Parallel programming

Nicolas Renon
CALMIP (Toulouse)

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
http://scemama.mooo.com
Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Intro

All the source files of this course can be found on GitHub:

http://github.com/scemama/tccm2014

> ***Warning***
>
> You are not expected to be able to do by yourself everything I will show!

My goal:

- Show you different visions of parallel computing
- Introduce some words you will here in the future
- Show you what exists, what can be done, and how

Don't panic and consider this class as *general knowledge*.

If you don't understand something, please STOP ME!

What is parallelism?

When solving a problem, multiple calculations can be carried out concurrently. If multiple computing hardware is used, concurrent computing is called **parallel computing**.

Many levels of parallelism:

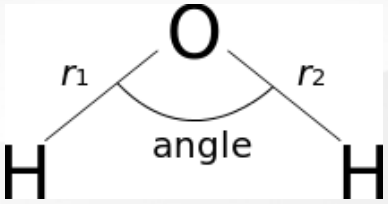
- Distributed, Loosely-coupled : Computing grids : shell scripts
- Distributed, Tightly-coupled : Supercomputers : MPI, sockets, CoArray Fortran, UPC,...
- Hybrid: wth accelerators like GPUs, FPGAs, Xeon Phi, etc
- Shared memory : OpenMP, threads
- Socket-level : Shared cache
- Instruction-level : superscalar processors
- Bit-level : vectorization

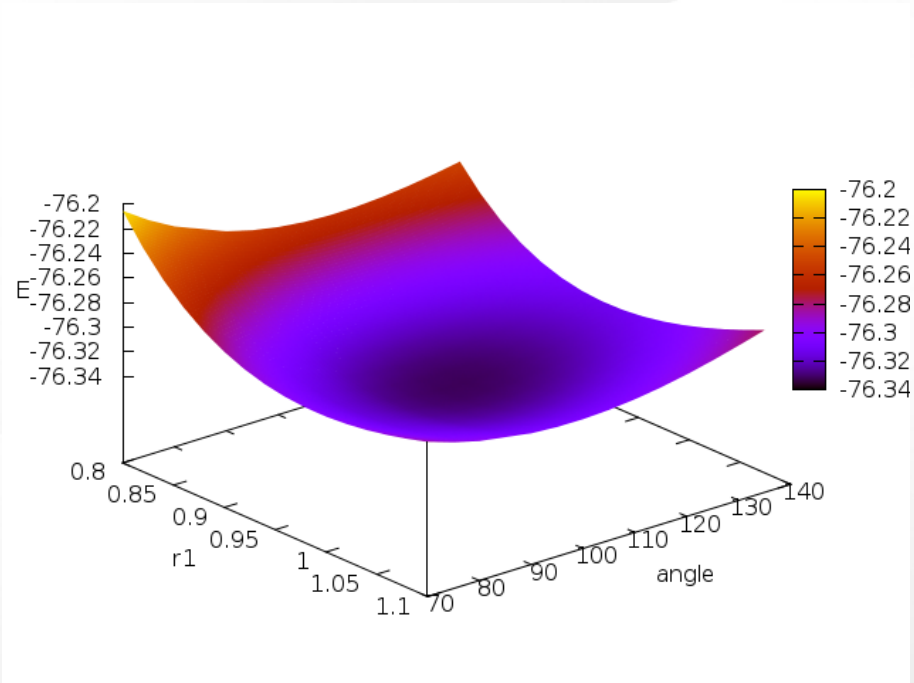
All levels of parallelism can be exploited in the same code, but every problem is not parallelizable at all levels.

Index

Problem 1 : Potential energy surface

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





Constraints:

- We want to compute 25x25x25 = 15625 points
- We are allowed to use 100 CPU cores simultaneously
- We like to use Gaussian09 to calculate the CCSD(T) energy

But:

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

Optimal solution: **work stealing**

- One grid point is E(r1,r2,angle)
- Dress the list of all the arguments (r1,r2,angle) : [(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(r1,r2,angle)
- All the results are stored in a single file
- The results are sorted for plotting

GNU Parallel

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

Example:

```
$ 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
/tmp
```

Jobs can be run on remote servers:

```
$ parallel ::: echo hostname
lpqdh82.ups-tlse.fr

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

File can be transfered to the remote hosts:

```
$ echo Hello > input
$ parallel -S lpqlx139.ups-tlse.fr cat ::: input
cat: input: No such file or directory

$ echo Hello > input
$ parallel -S lpqlx139.ups-tlse.fr --transfer --cleanup cat ::: input
Hello
```

Convert thousands of images from .gif to .jpg

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

To convert one *.gif* file to *.jpg* format:

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

Sequential execution:

```
$ time for i in {1..1011}
> do
> convert img${i}.gif img${i}.jpg
```

```
> done
real  0m7.936s
user  0m0.210s
sys   0m0.270s
```

Parallel execution on a quad-core:

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

Potential energy surface

1. Fetch the energy in an output file

Running a CCSD(T) calculation with Gaussian09 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:

```
g09 < input | grep "^ CCSD(T)=" | cut -d "=" -f 2
```

2. Script that takes r1, r2 and angle as arguments

We create a script *run_h2o.sh* that runs Gaussian09 for the water molecule taking r_1, r_2, and *angle* as command-line parameters, and prints the CCSD(T) energy:

```
#!/bin/bash

r1=$1
```

```
r2=$2
angle=$3

# Create Gaussian input file,  pipe it to Gaussian, grep 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 arguments

We prepare a file *r1_file* containing the *r* values:

```
0.75
0.80
0.85
0.90
0.95
1.00
```

then, a file *angle_file* containing the *angle* values:

```
100.
101.
```

```
102.
103.
104.
105.
106.
```

and a file *nodefile* containing the names of the machines and their number of CPUs:

```
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
```

4. Run with GNU parallel

Let's first run the job on 1 CPU:

```
$ time parallel -a r1_file -a r1_file -a angle_file \
  --keep-order --tag -j 1 $PWD/run_h2o.sh
0.75 0.75 100.         -0.76185942070D+02
0.75 0.75 101.         -0.76186697072D+02
0.75 0.75 102.         -0.76187387594D+02
[...]
0.80 1.00 106.         -0.76294078963D+02
0.85 0.75 100.         -0.76243282762D+02
0.85 0.75 101.         -0.76243869316D+02
[...]
1.00 1.00 105.         -0.76329165017D+02
1.00 1.00 106.         -0.76328988177D+02

real  15m5.293s
user  11m25.679s
```

```
sys   2m20.194s
```

Running in parallel on 64 CPUs with the *--keep-order* option, the output is the same, but it takes 39x less time!

```
$ time parallel -a r1_file -a r1_file -a angle_file \
  --keep-order --tag --sshloginfile nodefile $PWD/run_h2o.sh
0.75 0.75 100.         -0.76185942070D+02
0.75 0.75 101.         -0.76186697072D+02
0.75 0.75 102.         -0.76187387594D+02
[...]
0.80 1.00 106.         -0.76294078963D+02
0.85 0.75 100.         -0.76243282762D+02
0.85 0.75 101.         -0.76243869316D+02
[...]
1.00 1.00 105.         -0.76329165017D+02
1.00 1.00 106.         -0.76328988177D+02

real  0m23.848s
```

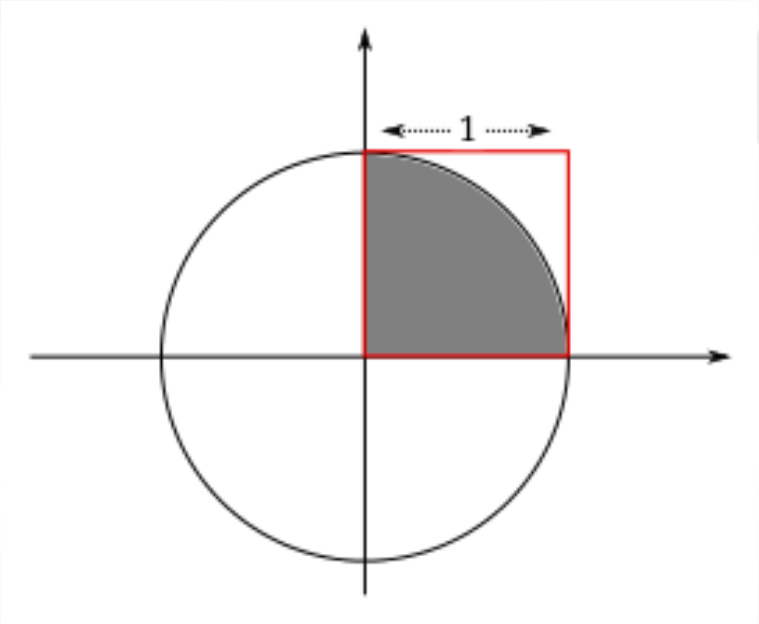
```
user   0m3.359s
sys    0m3.172s
```

Links

- O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.
- GNU parallel
- GNU parallel tutorial

Problem 2 : Computation of Pi

We want to compute the value of π with a Monte Carlo algorithm.

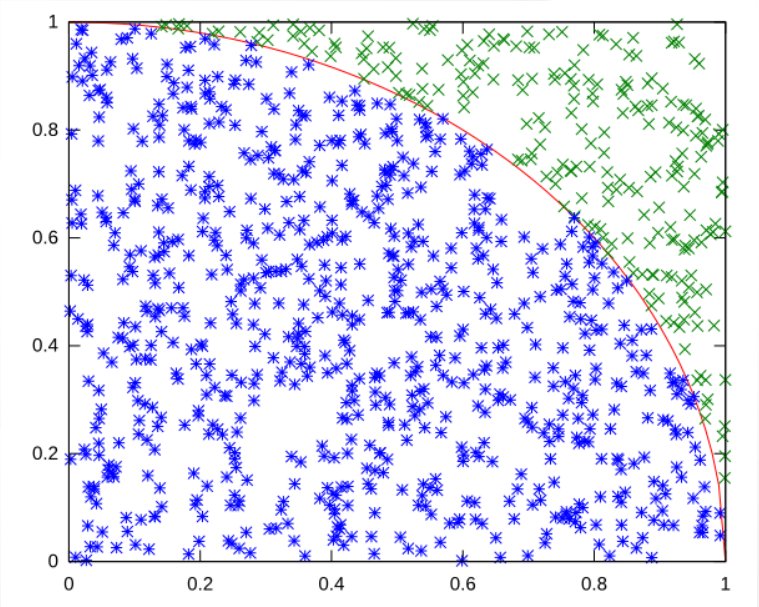


- The surface of the circle is πr^2 => For a unit circle, the surface is π
- The function in the red square is $y = 1 - x^2$ (the circle is $\sqrt{x^2 + y^2} = 1$)
- The surface in grey corresponds to

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

To compute this integral, a Monte Carlo algorithm can be used:

- Points (x,y) are drawn randomly in the unit square.
- Count how many times the points are inside the circle
- The ratio (inside)/(inside+outside) is $\pi/4$.



Constraints:

- A large number of Monte Carlo steps will be computed ($>10^{12}$)
- We are allowed to use 100 CPU cores simultaneously
- We stop when the statistical error is below a given threshold ($\sim 10^{-5}$)

Optimal algorithm:

- Each CPU core computes the its own average $X = 4N_{\text{in}}/N_{\text{total}}$ over a smaller number of Monte Carlo steps (10^7)

```
compute_pi() {
  result := 0
  for i=1 to NMAX {
    x = random() ; y = random()
    if ( x^2 + y^2 <= 1 ) {
      result := result + 1
    }
  }
  return 4*result/NMAX
```

```
}
```

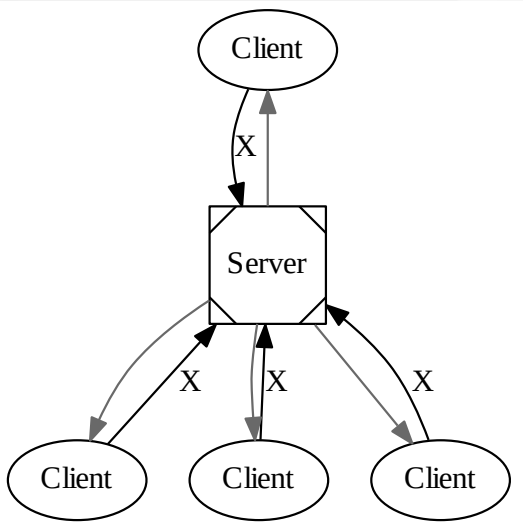
- All M results obtained on different CPU cores are independent, so they are Gaussian-distributed random variables (central-limit theorem)
- The X are sent to a central server
- The central server computes the running average

  $$\pi \sim \bar{X} = \frac{1}{M} \sum_{i=1}^{M} X_i$$

  and the variance

  $$\sigma^2 = \frac{1}{M-1} \sum_{i=1}^{M} (X_i - \bar{X})^2$$

  to compute the statistical error as $\delta\pi = \sigma / \sqrt{M}$
- The clients compute blocks as long as the central server asks them to do so when $\delta\pi$ is above the target error



Here, the calculations are no more independent: the stopping criterion depends on the results of all previous runs. We have introduced very simple **inter-process communications**.

Inter-process communication

Processes vs threads

Process:

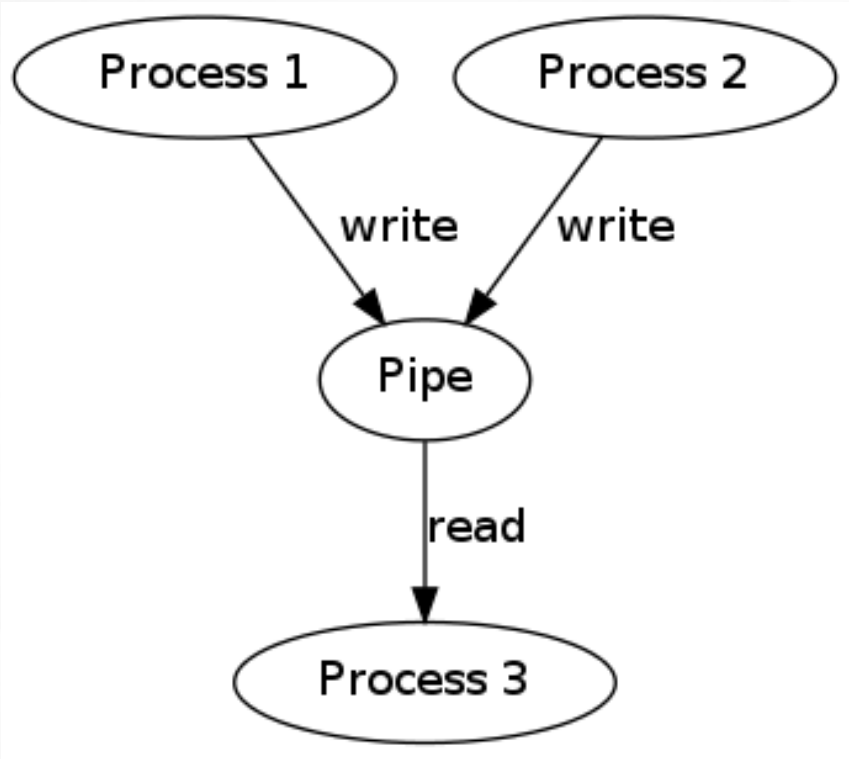
- Has its own memory address space
- Context switching between processes is slow
- Processes interact only through system-provided communication mechanisms
- Fork: creates a **copy** of the current process
- Exec: switches to running another binary executable
- Spawn: Fork and exec on the child

Theads:

