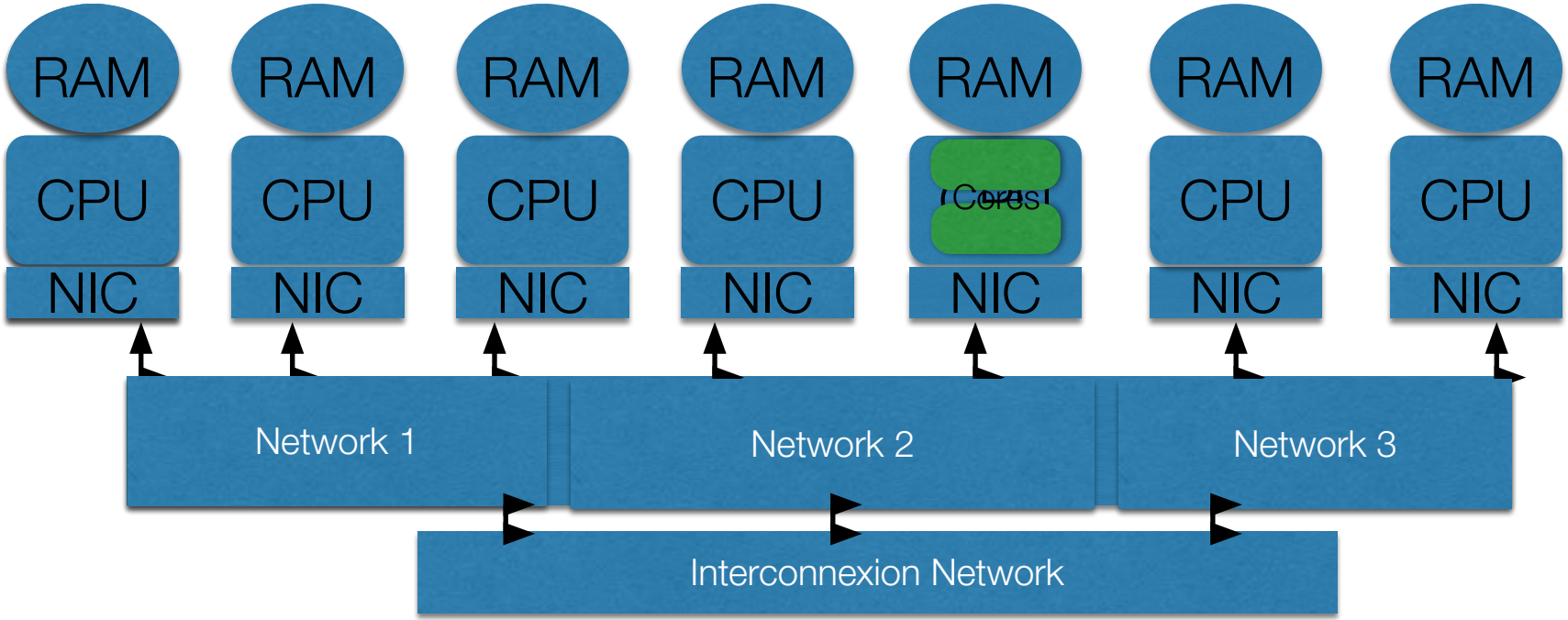
Parallel programming using Message passing paradigm

Georges Da Costa
georges.da-costa@irit.fr

Context



Distributed memory computer

Goals of this lecture

Know the basic mechanisms of programming parallel machines with distributed memory

Learn how to parallelize a code

Know how to evaluate the quality of parallelization

Know the MPI (Message Passing Interface) library

Core characteristics

No shared memory :

- All communications between processes are explicit:
  - message passing
- Or use of a distributed file system
  - (e.g. NFS, Network File System),
  - which is in fact a message passing

Core properties of message passing systems

Explicits parallel processes

Executions are asynchronous, so they must synchronize to send messages to each other

Each process executes potentially different instructions

Processes can be dynamic (in number and therefore in communications)

Explicit communications by passing messages (what?, to whom?)

Parallelization

- Identification of pieces of code that can be executed independently of each other
  - example: (with N processes)

```
for (i=0; i<N; i++) a[i] = i^3;
```

- Identify what needs to be communicated between processes
  - example : (with N processes)

```
b[0] = 1
for (i=1; i<N; i++) b[i] = b[i-1] * 3;
```

Ensure proper load balancing

If we have P processors (numbered from 0 to P-1), how do we parallelize the following code?

```
for (i=0; i<N; i++) a[i] = i^3;
```

Solution 1

T1 = 4*2

Solution 2

T2 = 10*2

Solution 3

T3 = 4*2

What is the time for each solution? (in number of multiplications)
What if P>N ?

