per core using the `mpiexec` command

CC BY NC SA