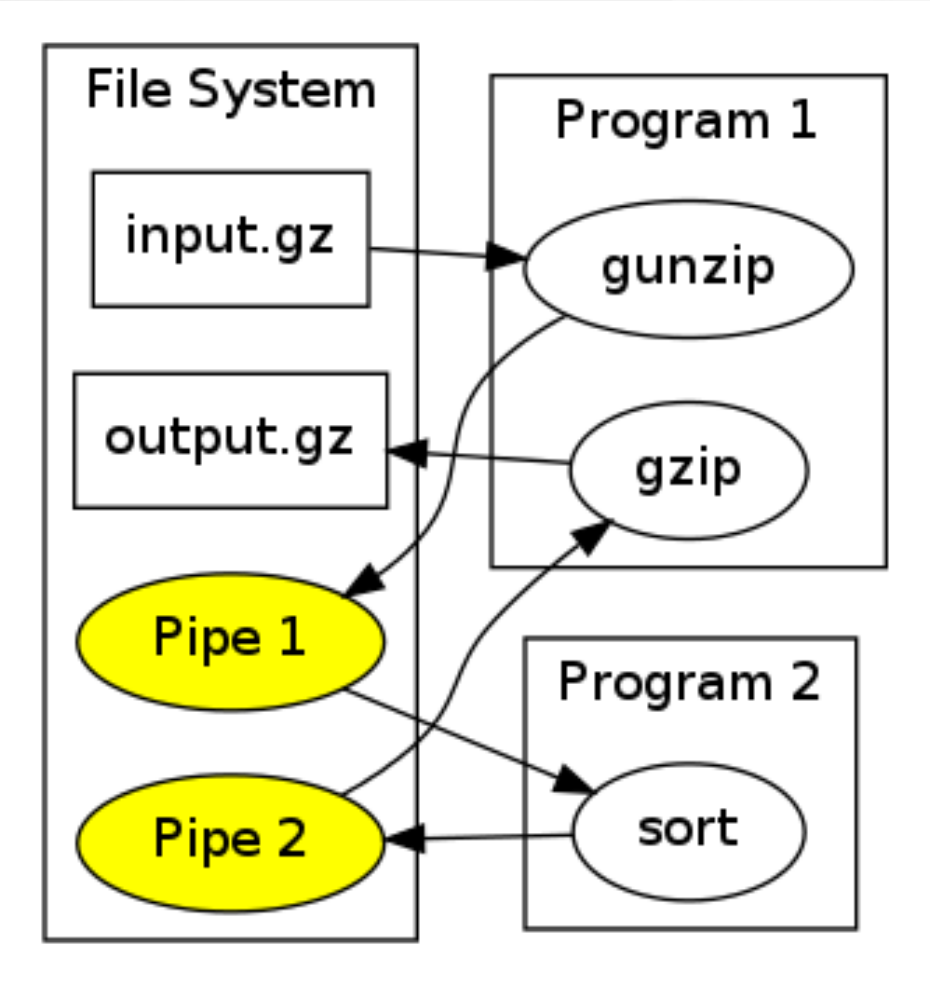
- Exist as subsets of a process
- Context switching between threads is fast
- Share the same memory address space : interact via shared memory

Communication with named pipes

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:



Example


File System
input.gz
output.gz
Pipe 1
Pipe 2
Program 1
gunzip
gzip
Program 2
sort

Process 1: *p1.sh*

```
#!/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
```

Process 2: *p2.sh*

```
#!/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
```

This simple example is equivalent to:

```
gunzip --to-stdout input.gz | sort | gzip > output.gz
```

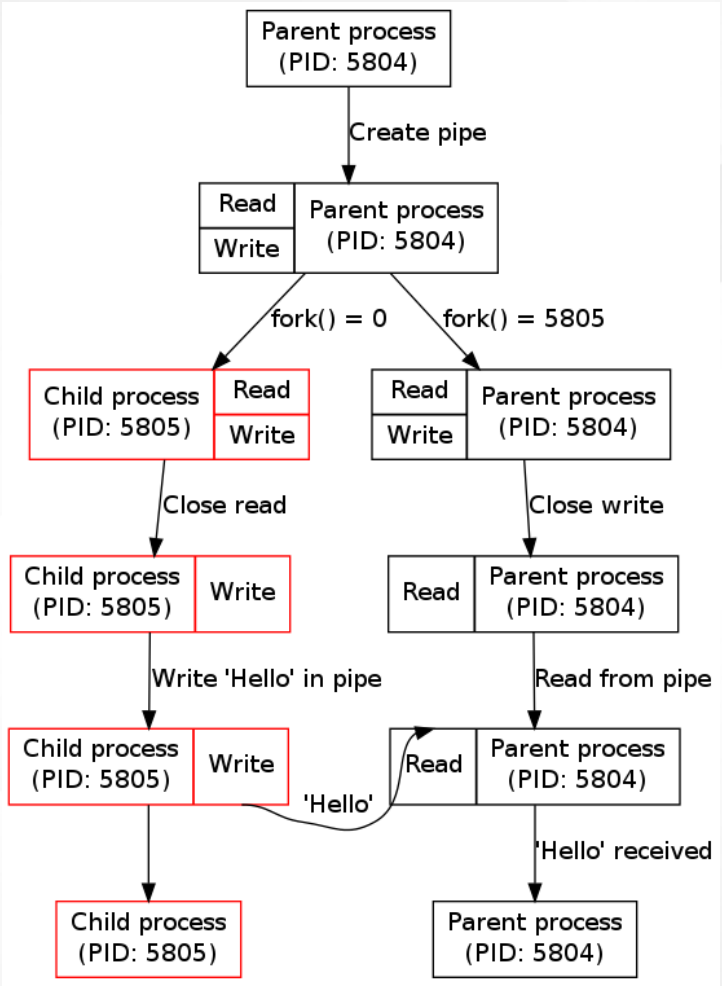
But the two programs *p1.sh* and *p2.sh*:

- can be started *independently* : p1 waits for p2 (blocking)
- can be run in different shells
- named pipes allow multiple processes to write in the same pipe

Communication with unnamed pipes

Unnamed pipes are equivalent to pipes, but they are opened/closed in the programs themselves. They imply a modification of the source files (apart from using unnamed pipes in the shell with the | operator).

Example


Parent process
(PID: 5804)
Create pipe
Read
Write
Parent process
(PID: 5804)
fork() = 0
fork() = 5805
Child process
(PID: 5805)
Read
Write
Read
Write
Parent process
(PID: 5804)
Close read
Close write
Child process
(PID: 5805)
Write
Read
Parent process
(PID: 5804)
Write 'Hello' in pipe
Read from pipe
Child process
(PID: 5805)
Write
Read
Parent process
(PID: 5804)
'Hello'
'Hello' received
Child process
(PID: 5805)
Parent process
(PID: 5804)

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

import sys,os

def main():
  # Print process ID (PID) of the current process
  print "PID: %d" % (os.getpid())

  # Open the pipe for inter-process communication
  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()
        r.close()
        print "Read '%s' from the child"%(s)

    else:
        # This is the child process
        print "  I am the child, my PID is %d"%(os.getpid())
        # Close read and open write file descriptors
        os.close(r)
        w = os.fdopen(w,'w')
        print "  Sending 'Hello' to the parent"
        # Send 'Hello' to the parent
        w.write( "Hello!" )
        w.close()
        print "  Sent 'Hello'"
```

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

```
$ ./fork.py
PID: 5804
I am the parent, my PID is 5804
and the PID of my child is 5805
  I am the child, my PID is 5805
Reading from the child
  Sending 'Hello' to the parent
  Sent 'Hello'
Read 'Hello!' from the child
```

Computation of π with pipes

Pseudo-code

```
for i=1 to NPROC {
  pipe(i) := create_pipe()
  fork()
  if ( Child process ) {
     close(pipe(i).read )
     open (pipe(i).write)
     do {
        X := compute_pi()
        write X into pipe
        if ( failure ) {
          exit process
        }
     }
  }
  close(pipe(i).write)
```

```
  open (pipe(i).read )
}

data := []
N := 0
do {
  for i=1 to NPROC {
    X := pipe(i).read()
    data := data+[X]
    N := N+1
    ave := average(data)
    err := error  (data)
    if (error < error_threshold) {
      print ave and err
      exit process
    }
  }
}
```

Python implementation

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

NMAX = 10000000              # Nb of MC steps/process
NMAX_inv = 1.e-7
error_threshold = 1.0e-4    # Stopping criterion
NPROC=4                      # Use 4 processes

import os
from random import random, seed
from math import sqrt

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

  result = 0.
```

```
  # Loop 10^7 times
  for i in xrange(NMAX):
    # Draw 2 random numbers x and y
    x = random()
    y = random()
    # Check if (x,y) is in the circle
    if x*x + y*y <= 1.:
      result += 1
  # X = estimation of pi
  result = 4.* float(result)*NMAX_inv
  return result


import sys

def main():
  # Reading edges of the pipes
  r = [None]*NPROC
```

```
# Running processes
pid = [None]*NPROC

for i in range(NPROC):
  # Create the pipe
  r[i], w = os.pipe()
  # Save the PIDs
  pid[i] = os.fork()
  if pid[i] == 0:
     # This is the child process
     os.close(r[i])
     w = os.fdopen(w,'w')
     while True:
       # Compute pi on this process
       X = compute_pi()
       # Write the result in the pipe
       try:
```

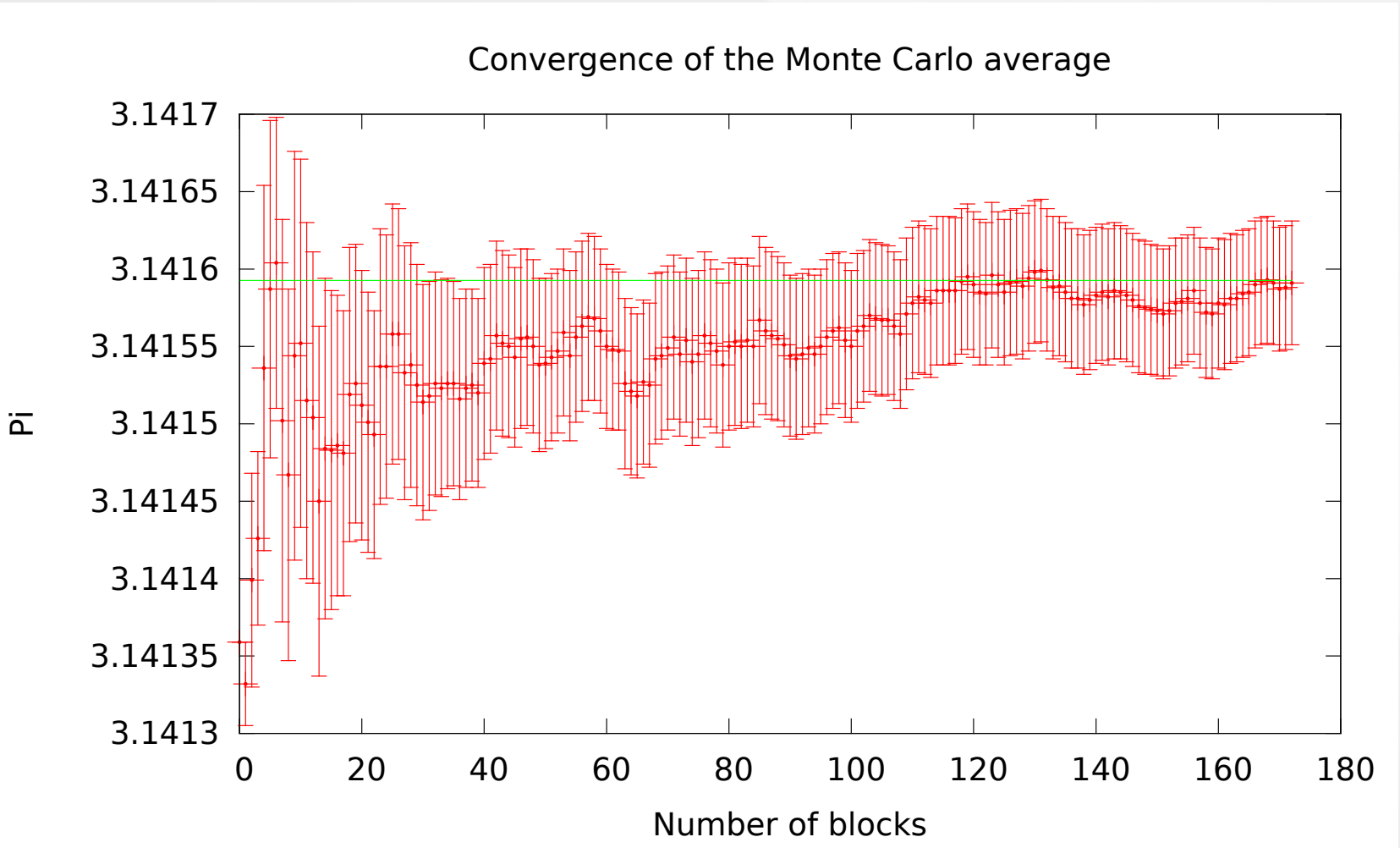
```
            w.write("%f\n"%(X))
            w.flush()
          except IOError:
            # Child process exits here
            sys.exit(0)
    else:
       # This is the parent process
       os.close(w)
       r[i] = os.fdopen(r[i],'r')


  data = []
  while True:
    for i in range(NPROC):
      # Read in the pipe of the corresponding process
      X =  float( r[i].readline() )
      data.append( float(X) )
      N = len(data)
```

```
# Compute average
average = sum(data)/N

# Compute variance
if N > 2:
  l = [ (x-average)*(x-average) for x in data ]
  variance = sum(l)/(N-1.)
else:
  variance = 0.

# Compute error
error = sqrt(variance)/sqrt(N)

print '%f +/- %f'%(average,error)

# Stopping condition
if N > 2 and error < error_threshold:
```

```
        # Kill children
        for i in range(NPROC):
          try: os.kill(pid[i],9)
          except: pass
        sys.exit(0)


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

```
$ ./pi_fork.py
3.142317 +/- 0.000000
3.141778 +/- 0.000000
3.141344 +/- 0.000534
3.141377 +/- 0.000379
3.141422 +/- 0.000297
3.141443 +/- 0.000243
3.141485 +/- 0.000210
```

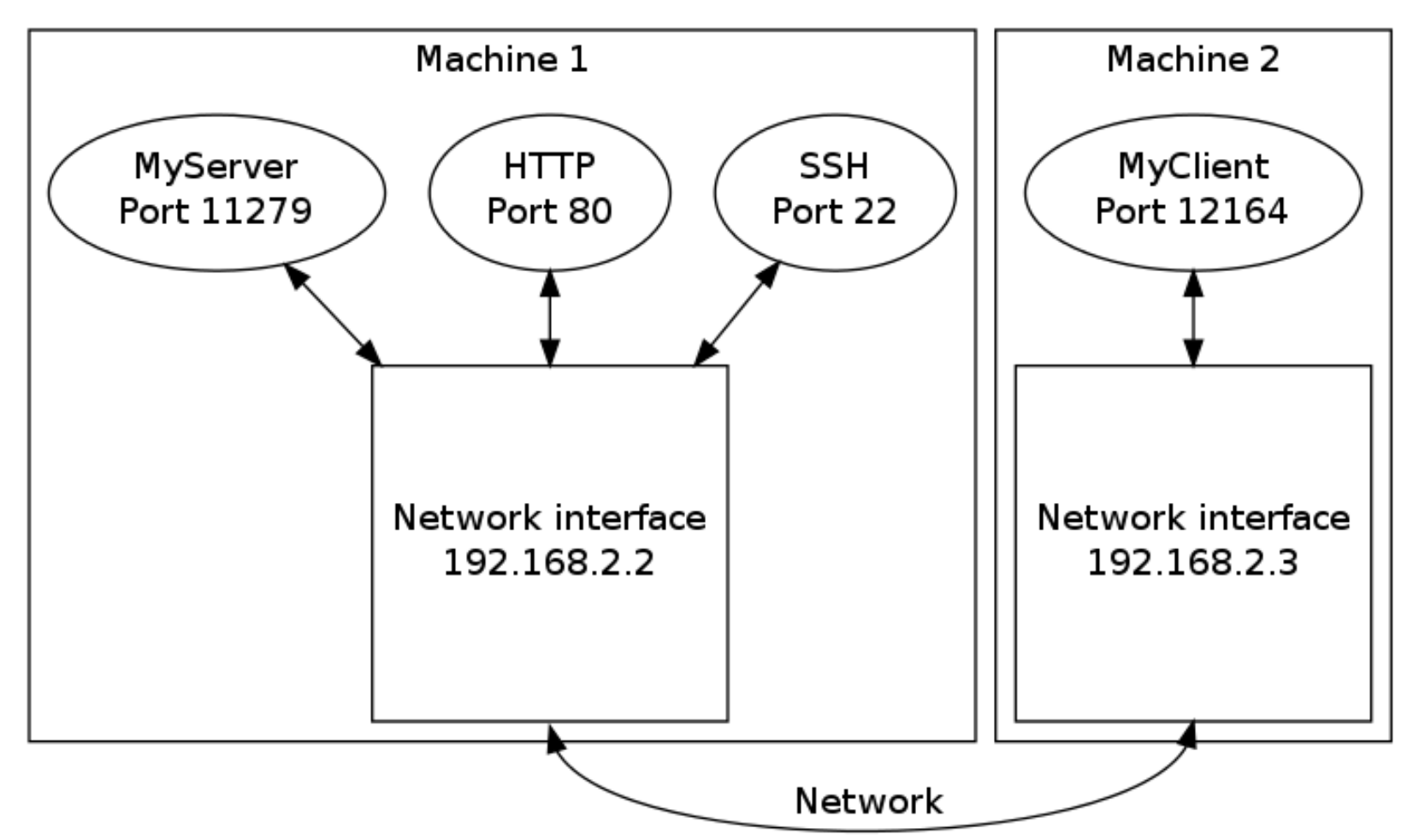
```
[...]
3.141513 +/- 0.000041
3.141513 +/- 0.000041
3.141514 +/- 0.000040
3.141512 +/- 0.000040
3.141513 +/- 0.000040
3.141515 +/- 0.000040
```


Convergence of the Monte Carlo average
Pi
3.1417
3.14165
3.1416
3.14155
3.1415
3.14145
3.1414
3.14135
3.1413
0
20
40
60
80
100
120
140
160
180
Number of blocks

Sockets

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)


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

Pseudo-code

Server code:

```
HOSTNAME := "server.tccm.fr"
PORT := 2014
socket := create_INET_socket()
bind( socket, (HOSTNAME, PORT) )
listen(socket)
(client_socket, address) := accept(socket)
data = recv(client_socket)
send(client_socket,"Thank you")
close(client_socket)
```

Client code:

```
HOSTNAME := "server.tccm.fr"
PORT := 2014
socket := create_INET_socket()
connect( socket, (HOSTNAME, PORT) )
message = "Hello, world !!!!!!"
```

```
send(socket,message)
reply = recv(socket)
```

Python implementation

Server code:

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

import sys,os
import socket
import datetime # For printing the time

now = datetime.datetime.now

def main():
  # Get host name
  HOSTNAME = socket.gethostname()
  PORT     = 11279
```

```
print now(), "I am the server : %s:%d"%(HOSTNAME,PORT)

# Create an INET socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
s.bind( (HOSTNAME, PORT) )

# Wait for incoming connections
s.listen(5)

# Accept connection
conn, addr = s.accept()
print now(), "Connected by", addr

# Buffered read of the socket
print now(), "Reading from socket"
data = ""
```

```
  while True:
    message = conn.recv(8)
    print now(), "Buffer : "+message
    data += message
    if message == "" or len(message) < 8: break
  print now(), "Received data : ", data

  print now(), "Sending thank you..."
  conn.send("Thank you")
  print now(), "Closing socket"
  conn.close()


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

Client code:

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

import sys,os
import socket
import datetime

now = datetime.datetime.now

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

  # Create an INET socket
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

  # Connect the socket to the address and port of the server
```

```
  s.connect( (HOSTNAME, PORT) )

  # Send the data
  message = "Hello, world !!!!!!"
  print now(), "Sending : "+message
  s.send(message)

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

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

Server execution:

```
$ ./sock_server.py
2014-09-04 01:13:49.903443 I am the server : lpqdh82:11279
```

```
2014-09-04 01:13:53.387956 Connected by ('127.0.0.1', 44373)
2014-09-04 01:13:53.388007 Reading from socket
2014-09-04 01:13:53.388029 Buffer : Hello, w
2014-09-04 01:13:53.388046 Buffer : orld !!!
2014-09-04 01:13:53.388060 Buffer : !!!
2014-09-04 01:13:53.388071 Received data :  Hello, world !!!!!!
2014-09-04 01:13:53.388081 Sending thank you...
2014-09-04 01:13:53.388157 Closing socket
```

Client execution:

```
$ ./sock_client.py lpqdh82 11279
2014-09-04 01:13:53.387347 The target server is : lpqdh82:11279
2014-09-04 01:13:53.387880 Sending : Hello, world !!!!!!
2014-09-04 01:13:53.388277 Received:  Thank you
```

Computation of π with sockets

Server:

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

HOSTNAME = "localhost"
PORT     = 1666
error_threshold = 4.e-5   # Stopping criterion

import sys,os
import socket
from math import sqrt

def main():
  data = []

  # Create an INET socket
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# Bind the socket to the address and port
s.bind( (HOSTNAME, PORT) )

while True:
  # Wait for incoming connections
  s.listen(5)

  # Accept connection
  conn, addr = s.accept()

  # Buffered read of the socket
  X = ""
  while True:
    message = conn.recv(128)
    X += message
    if message == "" or len(message) < 128: break
```

```
    data.append( float(X) )
    N = len(data)

    # Compute average
    average = sum(data)/N

    # Compute variance
    if N > 2:
      l = [ (x-average)*(x-average) for x in data ]
      variance = sum(l)/(N-1.)
    else:
      variance = 0.

    # Compute error
    error = sqrt(variance)/sqrt(N)

    print '%f +/- %f'%(average,error)
```

```
    # Stopping condition
    if N > 2 and error < error_threshold:
      conn.send("STOP")
      break
    else:
      conn.send("OK")

    conn.close()

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

Client:

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

NMAX = 10000000              # Nb of MC steps/process
NMAX_inv = 1.e-7
HOSTNAME = "localhost"
```

```
PORT      = 1666

from random import random, seed
import socket
import sys

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

  result = 0.
  # Loop 10^7 times
  for i in xrange(NMAX):
    # Draw 2 random numbers x and y
    x = random()
    y = random()
    # Check if (x,y) is in the circle
```

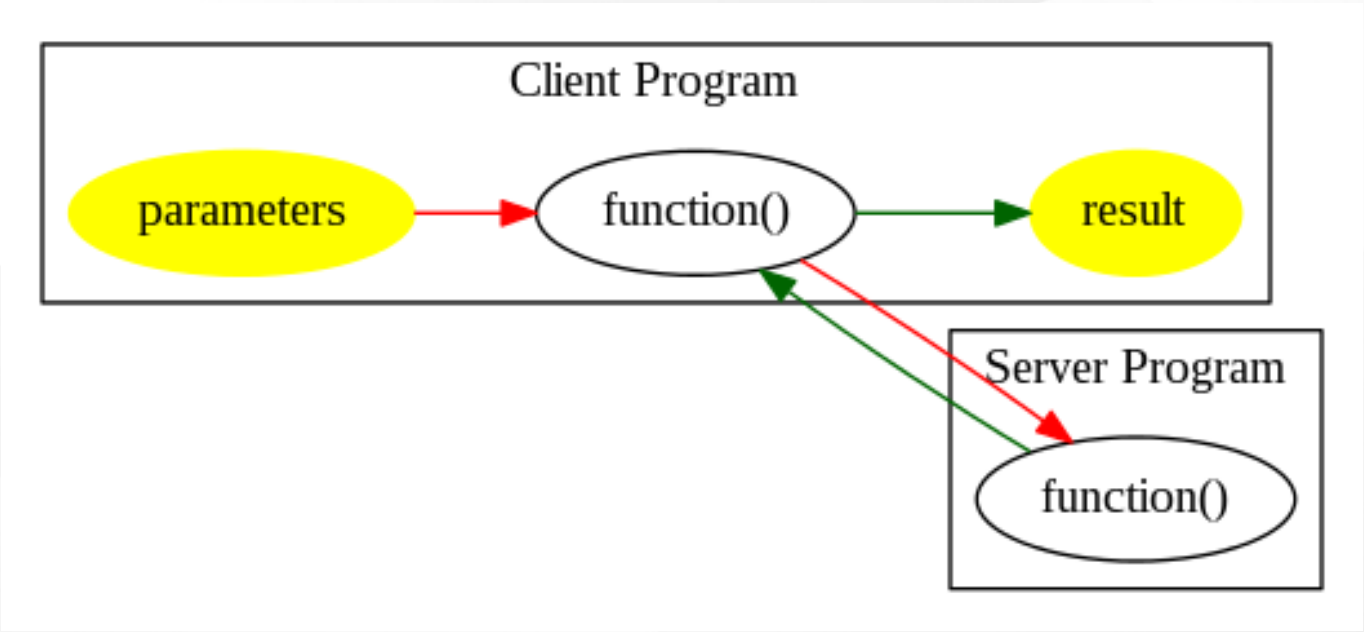
```
    if x*x + y*y <= 1.:
      result += 1
  # X = estimation of pi
  result = 4.* float(result)*NMAX_inv
  return result

def main():

  while True:
    X = compute_pi()

    # Create an INET socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the address and port of the server
    try:
      s.connect( (HOSTNAME, PORT) )
    except socket.error:
```

```
      break

    # Send the data
    message = str(X)
    s.send(message)

    # Read the reply of the server
    reply = s.recv(128)
    s.close()

    if reply == "STOP":
      break

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

Remote procedure call (RPC)

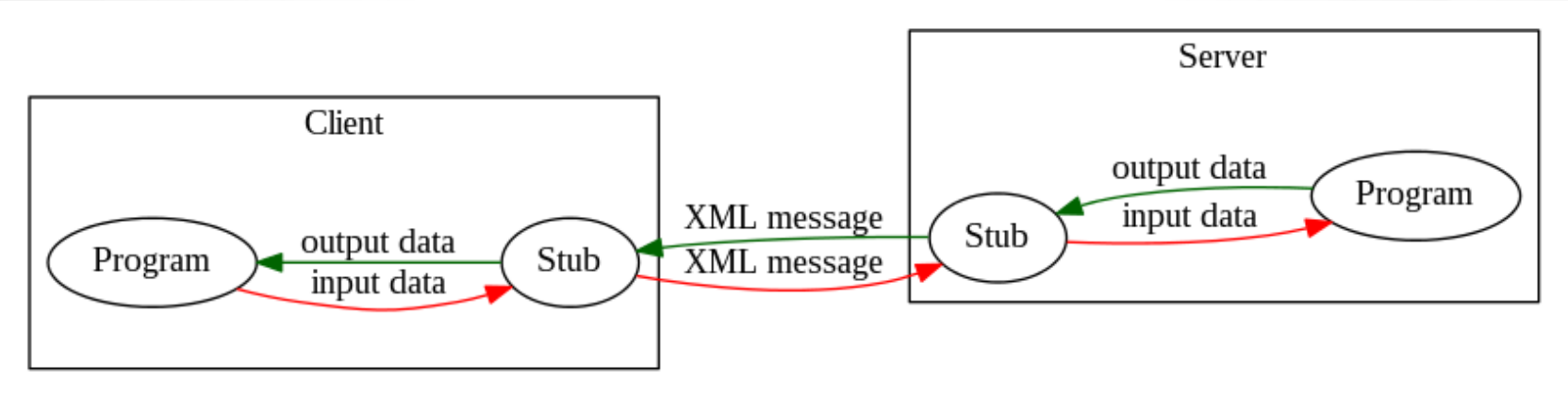
RPC enables software written in different languages and running on different computers to work with each other seamlessly.

One program running in a process (the client) calls a function belonging to another program running in another process (the server).



All the inter-process communication is hidden.

1. The client calls the *stub* : the parameters are converted to a standard representation (de-referencing pointers, big/little endian, etc)
2. The client stub *marshals* the parameters : they are packed together in a message.
3. The message is sent to the server
4. The server transmits the message to the server stub
5. The server stub *unmarshals* the message
6. The server calls its subroutine with the parameters
7. The output is sent back to the client using the same mechanism



Some RPC implementations:

- **XML-RPC**: XML is the encoding format and HTTP is the transport protocol
- **JSON-RPC**: JSON is the encoding format and HTTP is the transport protocol
- **SOAP**: Simple Object Access Protocol. Uses XML for encoding, but can use HTTP, HTTPS, SMTP, UDP, ... transport protocols
- **CORBA**: Common Object Request Broker Architecture
- etc...

XML-RPC simple example

Pseudo-code

Server code

```
function_1(x1)    { ... }
function_2(y1,y2) { ... }

server := create_XML_RPC_server( (HOSTNAME, PORT) )
server.register ( function_1, function_2 )
server.start()
```

Client code

```
server := connect_XML_RPC_server( (HOSTNAME,PORT) )

result_1 := server.function_1(x1)
result_2 := server.function_2(y1,y2)
```

Python implementation

Server code

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

import SimpleXMLRPCServer
import socket


class MyServer(object):

  def hostname(self):
      """Returns the name of the host on which the server runs"""
      return socket.gethostname()

  def split(self, string):
      """Splits a string in a list of words"""
      return string.split()
```

```
def main():
    # Display the name of the server in the standard output
    host = socket.gethostbyname( socket.gethostname() )
    port = 8000
    print "Server URL is http://%s:%d"%(host,port)

    # Create an instance of the server
    server = SimpleXMLRPCServer.SimpleXMLRPCServer( (host, port) )

    # Associate all functions of MyServer with the server
    server.register_instance( MyServer() )

    # Start the server
    server.serve_forever()


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

Client code

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

from socket import gethostname
import sys
import xmlrpclib   # XML-RPC library

def main():
  host = gethostname()
  print 'This host is: %s'%(host)

  # The URL of the server is the 1st argument of the command line
  url = sys.argv[1]

  # Create a proxy object for the server
  server = xmlrpclib.Server(url)

  # Run the 'hostname' function on the server and print the output
  remote = server.hostname()
```

```
  print 'Remote host is: %s'%(remote)

  # Run the 'split' function on the server and print the output
  s = "This is the string to split"
  splitted = server.split(s)
  print 'Splitted string has type:', type(splitted)
  print str(splitted)


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

Execution

```
scemama@lpqdh82 $ ./xmlrpc_server.py
Server URL is http://192.168.2.8:8000
lpqdh82 - - [29/Jul/2014 01:08:06] "POST /RPC2 HTTP/1.1" 200 -
lpqdh82 - - [29/Jul/2014 01:08:06] "POST /RPC2 HTTP/1.1" 200 -
```

```
scemama@pi $ ./xmlrpc_client.py http://192.168.2.8:8000
This host is: pi
Remote host is: lpqdh82
Splitted string has type: <type 'list'>
['This', 'is', 'the', 'string', 'to', 'split']
```

Monte Carlo Calculation of π with XML-RPC

Pseudo-code

Server code:

```
data = []
server_is_running := False

subroutine set_result( X ) {
    data := data + [X]
    if ( get_error() <= error_threshold ) {
      server_is_running := False
    }
}

function get_average() {
    return sum(data) / ( length(data) )
}
```

```
function get_variance() {
    average := get_average()
    v := 0
    for all x in data {
      v := variance + (x-average)^2
    }
    return v/(length(data)-1)
}

function get_error() {
    return sqrt( get_variance() / ( length(data) ) )
}

server := create_XML_RPC_server( (HOSTNAME, PORT) )
server.register ( set_result )
server.start()
server_is_running := True
while (server_is_running) {
```

```
    server.handle_request()
}

print get_average(), get_error()
```

Client code:

```
function compute_pi() {
    ...
}

server := connect_XML_RPC_server( (HOSTNAME,PORT) )

loop := True
while (loop) {
    X := compute_pi()
    reply := server.set_result(X)
    loop := ( reply = "CONTINUE" )
}
```

Python implementation

Server code:

```
#!/usr/bin/python -u

from SimpleXMLRPCServer import SimpleXMLRPCServer
from math import sqrt
from time import gmtime, strftime

# Termination condition
error_threshold = 1.e-4

class PiServer(object):

  def __init__(self):
      """Initialization of the server"""
      # Data is stored in a list
      self.data = []
```

```
        # N is the number of random events
        self.N    = 0

    def set_result(self,value,address):
        """Adds a value coming from a given host"""
        self.data.append( value )
        self.N += 1
        # Termination condition is calculated now
        if self.N > 4 and self.error() < error_threshold:
          self.terminate()
          result = 0
        else:
          result = 1
        # Each time a new event is added, display the
        # current average and error
        self.print_status(address)
        return result
```

```
  def terminate(self):
      """Terminate the run"""
      global running
      running = False

  def average(self):
      """Computes the running average"""
      return sum(self.data)/self.N

  def variance(self):
      """Computes the variance"""
      x_ave = self.average()
      l = [ (x-x_ave)*(x-x_ave) for x in self.data ]
      if self.N < 2:
        return 0.
      return sum(l)/(self.N-1)

  def error(self):
```

```
        """Computes the error bar"""
        return sqrt(self.variance())/sqrt(self.N)

    def print_status(self,address):
        """Displays something like:
        [ 15:39:59  127.0.0.1 ] : 3.141336  +/-  0.000120  (   7)
        """
        time = strftime("%H:%M:%S", gmtime())
        print "[ %8s  %15s ] : %f  +/-  %f  (%4d)"%(time, address,
            self.average(), self.error(),self.N)



running = True

from socket import gethostbyname, gethostname
import sys
```

```
def main():
    # Print the URL and port number of the server
    host = gethostbyname( gethostname() )
    port = 8000
    print >>sys.stderr, "Server URL is http://%s:%d"%(host,port)

    # Create the server
    server = SimpleXMLRPCServer( (host, port), logRequests=False )

    # All functions of PiServer are accessible via XML-RPC
    server.register_instance( PiServer() )

    # Run while the global variable 'running' is True
    while running:
       server.handle_request()

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

Client code:

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

# Compute X as an average over 10^7 MC steps
NMAX = 10000000
NMAX_inv = 1.e-7

from random import random, seed

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

  result = 0.
  # Loop 10^7 times
  for i in xrange(NMAX):
    # Draw 2 random numbers x and y
    x = random()
```

```
    y = random()
    # Check if (x,y) is in the circle
    if x*x + y*y <= 1.:
      result += 1
  # X = estimation of pi
  result = 4.* float(result)*NMAX_inv
  return result


import sys
import xmlrpclib
from socket import gethostbyname, gethostname

def main():
  # The URL of the server is the 1st command line argument
  url = sys.argv[1]
  address = gethostbyname(gethostname())
  # Proxy for the XML-RPC server
```

```
  server = xmlrpclib.Server(url)
  loop = True
  while loop:
      # Get a new estimate of pi
      pi = compute_pi()
      # If it is not possible to set the result on the
      # server, the server is down so stop the calculation
      try:
        cont = server.set_result(pi,address)
        loop = (cont == 1)
      except:
        loop = False


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

Example fo execution using a single client:

```
$ time ./pi_server.py
Server URL is http://130.120.229.82:8000
[ 15:43:26   130.120.229.82 ] : 3.141130  +/-  0.000000  (   1)
[ 15:43:29   130.120.229.82 ] : 3.141475  +/-  0.000345  (   2)
[ 15:43:33   130.120.229.82 ] : 3.141237  +/-  0.000310  (   3)
[ 15:43:37   130.120.229.82 ] : 3.141429  +/-  0.000292  (   4)
[ 15:43:40   130.120.229.82 ] : 3.141494  +/-  0.000235  (   5)
[ 15:43:44   130.120.229.82 ] : 3.141573  +/-  0.000207  (   6)
[ 15:43:48   130.120.229.82 ] : 3.141626  +/-  0.000183  (   7)
[ 15:43:51   130.120.229.82 ] : 3.141663  +/-  0.000163  (   8)
```

Average is 3.5 seconds/block

Example fo execution using a multiple clients:

```
$ time ./pi_server.py
Server URL is http://130.120.229.82:8000
[ 15:39:56         127.0.0.1 ] : 3.141700  +/-  0.000000  (   1)
[ 15:39:56         127.0.0.1 ] : 3.141630  +/-  0.000070  (   2)
[ 15:39:57         127.0.0.1 ] : 3.141590  +/-  0.000057  (   3)
```

```
[ 15:39:58         127.0.0.1 ] : 3.141404  +/-  0.000191  (   4)
[ 15:39:58   130.120.229.23 ] : 3.141325  +/-  0.000167  (   5)
[ 15:39:58   130.120.229.23 ] : 3.141306  +/-  0.000138  (   6)
[ 15:39:59         127.0.0.1 ] : 3.141336  +/-  0.000120  (   7)
[ 15:40:00         127.0.0.1 ] : 3.141444  +/-  0.000150  (   8)
[...]
[ 15:40:58   130.120.229.82 ] : 3.141526  +/-  0.000041  ( 177)
[ 15:40:58   130.120.229.82 ] : 3.141522  +/-  0.000041  ( 178)
[ 15:40:59         127.0.0.1 ] : 3.141524  +/-  0.000041  ( 179)
[ 15:40:59   130.120.229.23 ] : 3.141524  +/-  0.000041  ( 180)
[ 15:40:59         127.0.0.1 ] : 3.141523  +/-  0.000041  ( 181)
[ 15:41:00         127.0.0.1 ] : 3.141521  +/-  0.000040  ( 182)
[ 15:41:00   130.120.229.29 ] : 3.141518  +/-  0.000040  ( 183)
[ 15:41:00   130.120.229.27 ] : 3.141520  +/-  0.000040  ( 184)
[ 15:41:00         127.0.0.1 ] : 3.141517  +/-  0.000040  ( 185)
real    1m9.958s
user    0m0.168s
sys     0m0.028s
```

Average is 0.37 seconds/block

Problem 3 : Numerical computation of a 2-electron integral

We want to compute numerically the value of the following integral:

$$\langle \phi_1 \phi_2 | \phi_3 \phi_4 \rangle = \iint \phi_1(r_1)\phi_2(r_2)\frac{1}{r_{12}}\phi_3(r_1)\phi_4(r_2)dr_1\,dr_2$$

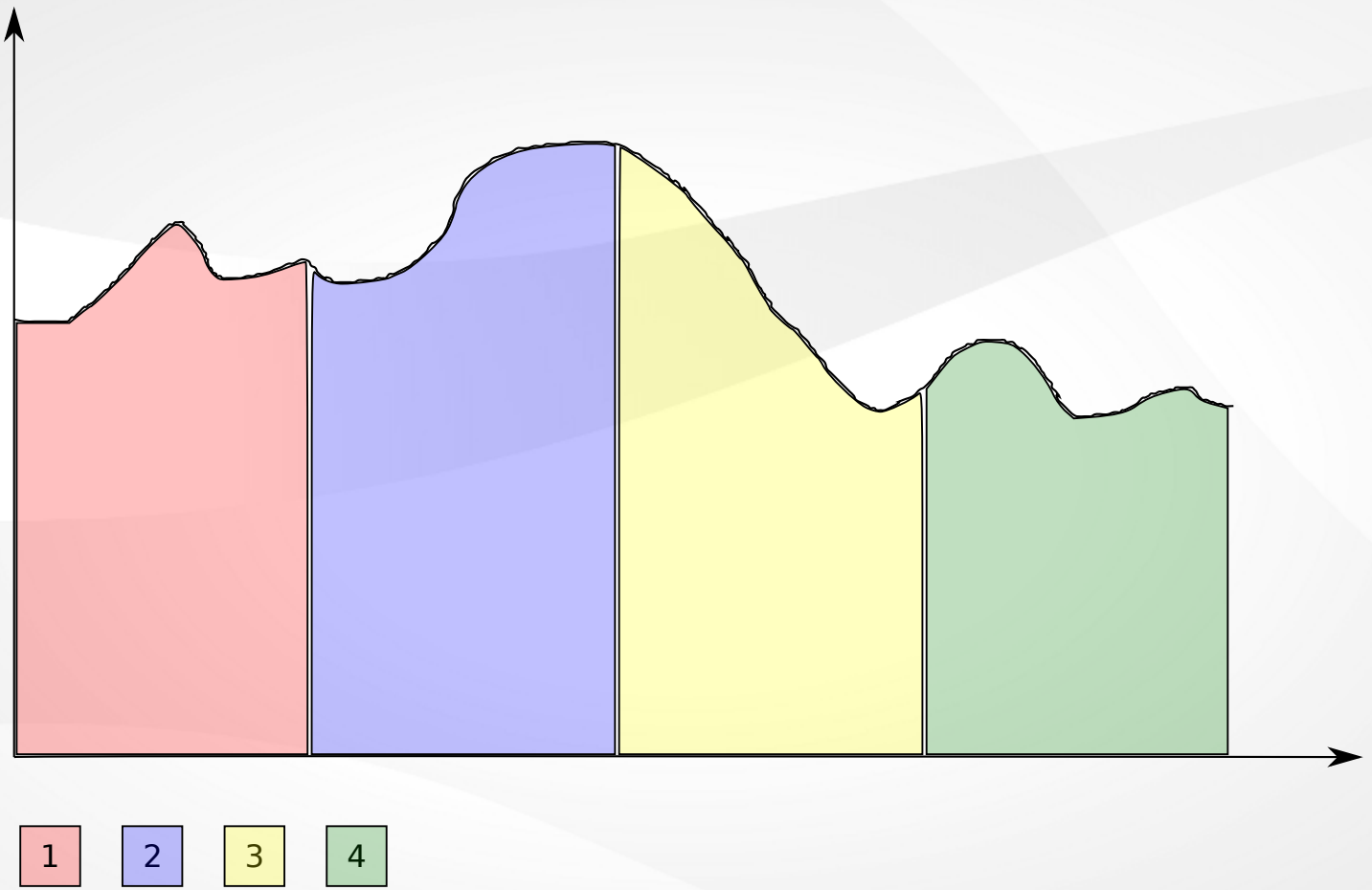
Constraints:

- We need to use Fortran
- A large number of points will be computed ($\sim 10^{10}$)

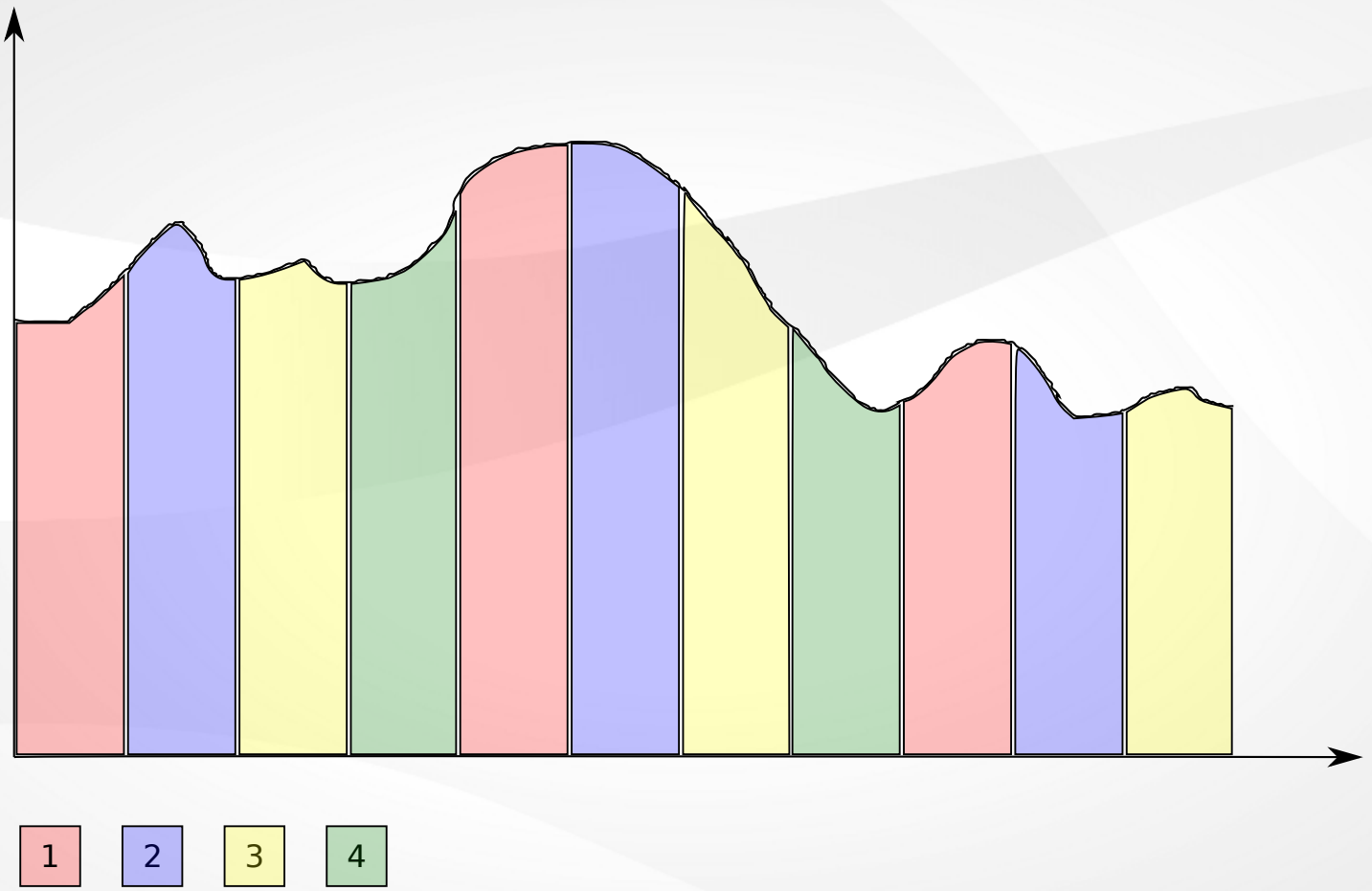
Simple solution:

- Compute the sum over a fixed number of grid points per CPU
- Use the Message Passing Interface (MPI) to communicate

Simple partition:



Better load balancing:



Message Passing Interface

MPI is a standard Application Programming Interface (API) which specifies how processes can communicate together.

- Each process has a *rank* and belongs to a *group* of processes.
- Processes can do *point-to-point* or *collective* communications

There is no need to pass the IP address and port number. All low-level communication is handled.

MPI programs start with a call to the *MPI_Init* function

```fortran
! Fortran
integer :: ierr
call MPI_Init(ierr)
```

```c
// C
#include <mpi.h>
int MPI_Init(int *argc, char ***argv)
```

```
// C++
#include <mpi.h>
void MPI::Init(int& argc, char**& argv)
void MPI::Init()
```

MPI programs end with a call to the *MPI_Finalize* function

```
integer :: ierr
call MPI_Finalize(ierr)
```

The rank of the current process is obtained with

```
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
```

and the total number of processes is obtained with

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
```

Synchronization

```
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

All the processes are blocked until they are *all* at this point. They wait for each other.