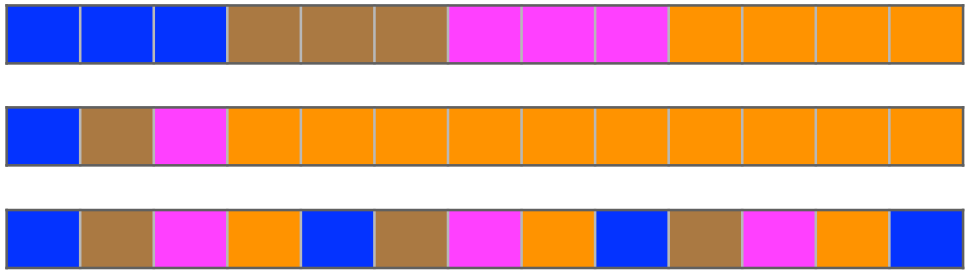
Take into account communication time

If we have P processors, how can we parallelize the following code?

```
b[0] = 1;

for (i=1; i<N; i++) b[i] = b[i-1] * 3;
```

How many network messages in the three following cases ?



M1 = 3

T2 = 3

T3 = 12

Formally

Time of a parallel program = Computation time + communication time

- Reduce the computation time:
  - Add processors, but at the risk of increasing communications
- Reduce the communication time:
  - Asynchronous communications
  - Overlap calculation-communication
  - Avoid waiting between processors (barriers, synchro)

Speed-up

Let **T** = the time of the sequential program

Let **T2 = T//(N)** the time of the program in parallel with N processors

Speedup(N) = S(N) = T / T2

Example :

- A sequential program runs in 12s.
- The same program adapted to run in parallel runs in 4s when it is run in parallel on 5 processors
- Speedup(5)?

S(5) = 12 / 4 = 3. It goes 3 times faster with 5 processors than with 1 processor.

Speedup taxonomy

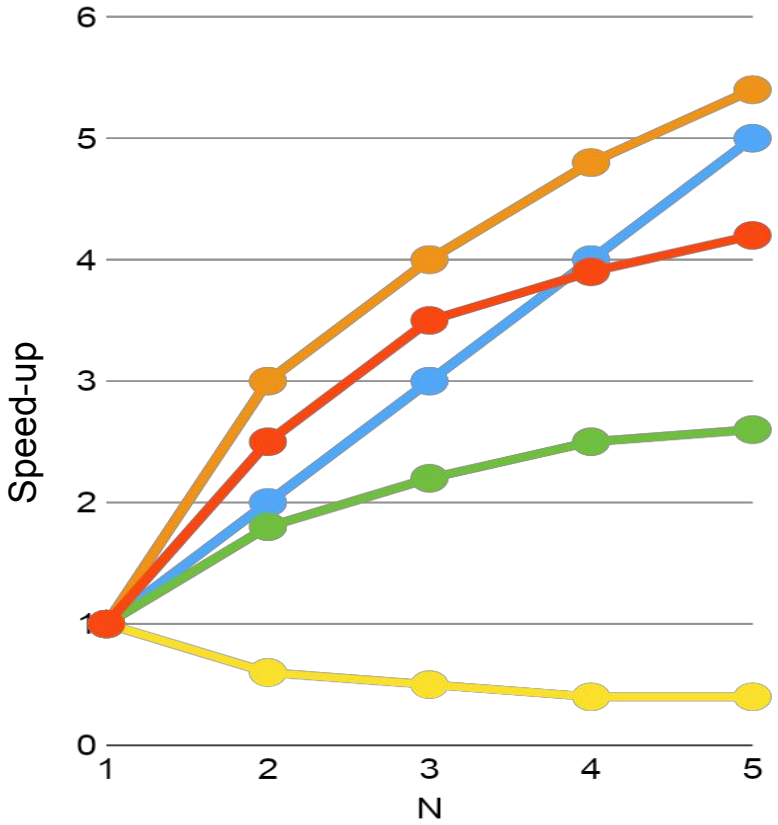
Linear acceleration if the ratio between Ts and T//(N) is equal to N

Sublinear acceleration if the ratio is < N

Slowdown if the ratio is < 1

Over-linear acceleration if the ratio is > N

More complicated situation, the characterization depends on N (example: over-linear then sublinear...)



Amdahl's Law

Not all the code can be parallelized

- For example a read on a parameter file at the beginning of the program.

Let **s** be the percentage of time of the purely sequential program.