Point-to-point send/receive operation

```
include 'mpif.h'
<type>   :: BUF(*)
integer  :: n, datatype, tag, comm, ierr
integer  :: status(MPI_STATUS_SIZE)
integer  :: sender, receiver

if (my_id == sender) then
  call MPI_SEND(buffer, n, datatype, receiver, tag, comm, ierr)
else if (my_id == receiver) then
  call MPI_RECV(buffer, n, datatype, sender, tag, comm, status, ierr)
endif
```

- *sender* : Rank of the process sending the data
- *receiver* : Rank of the process receiving the data
- <*type*> : Type of data (*double precision*, *integer*, etc)
- *buffer* : array of type <*type*>
- *n* : number of elements to send
- *datatype* : MPI type of data (*MPI_DOUBLE_PRECISION*, *MPI_INTEGER4*, etc)
- *tag* : Message tag. Used to identify the message.
- *comm* : Communicator. Usually *MPI_COMM_WORLD*
- *ierr* : if *ierr == MPI_SUCCESS*, everything went fine
- *status* : Contains some information about the incoming message to track failures

Collective communications

Broadcast : one-to-all communication. Send the same data to all processes.

```
include 'mpif.h'
<type>  :: buffer(*)
integer :: n, datatype, sender, comm, ierr
call MPI_BCAST(buffer, n, datatype, sender, comm, ierr)
```

- *buffer* : Data to send to all processes
- *n* : Number of elements in *buffer*

Reductions: all-to-one communication.

```
include 'mpif.h'
<type>  ::  sendbuf(*), recvbuf(*)
integer ::  n, datatype, op, sender, comm, ierr
call MPI_REDUCE(sendbuf, recvbuf, n, datatype, op, sender, comm, ierr)
```

- *sendbuf* : Buffer of data to send
- *recvbuf* : Buffer in which the data will be received
- *op* : Reduction operation to perform. Examples: *MPI_SUM*, *MPI_MAX*, *MPI_PROD*, etc

The all-to-all variant is *MPI_ALLREDUCE*.

MPI has lots of routines, have a look a the documentation.

Two-electron integral using MPI

Pseudo-code

```
function f(r1,r2) {
   ...
}

MPI_Init()
myid := MPI_COMM_RANK( MPI_COMM_WORLD )
nproc := MPI_COMM_SIZE( MPI_COMM_WORLD )

dx := (xmax-xmin)/(nmax-1)
dv := dx^6

local_result := 0.
// For 4 processors,
// Processor 0 runs over 1,5,9 ,13,...
// Processor 1 runs over 2,6,10,14,...
```

```
// Processor 2 runs over 3,7,11,15,...
// Processor 3 runs over 4,8,12,16,...
for i = myid+1 to nmax with a step of nproc {
  for j,k,l,m,n = 1 to nmax {
     r1(1) := (i-1) * dx + xmin
     r1(2) := (j-1) * dx + xmin
     r1(3) := (k-1) * dx + xmin
     r2(1) := (l-1) * dx + xmin + dx/2
     r2(2) := (m-1) * dx + xmin + dx/2
     r2(3) := (n-1) * dx + xmin + dx/2
     // (+ dx/2 : Avoids divergence in 1/r12)

     local_result := local_result + f(r1,r2) * dv
  }
}

result := MPI_REDUCE(local_result, MPI_SUM, MPI_COMM_WORLD)
```

```
if (myid = 0) {
   print result
}

MPI_Finalize()
```

Fortran implementation

```
double precision function f(r1,r2)
  implicit none
  double precision, intent(in) :: r1(3), r2(3)

  ! < Phi_1 (r1)  Phi_2 (r1)  1/r12  Phi_3 (r2)  Phi_4 (r2) >

  double precision :: Phi_1, Phi_2, Phi_3, Phi_4
  double precision :: r12_inv

  double precision,parameter :: alpha_1=1.d0 , alpha_3=1.5d0
  double precision,parameter :: alpha_2=4.2d0, alpha_4=2.3d0
```

```
double precision,parameter :: X_1(3)=(/ 0.d0, 0.d0, 0.d0 /)
double precision,parameter :: X_2(3)=(/ 0.d0, 1.d0, 0.d0 /)
double precision,parameter :: X_3(3)=(/ 0.d0, 1.d0, 1.d0 /)
double precision,parameter :: X_4(3)=(/ 1.d0, 1.d0, 0.d0 /)

Phi_1 = exp (-alpha_1*((r1(1)-X_1(1))*(r1(1)-X_1(1)) + &
                       (r1(2)-X_1(2))*(r1(2)-X_1(2)) + &
                       (r1(3)-X_1(3))*(r1(3)-X_1(3))) )

Phi_2 = exp (-alpha_2*((r2(1)-X_2(1))*(r2(1)-X_2(1)) + &
                       (r2(2)-X_2(2))*(r2(2)-X_2(2)) + &
                       (r2(3)-X_2(3))*(r2(3)-X_2(3))) )

Phi_3 = exp (-alpha_3*((r1(1)-X_3(1))*(r1(1)-X_3(1)) + &
                       (r1(2)-X_3(2))*(r1(2)-X_3(2)) + &
                       (r1(3)-X_3(3))*(r1(3)-X_3(3))) )
```

```
  Phi_4 = exp (-alpha_4*((r2(1)-X_4(1))*(r2(1)-X_4(1)) + &
                         (r2(2)-X_4(2))*(r2(2)-X_4(2)) + &
                         (r2(3)-X_4(3))*(r2(3)-X_4(3))) )

  r12_inv = 1.d0/dsqrt ( (r1(1)-r2(1))*(r1(1)-r2(1)) + &
                         (r1(2)-r2(2))*(r1(2)-r2(2)) + &
                         (r1(3)-r2(3))*(r1(3)-r2(3)) )

  f = Phi_1 * Phi_2 * r12_inv * Phi_3 * Phi_4
end

program bielec

  implicit none
  include 'mpif.h'

  integer :: ierr
  integer :: myid
```

```
integer :: nproc

integer :: i,j,k,l,m,n
integer, parameter :: nmax=30
double precision, parameter :: xmin = -2.d0, xmax = 2.d0

double precision, external :: f
double precision :: r1(3), r2(3)
double precision :: local_result, result
double precision :: dx,dv

! Initialize the MPI library
call MPI_Init(ierr)

! Get the rank of the current process
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

! Get the the total number of processes
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)

! Compute a partial result locally
local_result = 0.d0
dx = (xmax-xmin)/dble(nmax-1)
dv = dx**6

! For 4 processes,
! Proces 0 runs over 1,5,9 ,13,...
! Proces 1 runs over 2,6,10,14,...
! Proces 2 runs over 3,7,11,15,...
! Proces 3 runs over 4,8,12,16,...
do i=myid+1,nmax,nproc
 r1(1) = dble(i-1) * dx + xmin
 do j=1,nmax
  r1(2) = dble(j-1) * dx + xmin
  do k=1,nmax
   r1(3) = dble(k-1) * dx + xmin
```

```
    do l=1,nmax
     r2(1) = dble(l-1) * dx + xmin + dx/2
     ! + dx/2 : Avoids divergence in r1=r2
     do m=1,nmax
      r2(2) = dble(m-1) * dx + xmin + dx/2
      do n=1,nmax
       r2(3) = dble(n-1) * dx + xmin + dx/2
       local_result = local_result + f(r1,r2) * dv
      enddo
     enddo
    enddo
   enddo
  enddo

! Sum the local results of all processes
! into the master process
call MPI_REDUCE(local_result, result, 1, &
```

```
    MPI_DOUBLE_PRECISION, MPI_SUM, &
    0, MPI_COMM_WORLD, ierr)

  if (myid == 0) then
    print *,  result
  endif

  ! Terminate the MPI library
  call MPI_Finalize(ierr)

end
```

Links

- Open MPI : Open source MPI implementation : http://www.open-mpi.org/
- Open MPI documentation : http://www.open-mpi.org/doc/v1.8/

Coarray Fortran (CAF)

Extension of the Fortran 2008 standard.

- Each running process is called an *image*.
- The number of images is obtained with the built-in *num_image()* function
- The rank of the current process is obtained with *this_image()*

A *codimension* can be given to arrays in square brackets, for example:

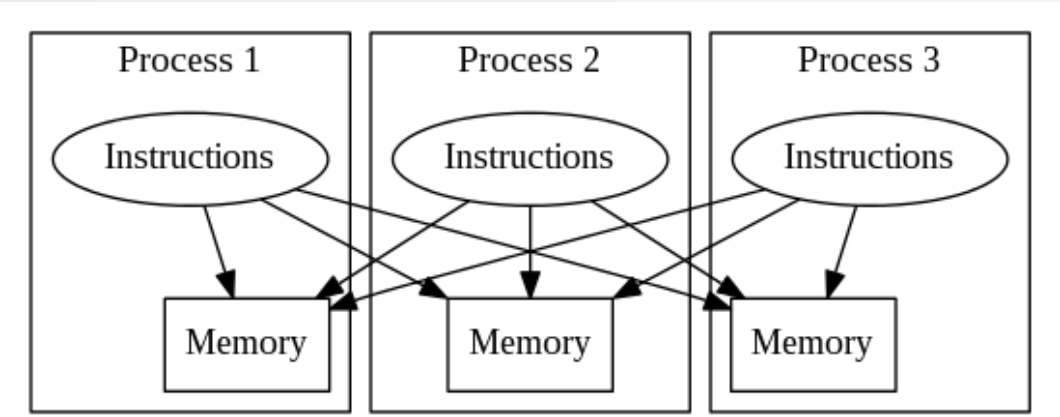
```
integer          :: i[*]
double precision :: A(10)[*]
```

For any image,

- *i[2]* : value of *i* in image number 2
- *A(5)[4]* : value of *A(5)* in image number 4

Any image can directly have access an element in the memory of another image.

PGAS : Partitioned Global Address Space.



Much simpler than MPI:

- Higher level of abstraction than MPI
- Types, message sizes, tags, etc are known by the compiler
- Compiler can place the communication instructions where it is the best (asynchronous comm)
- Better performance obtained by non-experts

But:

- Experts can get more performance with MPI : more flexibility (lower level)
- Having knowledge of how MPI works helps to write efficient (CAF) code

Calculation of the 2-electron integral

```
double precision function f(r1,r2)
  implicit none
  double precision, intent(in) :: r1(3), r2(3)

  ! < Phi_1 (r1)  Phi_2 (r1)  1/r12  Phi_3 (r2)  Phi_4 (r2) >

  double precision :: Phi_1, Phi_2, Phi_3, Phi_4
  double precision :: r12_inv

  double precision,parameter :: alpha_1=1.d0 , alpha_3=1.5d0
  double precision,parameter :: alpha_2=4.2d0, alpha_4=2.3d0

  double precision,parameter :: X_1(3)=(/ 0.d0, 0.d0, 0.d0 /)
  double precision,parameter :: X_2(3)=(/ 0.d0, 1.d0, 0.d0 /)
  double precision,parameter :: X_3(3)=(/ 0.d0, 1.d0, 1.d0 /)
  double precision,parameter :: X_4(3)=(/ 1.d0, 1.d0, 0.d0 /)
```

```
Phi_1 = exp (-alpha_1*((r1(1)-X_1(1))*(r1(1)-X_1(1)) + &
                       (r1(2)-X_1(2))*(r1(2)-X_1(2)) + &
                       (r1(3)-X_1(3))*(r1(3)-X_1(3))) )

Phi_2 = exp (-alpha_2*((r2(1)-X_2(1))*(r2(1)-X_2(1)) + &
                       (r2(2)-X_2(2))*(r2(2)-X_2(2)) + &
                       (r2(3)-X_2(3))*(r2(3)-X_2(3))) )

Phi_3 = exp (-alpha_3*((r1(1)-X_3(1))*(r1(1)-X_3(1)) + &
                       (r1(2)-X_3(2))*(r1(2)-X_3(2)) + &
                       (r1(3)-X_3(3))*(r1(3)-X_3(3))) )

Phi_4 = exp (-alpha_4*((r2(1)-X_4(1))*(r2(1)-X_4(1)) + &
                       (r2(2)-X_4(2))*(r2(2)-X_4(2)) + &
                       (r2(3)-X_4(3))*(r2(3)-X_4(3))) )

r12_inv = 1.d0/dsqrt ( (r1(1)-r2(1))*(r1(1)-r2(1)) + &
```

```
                              (r1(2)-r2(2))*(r1(2)-r2(2)) + &
                              (r1(3)-r2(3))*(r1(3)-r2(3)) )

  f = Phi_1 * Phi_2 * r12_inv * Phi_3 * Phi_4
end

program bielec

  implicit none

  integer :: i,j,k,l,m,n
  integer, parameter :: nmax=30
  double precision, parameter :: xmin = -2.d0, xmax = 2.d0

  double precision, external :: f
  double precision :: r1(3), r2(3)
  double precision :: local_result[*], result
  double precision :: dx,dv
```

```
! Compute a partial result locally
local_result = 0.d0
dx = (xmax-xmin)/dble(nmax-1)
dv = dx**6

! Image 0 runs over 1,5,9 ,13,...
! Image 1 runs over 2,6,10,14,...
! Image 2 runs over 3,7,11,15,...
! Image 3 runs over 4,8,12,16,...
do i=this_image()+1,nmax,num_images()
 r1(1) = dble(i-1) * dx + xmin
 do j=1,nmax
  r1(2) = dble(j-1) * dx + xmin
  do k=1,nmax
   r1(3) = dble(k-1) * dx + xmin
   do l=1,nmax
    r2(1) = dble(l-1) * dx + xmin + dx/2
```

```
      ! + dx/2 : Avoids divergence in r1=r2
      do m=1,nmax
       r2(2) = dble(m-1) * dx + xmin + dx/2
       do n=1,nmax
        r2(3) = dble(n-1) * dx + xmin + dx/2
        local_result = local_result + f(r1,r2) * dv
       enddo
      enddo
     enddo
    enddo
   enddo
  enddo

  ! Sum the local results of all processes
  do i=1,num_images()
    result = result + local_result[i]
  enddo
```

```
  if (this_image() == 1) then
    print *,  result
  endif

end
```

Links

- Coarray Fortran http://www.co-array.org/
- Rice University http://caf.rice.edu/
- Coarray with gfortran http://gcc.gnu.org/wiki/Coarray

Problem 4: Parallelization of a matrix product

Matrix products are usually not written by the user. It is preferable to use optimized libraries to perform linear algebra. A standardized API exists (**Lapack**) on top of the **BLAS** API. Every CPU manufacturer provides optimized libraries (MKL, ATLAS, NAG, ACML, CULA, etc).