T = sT + (1-s)T

so **T//(N) = sT + (1-s) T / N** (with N the number of processors)

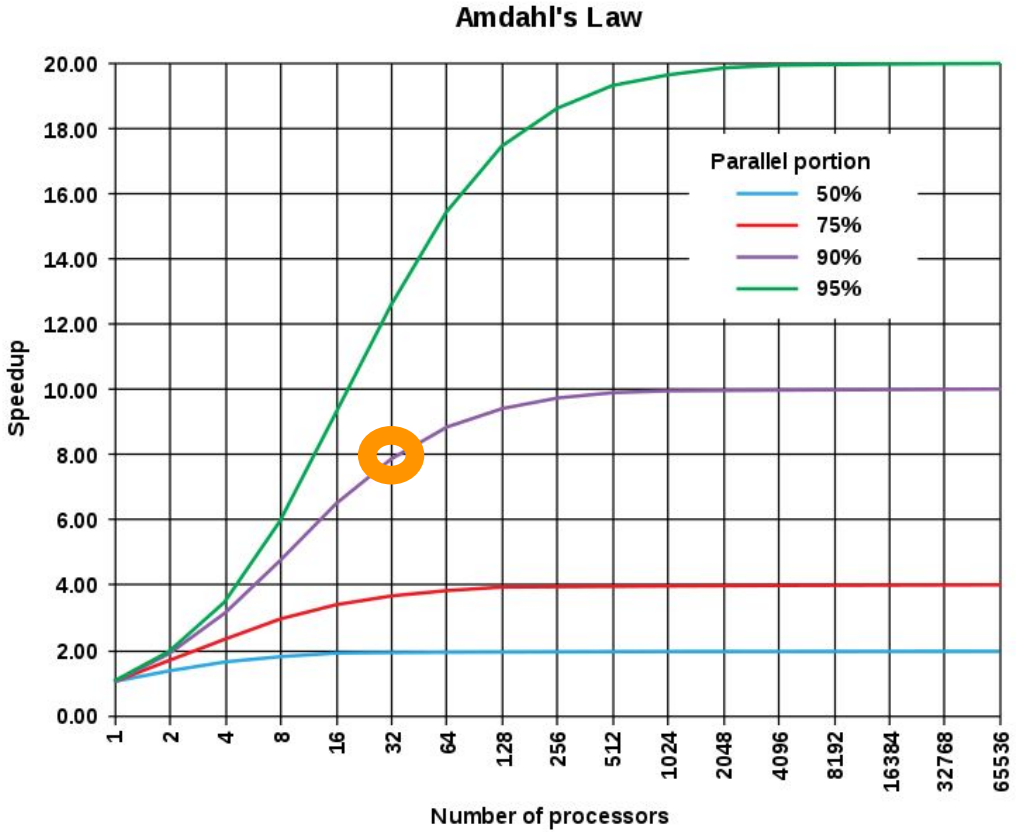
then **S(N) = T / T//(N) = 1 / (s + (1-s) / N)**

Example:

- A sequential program runs in 12s.
- It starts by reading a parameter file for 3s
- The remaining is parallelizable.
- The parallel part benefits linearly from the addition of processors.

What is the acceleration with 5 processors?

$T = 3/12\ T + 9/12\ T \rightarrow T// = 3/12\ T + 9/12\ T / N$
$\rightarrow S(5) = 1 / (3/12 + (9/12) / 5) = 2.5$



$S(N) <= 1/s$

$\lim_{N \to +\infty} S(N) = 1/s$

from Wikipedia

Example: N = 32. If the acceleration is linear on the parallel part, and if s=10%, then S(32) = 1 / (0.10 + 0.90 / 32) = **7.8**

Data parallelism versus task parallelism

Data parallelism: data are distributed on the different processors (example of the previous table **a**)

Parallelism of tasks : the tasks are distributed on the different processors

- Example : at the initialization, an array **tab** is copied on 2 machines P1 and P2.

P1:

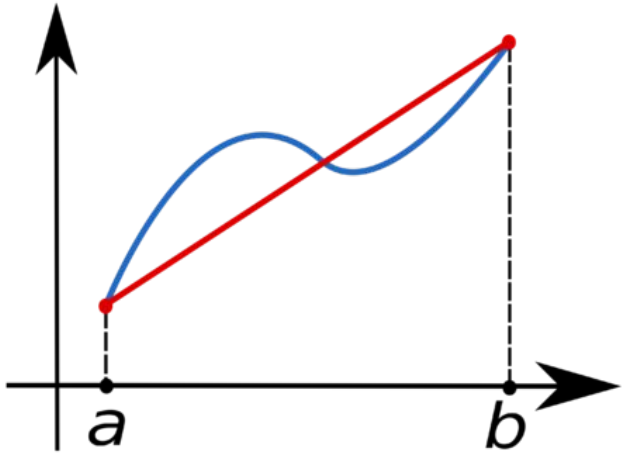
```
while True:
    r1 = compute_sum(tab)
    send_to(2, r1)
    r2 = recv_from(2)
    tab = update(tab, r1*r2)
```