For matrix products, we use **DGEMM**:

- D : double precision
- Ge : General
- MM : Matrix Multiplication

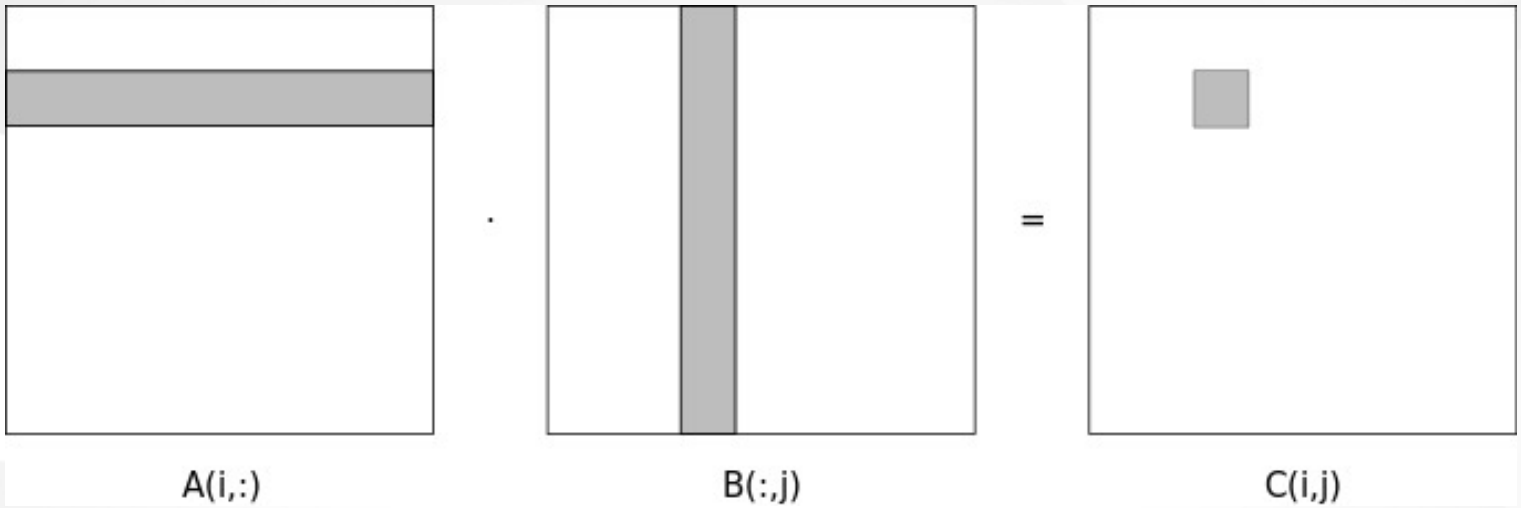
```
NAME
      DGEMM - perform one of the matrix-matrix operations
      C := alpha*op( A )*op( B ) + beta*C
```

```
SYNOPSIS
     SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
                        B, LDB, BETA, C, LDC )

          CHARACTER*1      TRANSA, TRANSB
          INTEGER          M, N, K, LDA, LDB, LDC
          DOUBLE PRECISION ALPHA, BETA
          DOUBLE PRECISION A( LDA, * ), B( LDB, * ), C( LDC,* )
...
```

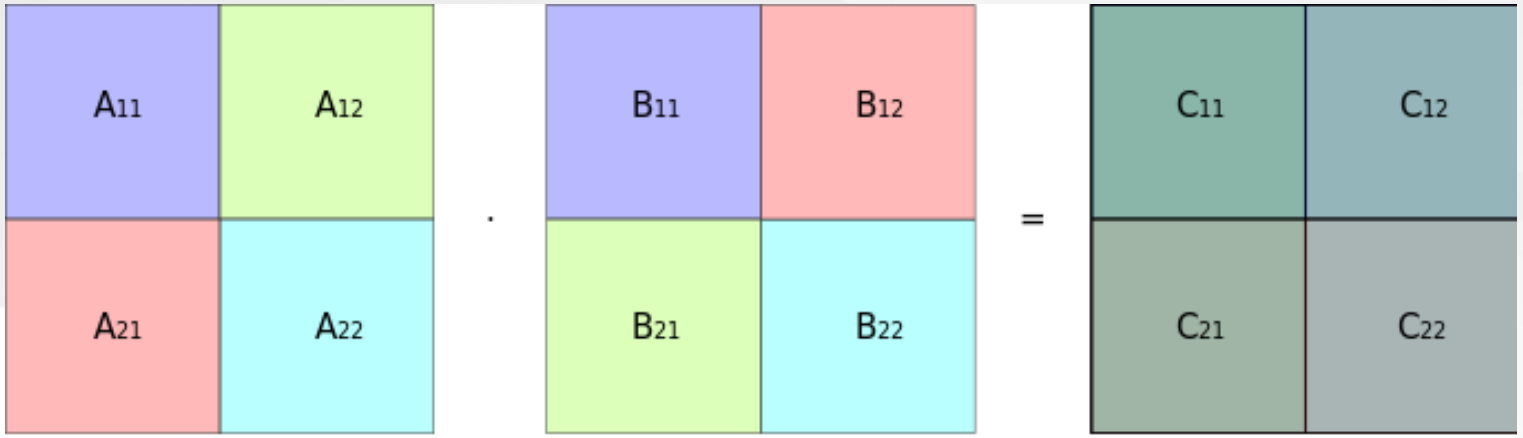
$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

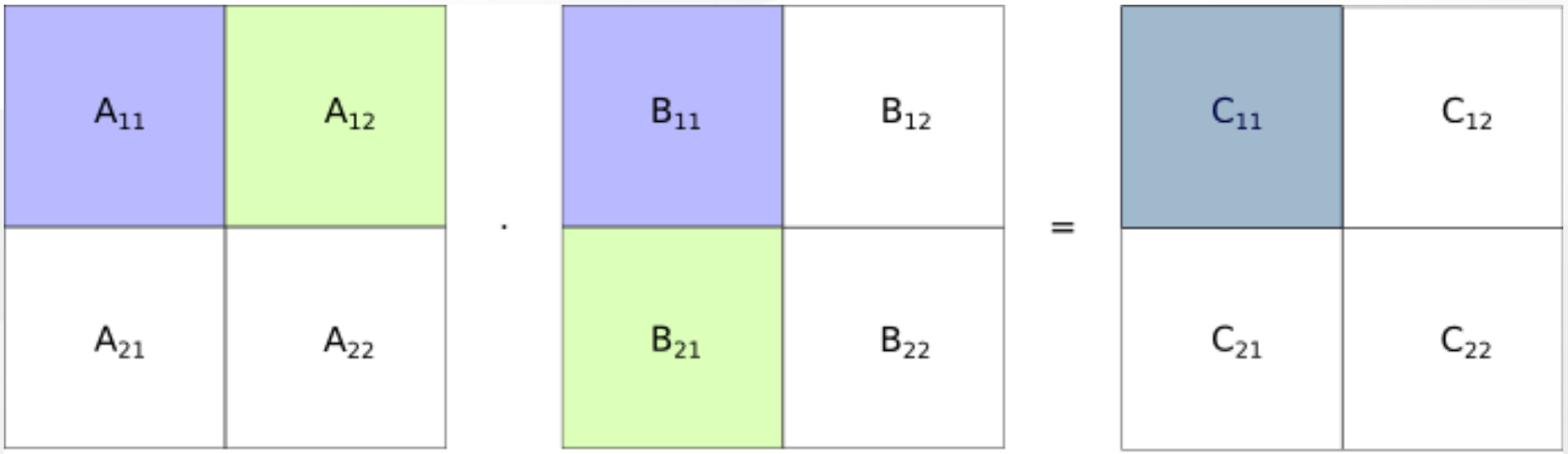
```
C = 0.
do j=1,N
  do i=1,N
    do k=1,N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end do
```

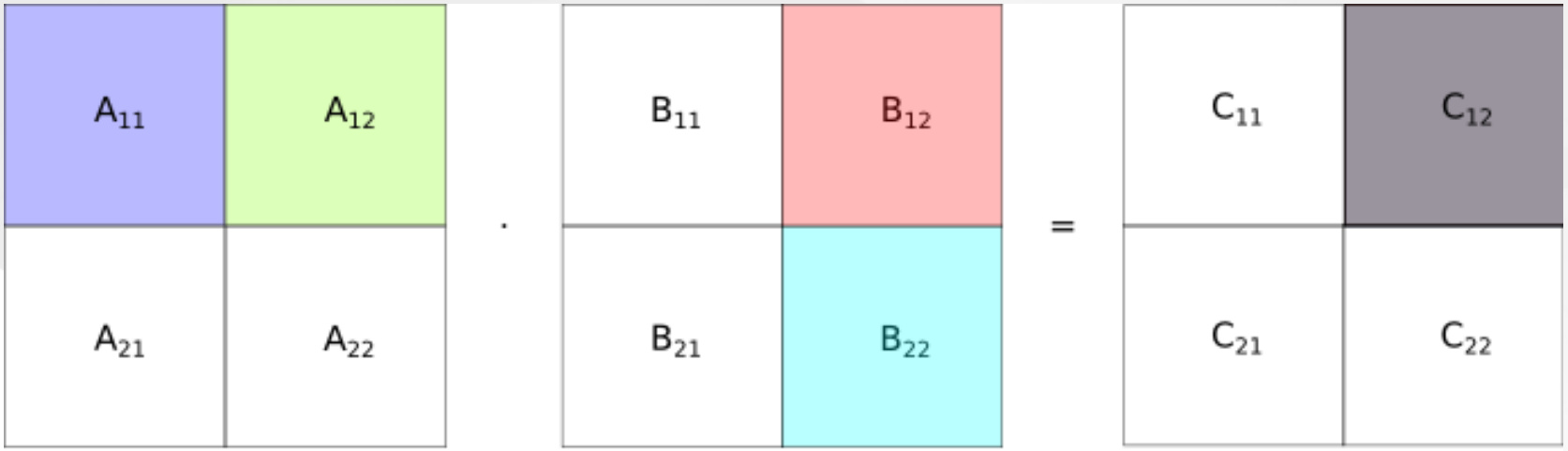
```
   end do
end do
```



The final matrix can be split, such that each CPU core builds part of it.


A11
A12
A21
A22
.
B11
B12
B21
B22
=
C11
C12
C21
C22


A11
A12
A21
A22
.
B11
B12
B21
B22
=
C11
C12
C21
C22



$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

The large *N* x *N* matrix product can be performed by doing 8 smaller matrix products of size *N*/2 x *N*/2, that can be done simultaneously by 8 CPUs.

Data access is *slow* with respect to calculation:

Operation	Latency (ns)
Int ADD	0.3
FP ADD	0.9
FP MUL	1.5
L1 cache	1.2
L2 cache	3.5
L3 cache	13
RAM	79
Infiniband	1 200
Ethernet	50 000
Disk (SSD)	50 000
Disk (15k)	2 000 000

Arithmetic intensity : Flops/memory access

Sequential algorithm:

- The most efficient operation on a computer : ~95% of the peak performance
- $\mathcal{O}(N^2)$ data access and $\mathcal{O}(N^3)$ flops -> High arithmetic intensity -> Compute bound.
- (2 x N^2) data reads, (N) data writes and (N^3) flops
- Arithmetic intensity = N/2

4-way parallel algorithm:

- Here, the data can not be disjoint between the CPUs
- To build one block, 4 blocks are needed
- The same block will be read by different CPUs
- (2 x N x N/2) data reads, (N/2 x N/2 x N) flops
- Arithmetic intensity = N/4 : less than sequential algorithm

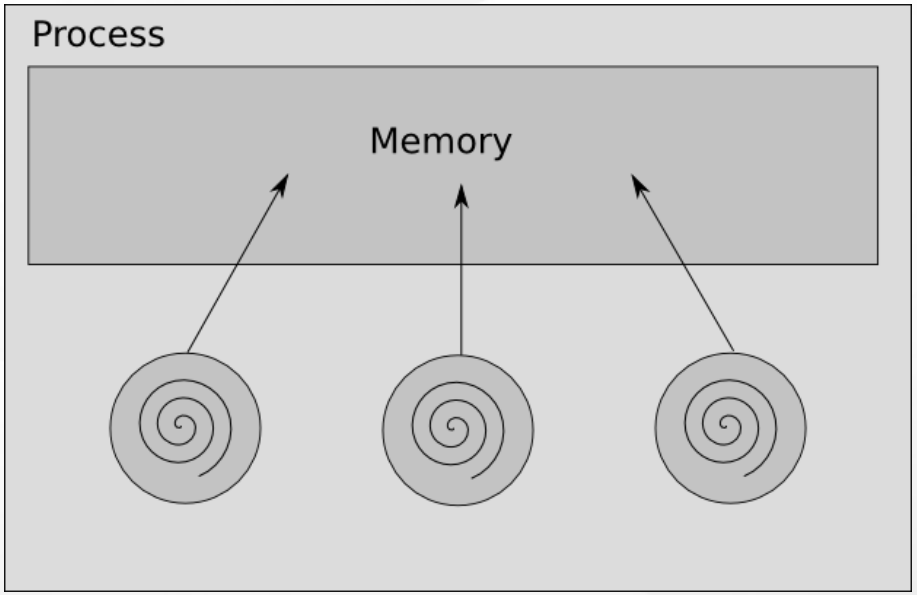
Difficulty:

- A modern CPU can perform 8 FP ADD and 8 FP MUL per cycle (!!!)
- A *random* memory access takes ~300 cycles (4 800 flops!)

- A network access takes ~4000 cycles (64 000 flops!)
- To benefit from *distributed* parallelism, the matrices have to be *very* large

Proposed solution: Use **shared-memory** parallelism

- Avoids network bottleneck (~10x slower than RAM)
- L3 cache sharing optimizes data access (~6x faster than RAM)
- Hardware memory prefetchers will mask the RAM latencies



Threads

pthreads

- When starting a new thread, a concurrent execution of a function is started in the same memory domain.
- A private memory domain is created for the thread
- The parent process can wait until all the children threads have finished their work
- Fork/join model

Example in pseudo-code

```
function f() { ... }

t = pthread_create(f);
```

Example in Python

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

import threading
import time

A = 0

def f(x):
  global A
  time.sleep(1.)
  A = x
  print x, "written by thread"

def main():
  t = threading.Thread(target=f, args = [2] )
  print "Before thread starts, A= ", A
  t.start()
  time.sleep(0.5)
```

```
  print "A= ", A
  time.sleep(1.)
  print "A= ", A
  time.sleep(1.)
  t.join()
  print "After join, A=", A

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

What happens when 2 threads read from the same memory address at the same time?

Nothing special

What happens when 2 threads write at the same memory address at the same time?

If you are lucky, the program crashes. Otherwise, it is unpredictible.

Locks

To avoid writing simultaneously at the same memory location, we introduce **Locks**:

acquire_lock(L)

if L is free, the current thread gets the lock. Otherwise, block until the lock can be acquired

release_lock(L)

the lock is released by the current thread

Example of wrong code

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

A = 0

def f(x):
```

```
  global A
  for i in range(x):
    A = A+1

def main():
  t = [None for i in range(10)]
  for i in range(10):
    t[i] = threading.Thread(target=f, args = [100000] )
  for i in range(10):
    t[i].start()

  for i in range(10):
    t[i].join()
  print A

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

Using a lock:

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

A = 0
lock = threading.Lock()

def f(x):
  global A
  a = 0
  for i in range(x):
    a = a+1
  lock.acquire()
  A = A+a
  lock.release()

def main():
  t = [None for i in range(10)]
```

```
  for i in range(10):
    t[i] = threading.Thread(target=f, args = [100000] )
  for i in range(10):
    t[i].start()

  for i in range(10):
    t[i].join()
  print A

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

A **semaphore** is more general than a lock : it can be taken simultaneously by more than 1 thread.

OpenMP

OpenMP is an extension of programming languages that enable the use of multi-threading to parallelize the code using directives given as comments. The *same* source code can be compiled with/without OpenMP.

For example:

```
!$OMP PARALLEL  DEFAULT(SHARED)  PRIVATE(i)
!$OMP DO
do i=1,n
  A(i) = B(i) + C(i)
end do
!$OMP END DO
!$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

Other important directives:

- !$OMP CRITICAL ... !$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
- omp_get_thread_num() : returns the ID of the current running thread
- omp_get_num_threads() : returns the total number of running threads
- OMP_NUM_THREADS : Environment variable (shell) that fixes the number of threads to run

Matrix product : simple OpenMP example

Loop parallelism

```
A = create_matrix()
B = create_matrix()

// parallelize loop over i and j
for i=1 to N using a step of N/2 {
  for j=1 to N using a step of N/2 {
    for k=1 to N using a step of N/2 {
      // C_ij = A_ik.B_kj
      DGEMM ( C(i,j), A(i,k), B(k,j), (N/2, N/2) )
    }
  }
}
```

```
program submatrix_openmp
  implicit none
  integer, parameter                  :: sze = 5000
  double precision, allocatable, dimension (:,:) :: A, B, C
  double precision                    :: cpu_0, cpu_1

  integer                             :: istart(2), iend(2)
  integer                             :: jstart(2), jend(2)
  integer                             :: i,j

  integer                             :: i1,i2,j1,j2,step
  integer, external                   :: omp_get_thread_num
  double precision                    :: s

  allocate (A(sze,sze), B(sze,sze), C(sze,sze))

  C = 0.d0
  step = sze/2
```

```
!$OMP PARALLEL DEFAULT(NONE)                                    &
    !$OMP PRIVATE(i1,j1,j2,istart,jstart,iend,jend,  &
    !$OMP   cpu_0,cpu_1)                                        &
    !$OMP SHARED(A,B,C,step)

!$OMP MASTER
call wall_time(cpu_0)
!$OMP END MASTER

!Build the submatrices

!$OMP DO COLLAPSE(2)
do i1=1,sze,step
  do j2=1,sze,step
    istart(1) = i1
    iend(1) = istart(1)+step-1
    jstart(1) = j2
```

```
      jend(1) = jstart(1)+step-1
      call create_matrix(A,sze,7.d0,istart(1),        &
                         iend(1),jstart(1),jend(1))
      call create_matrix(B,sze,11.d0,istart(1),       &
                         iend(1),jstart(1),jend(1))
    enddo
  enddo
  !$OMP END DO

  !$OMP MASTER
  call wall_time(cpu_1)
  write(0,*)  'Matrix build time : ', cpu_1-cpu_0, 's'
  call wall_time(cpu_0)
  !$OMP END MASTER

  !$OMP DO COLLAPSE(2)
  do i1=1,sze,step
    do j2=1,sze,step
```

```
      istart(1) = i1
      jstart(2) = j2
      iend(1) = istart(1)+step-1
      jend(2) = jstart(2)+step-1
      do j1=1,sze,step
        jstart(1) = j1
        istart(2) = j1
        jend(1) = jstart(1)+step-1
        iend(2) = istart(2)+step-1

        ! Compute the submatrix product
        call dgemm('N','N',                          &
            1+iend(1)-istart(1),                     &
            1+jend(1)-jstart(1),                     &
            1+jend(2)-jstart(2),                     &
            1.d0, A(istart(1),jstart(1)),sze,        &
            B(istart(2),jstart(2)),sze,              &
            1.d0, C(istart(1),jstart(2)),sze )
```

```
      enddo
    enddo
  enddo
  !$OMP END DO
  !$OMP MASTER
  call wall_time(cpu_1)
  write(0,*)  'Compute Time : ', cpu_1-cpu_0, 's'
  !$OMP END MASTER

  !$OMP END PARALLEL

  ! Print the sum of the elements
  s = 0.d0
  do j=1,sze
    do i=1,sze
      s = s+C(i,j)
    enddo
  enddo
```

```
  deallocate (A,B,C)
  print *,  s
end
```

Task parallelism

Shared-memory work stealing

```
A = create_matrix()
B = create_matrix()

queue= []

for i=1 to N using a step of N/2 {
  for j=1 to N using a step of N/2 {
    for k=1 to N using a step of N/2 {
      // C_ij = A_ik.B_kj
      queue = queue + [ ( i, j, k ) ]
    }
  }
}

sem = semaphore(nproc)
```

```
function do_work( i,j,k ) {
  DGEMM (A,B,C,i,j,k)
  release_semaphore(sem)
}

do while queue is not empty
{
  acquire_semaphore(sem)
  // Pop out the 1st element of the queue
  params = queue.pop()
  pthread_create( do_work, params )
}
```

```
program submatrix_openmp
  implicit none
  integer, parameter                :: sze = 5000
  double precision, allocatable, dimension (:,:) :: A, B, C
```

```
double precision                    :: wall_0, wall_1

integer                             :: istart(2), iend(2)
integer                             :: jstart(2), jend(2)
integer                             :: i,j

integer                             :: i1,i2,j1,j2,step

double precision                    :: s

allocate (A(sze,sze), B(sze,sze), C(sze,sze))

C = 0.d0
step = sze/2

!$OMP PARALLEL DEFAULT(NONE)                                  &
    !$OMP PRIVATE(i1,j1,j2,istart,jstart,iend,jend)  &
    !$OMP SHARED(A,B,C,step,wall_0,wall_1)
```

```
!$OMP MASTER
call wall_time(wall_0)
!Build the submatrices
do i1=1,sze,step
  do j2=1,sze,step
    istart(1) = i1
    iend(1) = istart(1)+step-1
    jstart(1) = j2
    jend(1) = jstart(1)+step-1
    !$OMP TASK
    call create_matrix(A,sze,7.d0,istart(1),          &
                       iend(1),jstart(1),jend(1))
    !$OMP END TASK
    !$OMP TASK
    call create_matrix(B,sze,11.d0,istart(1),         &
                       iend(1),jstart(1),jend(1))
    !$OMP END TASK
```

```
  enddo
enddo
!$OMP END MASTER

!$OMP TASKWAIT

!$OMP MASTER
call wall_time(wall_1)
write(0,*)  'Matrix build time : ', wall_1-wall_0, 's'
call wall_time(wall_0)
do i1=1,sze,step
  do j2=1,sze,step
    istart(1) = i1
    jstart(2) = j2
    iend(1) = istart(1)+step-1
    jend(2) = jstart(2)+step-1
    do j1=1,sze,step
      jstart(1) = j1
```

```
        istart(2) = j1
        jend(1) = jstart(1)+step-1
        iend(2) = istart(2)+step-1

        ! Compute the submatrix product
        !$OMP TASK
        call dgemm('N','N',                              &
            1+iend(1)-istart(1),                         &
            1+jend(1)-jstart(1),                         &
            1+jend(2)-jstart(2),                         &
            1.d0, A(istart(1),jstart(1)),sze,            &
            B(istart(2),jstart(2)),sze,                  &
            1.d0, C(istart(1),jstart(2)),sze )
        !$OMP END TASK
      enddo
    enddo
  enddo
  !$OMP END MASTER
```

```
  !$OMP TASKWAIT

  !$OMP END PARALLEL
  call wall_time(wall_1)
  write(0,*)  'Compute Time : ', wall_1-wall_0, 's'

  ! Print the sum of the elements
  s = 0.d0
  do j=1,sze
    do i=1,sze
      s = s+C(i,j)
    enddo
  enddo
  deallocate (A,B,C)
  print *,  s
end
```

Divide and Conquer algorithms

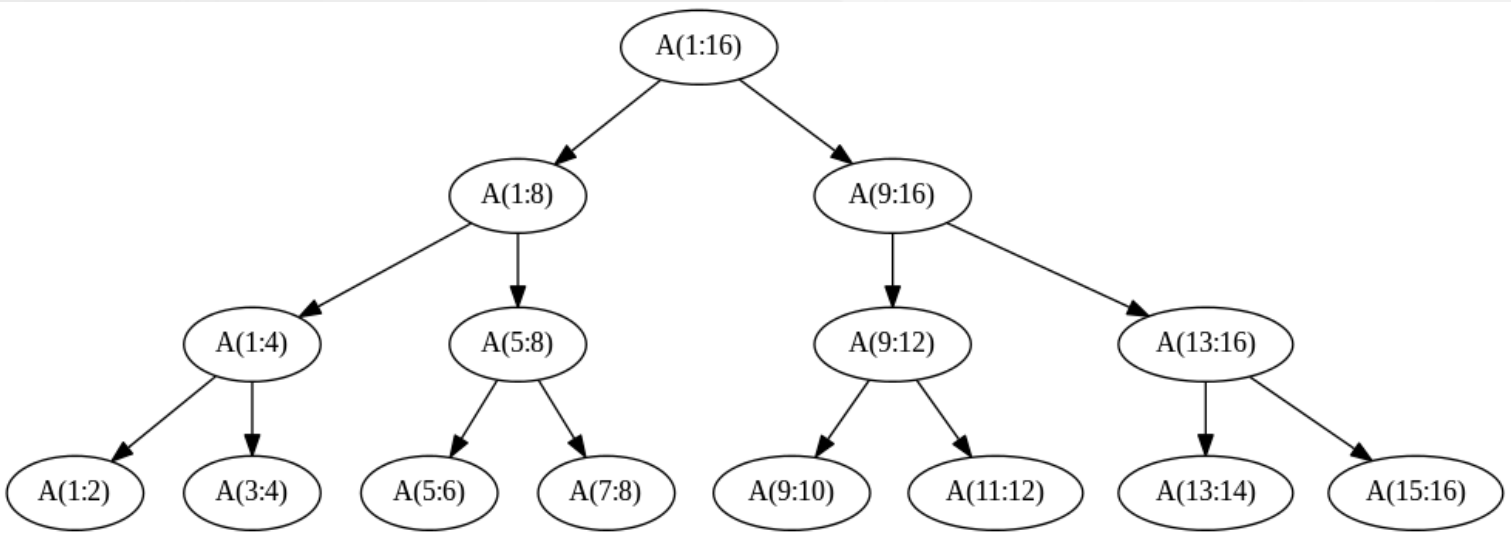
Algorithm based on recursion. The problem is divided in sub-problems that are solved in the same way as the large problem.

Example : Sum

Suppose you want to compute the sum of all the elements of the array A(1:16). This sum can be expressed as the sum of the two halves of the array :

S[A(1:16)] = S[A(1:8)] + S[A(9:16)]

The *S* function will be applied *recursively*.



Python

```
#!/usr/bin/python

sze_A = 5000000
A = [ i*1.5 for i in range(sze_A) ]

def sum_half(X):
  sze = len(X)
```

```
  if sze > 1 :
    return sum_half(X[:sze/2]) + sum_half(X[sze/2:])
  else:
    return X[0]

s = sum_half(A)
print 'DC    : ', s
print 'Exact :   1.875000375E+13'
```

Fortran OpenMP

```
program dc
  implicit none
  real, allocatable                 :: A(:)
  integer, parameter                :: sze = 5000000
  real                              :: s
  integer                           :: i

  allocate (A(sze))
```

```
! Initialize array
do i=1,sze
  A(i) = dble(i)*1.5
enddo

!$OMP PARALLEL DEFAULT(NONE) SHARED(A,s)

!$OMP SINGLE
call sum_half( A(1), sze, s)
!$OMP END SINGLE

!$OMP TASKWAIT
!$OMP END PARALLEL
print *,  'Loop  : ', sum(A)
print *,  'DC    : ', s
print *,  'Exact :   1.875000375E+13'
```

```
end

recursive subroutine sum_half(A,sze,s)
  implicit none

  integer, intent(in)                :: sze
  real, intent(in)                   :: A(sze)
  real, intent(out)                  :: s

  real                               :: sa, sb
  integer                            :: i, sze_new

  if ( sze > 1 ) then
    sze_new = sze/2

    !$OMP TASK SHARED(A,sa) FIRSTPRIVATE(sze_new)
    call sum_half(A(1), sze_new, sa)
    !$OMP END TASK
```

```
    !$OMP TASK SHARED(A,sb) FIRSTPRIVATE(sze_new,sze)
    call sum_half(A(sze_new+1), sze-sze_new, sb)
    !$OMP END TASK

    !$OMP TASKWAIT
    s = sa+sb

  else
    s = A(1)
  endif

end
```

Divide and Conquer matrix product

Pseudo-code

```
recursive subroutine divideAndConquer(A,B,C,sze,ie1,je2)

  if ( (ie1 < 200).and.(je2 < 200) ) then
    call DGEMM
  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

    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
```

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

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

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

    !$OMP END TASK
    !$OMP TASKWAIT

  endif

end

!$OMP PARALLEL DEFAULT(SHARED)
```

```
  !$OMP SINGLE
    call divideAndConquer(A,B,C,sze, sze, sze )
  !$OMP END SINGLE NOWAIT
  !$OMP TASKWAIT
!$OMP END PARALLEL
```

Fortran implementation

```
program submatrix_dc
  implicit none
  double precision, allocatable, dimension (:,:) :: A, B, C
  integer                                :: istart(2), iend(2)
  integer                                :: jstart(2), jend(2)
  integer, parameter                     :: sze = 5000
  double precision                       :: wall_0, wall_1
  double precision                       :: s
  integer :: i1,j1,i2,j2, i,j, step

  allocate (A(sze,sze), B(sze,sze), C(sze,sze))
```

```
call wall_time(wall_0)
C = 0.d0
step = sze/2

call wall_time(wall_0)

!$OMP PARALLEL DEFAULT(NONE)                                &
    !$OMP PRIVATE(i1,j1,j2,istart,jstart,iend,jend) &
    !$OMP SHARED(A,B,C,step)

!$OMP SINGLE
!Build the submatrices
do i1=1,sze,step
  do j2=1,sze,step
    istart(1) = i1
    iend(1) = istart(1)+step-1
    jstart(1) = j2
```

```
      jend(1) = jstart(1)+step-1
      !$OMP TASK SHARED(A)
      call create_matrix(A,sze,7.d0,istart(1), &
                         iend(1),jstart(1),jend(1))
      !$OMP END TASK
      !$OMP TASK SHARED(B)
      call create_matrix(B,sze,11.d0,istart(1), &
                         iend(1),jstart(1),jend(1))
      !$OMP END TASK
    enddo
  enddo
  !$OMP END SINGLE NOWAIT

  !$OMP TASKWAIT
  !$OMP END PARALLEL

  call wall_time(wall_1)
  write(0,*)  'Matrix build time : ', wall_1-wall_0, 's'
```

```
call wall_time(wall_0)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP SINGLE
call divideAndConquer(A,B,C,sze, sze, sze )
!$OMP END SINGLE NOWAIT
!$OMP TASKWAIT
!$OMP END PARALLEL

call wall_time(wall_1)
write(0,*)  'Compute Time : ', wall_1-wall_0, 's'

! Print the sum of the elements
s = 0.d0
do j=1,sze
  do i=1,sze
    s = s+C(i,j)
  enddo
enddo
```

```
  deallocate (A,B,C)
  print *,  s

end

recursive subroutine divideAndConquer(A,B,C,sze,ie1,je2)
  implicit none
  double precision                :: wall_0, wall_1

  integer, intent(in)             :: sze
  double precision, dimension (sze,sze), intent(in) :: A, B
  double precision, dimension (sze,sze), intent(out) :: C
  integer, intent(in)             :: ie1,je2

  if ( (ie1 < 200).and.(je2 < 200) ) then
    call dgemm('N','N',                          &
        ie1,                                     &
        je2,                                     &
```

```
        sze,                                    &
        1.d0, A,sze,                            &
        B,sze,                                  &
        1.d0, C,sze )
  else

    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
    call divideAndConquer(                      &
        A(1,1),                                 &
        B(1,1),                                 &
        C(1,1),                                 &
        sze,                                    &
        ie1/2,                                  &
        je2/2)

    !$OMP END TASK

    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
```

```
    call divideAndConquer(                    &
        A(1,1),                               &
        B(1,1+je2/2),                         &
        C(1,1+je2/2),                         &
        sze,                                  &
        ie1/2,                                &
        je2-(je2/2))
    !$OMP END TASK

    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
    call divideAndConquer(                    &
        A(1+ie1/2,1),                         &
        B(1,1),                               &
        C(1+ie1/2,1),                         &
        sze,                                  &
        ie1-(ie1/2),                          &
        je2/2)
    !$OMP END TASK
```

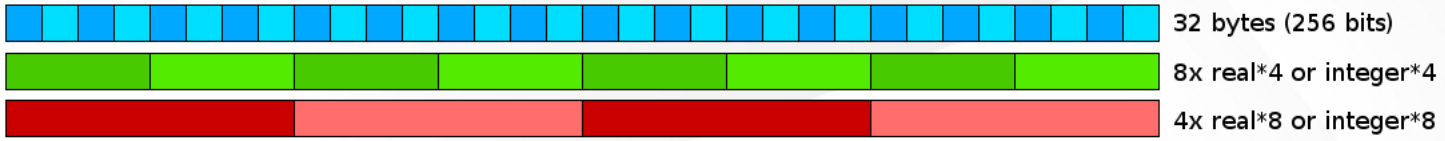
```
    !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
    call divideAndConquer(                    &
        A(1+ie1/2,1),                         &
        B(1,1+je2/2),                         &
        C(1+ie1/2,1+je2/2),                   &
        sze,                                  &
 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

    !$OMP END TASK
    !$OMP TASKWAIT

  endif

end
```

Vectorization

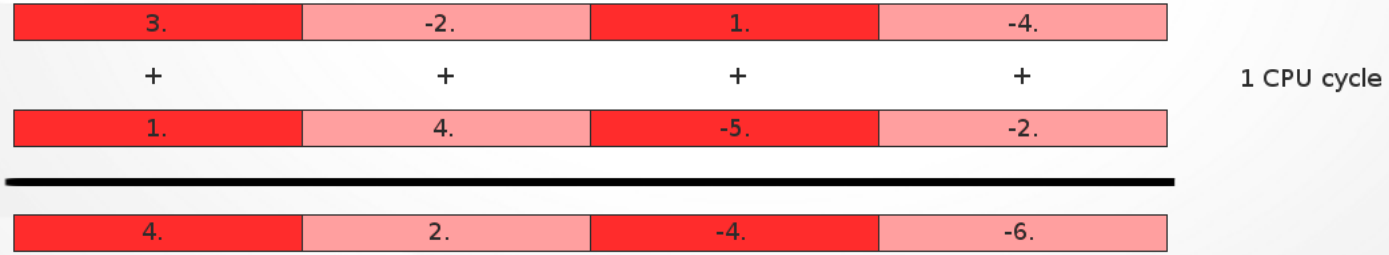
Parallelism that happens on a single CPU core.

SIMD : Single Instruction, Multiple Data

Execute the *same* instruction in parallel on all the elements of a vector:



Example : AVX vector ADD in double precision:



Different instruction sets exist in the x86 micro-architecture:

- MMX : Integer (64-bit wide)
- SSE -> SSE4.2 : Integer and Floating-point (128-bit)

- AVX : Integer and Floating-point (256-bit)
- AVX-512 : Integer and Floating-point (512-bit)

Requirements:

1. The elements of each SIMD vector must be contiguous in memory
2. The first element of each SIMD vector must be *aligned* on a proper boundary (64, 128, 256 or 512-bit).

Automatic vectorization

The compiler can generate automatically vector instructions when possible. A double precision AVX auto-vectorized loop generates 3 loops:

Peel loop (scalar)

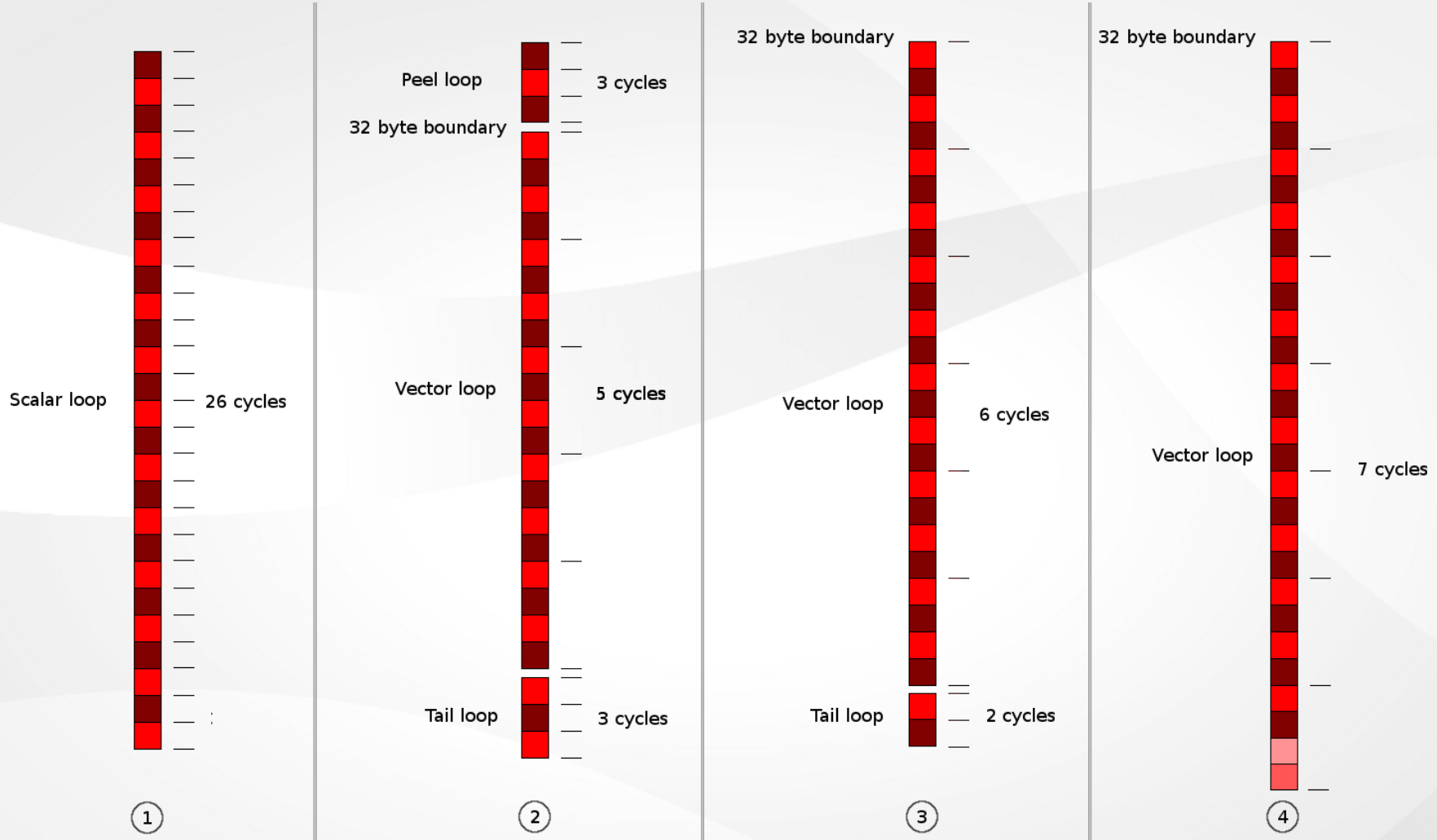
First elements until the 256-bit boundary is met

Vector loop

Vectorized version until the last vector of 4 elements

Tail loop (scalar)

Last elements


Scalar loop
26 cycles
1
Peel loop
3 cycles
32 byte boundary
Vector loop
5 cycles
Tail loop
3 cycles
2
32 byte boundary
Vector loop
6 cycles
Tail loop
2 cycles
3
32 byte boundary
Vector loop
7 cycles
4

Intel specific Compiler directives

To remove the peel loop, you can tell the compiler to align the arrays on a 32 byte boundary using:

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

Then, before using the arrays in a loop, you can tell the compiler that the arrays are aligned. Be careful: if one array is not aligned, this may cause a segmentation fault.

```
!DIR$ VECTOR ALIGNED
do i=1,n
  A(i) = A(i) + B(i)
end do
```

To remove the tail loop, you can allocate A such that its dimension is a multiple of 4 elements:

```
n_4 = mod(n,4)
if (n_4 == 0) then
  n_4 = n
else
  n_4 = n - n_4 + 4
endif
allocate ( A(n_4), B(n_4) )
```

and rewrite the loop as follows:

```
do i=1,n,4
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,3
    A(i+k) = A(i+k) + B(i+k)
  end do
end do
```

In that case, the compiler knows that each inner-most loop cycle can be transformed safely into only vector instructions, and it will not produce the tail and peel loops with the branching. For small arrays, the gain can be significant.

For multi-dimensional arrays, if the 1st dimension is a multiple of 4 elements, all the columns are aligned:

```
double precision, allocatable :: A(:,:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A
allocate( A(n_4,m) )
do j=1,m
 do i=1,n,4
   !DIR$ VECTOR ALIGNED
   !DIR$ VECTOR ALWAYS
   do k=0,3
     A(i+k,j) = A(i+k,j) * B(i+k,j)
   end do
 end do
end do
```

Warning

In practice, using multiples of 4 elements is not always the best choice. Using multiples of 8 or 16 elements can be better because the inner-most loop may be unrolled by the compiler to improve the efficiency of the pipeline.

Instruction-level parallelism (ILP)

MIMD : Multiple instruction, Multiple data

With ILP, different execution units are used in parallel. For example, Sandy-Bridge (2011) x86 CPUs can perform simultaneously:

- 1 vector ADD
- 1 vector MUL
- 2 vector LOADs
- 1 vector STORE
- 1 integer ADD

Ideal for a scalar product (or a matrix product):

```
do i=1,N
  x = x + B(i)*C(i)
end do
```

Peak : 4 ADD + 4 MUL per cycle => 8 flops/cycle. For a 10-core CPU at 2.8GHz: 8 x 2.8E9 x 10 = 224 Gflops/s in double precision

Example:

```
do i=1,N
  A(i) = X(i) + Y(i)
end do
```

and

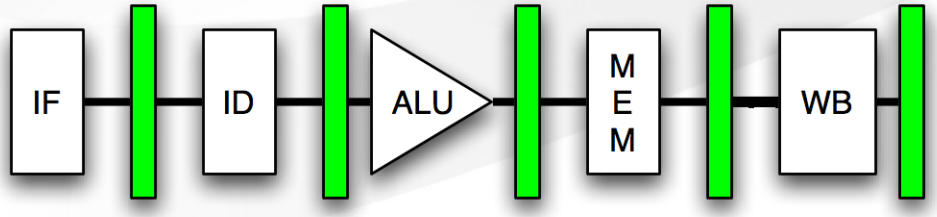
```
do i=1,N
  A(i) = 2.d0*(X(i) + Y(i))
end do
```

take the same amount of time.

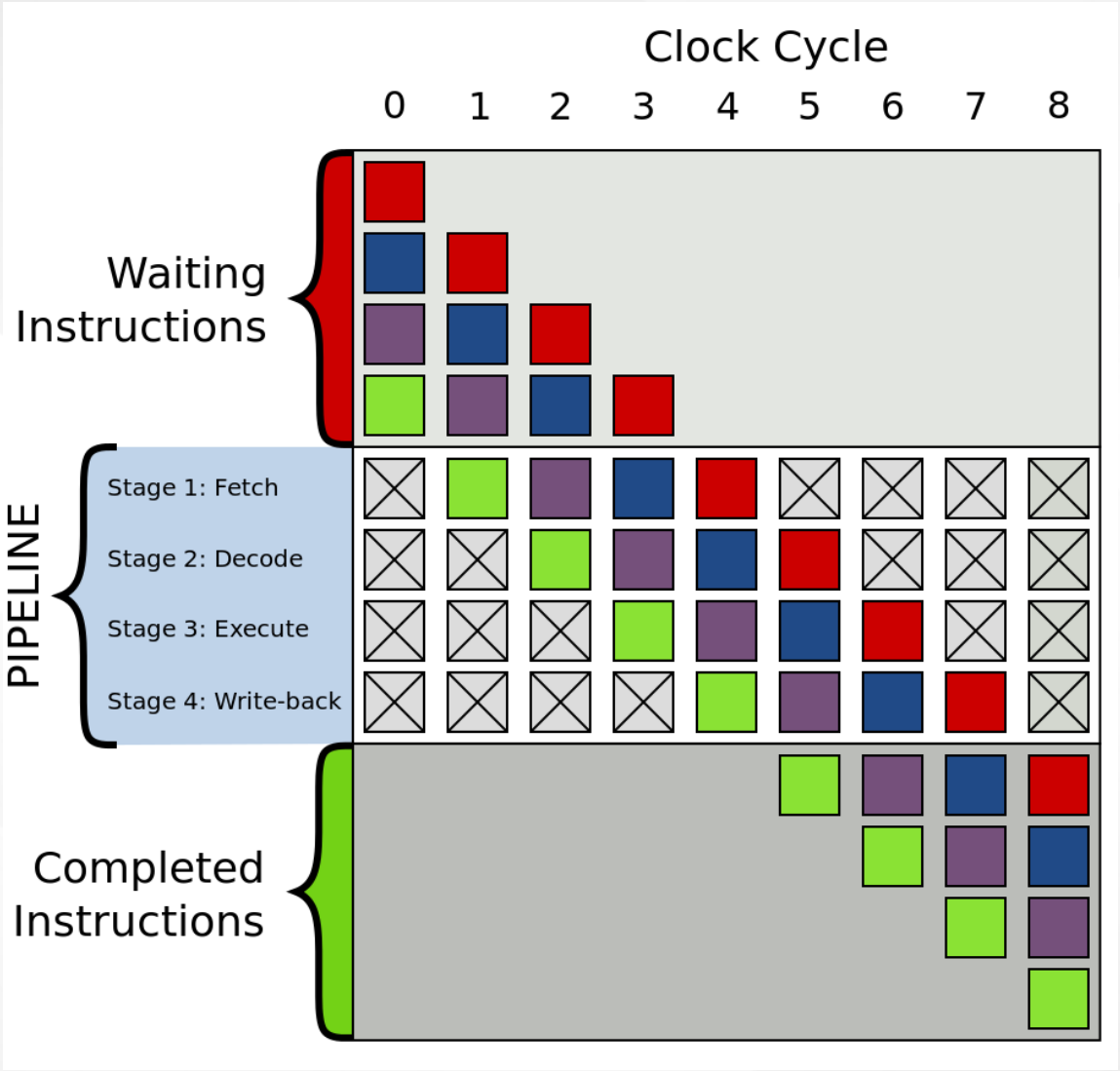
Pipelining

Here we consider a typical RISC processor with 4 different stages to perform an operation:

1. Instruction fetch
2. Instruction decode
3. Execution
4. Memory access+ write-back



Each stage can be executed using different physical units, such that all 4 units can be kept busy:


Clock Cycle
0
1
2
3
4
5
6
7
8
Waiting Instructions
PIPELINE
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed Instructions

In this example:

Latency

4 cycles. It takes 4 cycles to perform one single operation

Throughput

1 cycle. We get one result every cycle

Out of order execution

Inside the CPU, the instructions are not executed in the exact sequence of the code, provided that it does not affect the result: independent instructions can be executed in any order.

The CPU can choose an execution order that improves the efficiency of the pipeline.

Branch prediction

When an *if* statement occurs, two paths can be taken by the program: it is a branch.

The pipeline has to be filled differently depending on the branch.

Branch prediction: the CPU assumes that one branch is more likely to be chosen, and fills the pipeline for it (speculative execution).

If the branch is mispredicted, the pipeline is emptied and the calculation is rolled back.

Branch mispredictions can have a large penalty on the execution.

Many branch predictors exist:

- Static predictor : always assume the condition is true
- Saturating counter : 1. Strongly not taken 2. Weakly not taken 3. Weakly taken 4. Strongly taken
- Two-level adaptive predictor : a branch might be taken depending upon whether the previous two were taken
- Local branch prediction : one history buffer (~4 bits) for each conditional
- Global branch prediction : keep a global history buffer for all branches
- Loop predictor
- etc...

Example:

```
do i=1,N
  if ( mod(i,2) == 0 ) then
    ...
  else
    ...
  endif
end do
```

- Static : 50% success
- Saturating : 50% success
- Local : 100% success (history = 1010)

Links

- "Pipeline-base" by Hellisp - Own work. Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pipeline-base.png

- "Pipeline, 4 stage" by en:User:Cburnett - Own workThis vector image was created with Inkscape.. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pipeline,_4_stage.svg

Summary

- Multiple levels of parallelism : Coarse-grained -> Fine-grained
- Coarse-grained will give the highest level of parallel efficiency (lowest Communication/Computation ratio)
- Different levels of parallelism can be combined

The tools you use should be adapted to your problem:

For example

- doing a Monte Carlo calculation using OpenMP is a bad choice:
  - Shared memory is not required
  - Communication is generally low
  - Synchronization barriers can be avoided
  - Scaling would be limited to the number of cores/node
- diagonalizing a matrix with XML-RPC would not give a good scaling:
  - A lot of communication (matrix products)
  - Synchronizations necessary

If you need to do a Monte Carlo calculation where every Monte Carlo step diagonalizes a very large matrix, you can use OpenMP for the diagonalization and XML-RPC for the distribution of the MC steps.