P2:

```
while True
    r1 = compute_product(tab)
    send_to(1, r1)
    r2 = recv_from(1)
    tab = update(tab, r1+r2)
```

Parallelization of a code

Trapezoidal method for estimating an integral

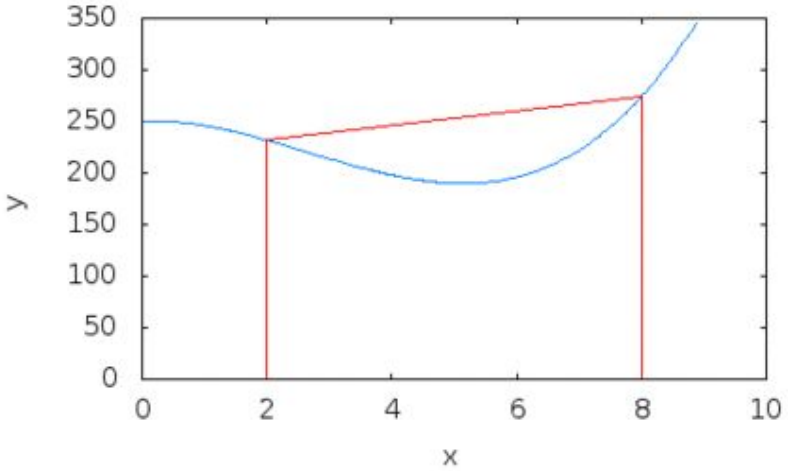


The integral of the function f in blue is approximated by the area under the line in red

$$\int_a^b f(x)\,dx \approx (b-a)\left[\frac{f(a)+f(b)}{2}\right]$$

Parallelization of a code

Trapezoidal method for estimating an integral



$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{k=1}^{N}\left(f(x_{k+1}) + f(x_k)\right)$$

To improve the estimation, intermediate points are added, for a total of N+1 points

Parallelization of a code

It is assumed that we have P processors and N trapezoids with P that divides N.

Algorithm 1:

1. The process **P0** calculates the interval **[a_i,b_i]** that each process **[0...P-1]** will have to calculate, and the number of trapezoids **T_i** in this interval **[a_i,b_i]**.
2. **P0** sends the interval **[a_i, b_i]** and **T_i** to each process **Pi**, with **0<i<P**
3. Each **Pi**, **0<i<P** receives its interval **[a_i, b_i]** and **T_i**
4. Each **Pi**, **0≤i<P**, calculates the function on its interval
5. Each **Pi**, **0<i<P**, sends its local result to **P0**
6. **P0** makes the final sum

Example: P=2, N=10, $y=x^2$, x in [0,1]

- P0 computes intervals for all:
  - P0: [0, 0.5], 5 Trap; P1: [0.5, 1], 5 Trap
- P0 sends [0.5, 1], T_1=5 to P1
- P1 receives them
- Both processors compute in parallel
  - P0 computes 0.0425
  - P1 computes 0.2925
- P1 sends 0.2925 to P0
- P0 computes the final result
  - result = 0.0425 + 0.2925 = 0.335

Parallelization of a code

It is assumed that we have P processors and N trapezoids with P that divides N.

Algorithm 1:

1. The process **P0** calculates the interval **$[a_i,b_i]$** that each process **[0...P-1]** will have to calculate, and the number of trapezoids **T_i** in this interval **$[a_i,b_i]$**.
2. **P0** sends the interval **$[a_i, b_i]$** and **T_i** to each process **Pi**, with **0<i<P**
3. Each **Pi**, **0<i<P** receives its interval **$[a_i, b_i]$** and **T_i**
4. Each **Pi**, **0≤i<P**, calculates the function on its interval
5. Each **Pi**, **0<i<P**, sends its local result to **P0**
6. **P0** makes the final sum

1. What are the sequential steps?
   a. Is it possible to remove them?

2. What if P does not divide N?
   a. Example: N=109, P=10
   b. How to modify the program to tackle this?
   c. This is called load balancing

Comparison with Synchronization

Synchronization

- Multiple workflow at the same time: MIMD
- Each process works at its own speed
- Uses events / shared data to cooperate

Parallelism

- One synchronized workflow: SIMD
- Each process alternates parallel tasks and synchronization
- One process coordinates the tempo

MPI : Message Passing Interface

MPI : Message Passing Interface

- MPI: a standard defining a library of functions for message passing (including on shared memory machines) and parallel I/O
- Specialized implementations optimized for parallel machines (**OpenMPI**, MPICHv2)
- C, C++, Fortran, Java, Perl, OCaml, Python interfaces (**mpi4py**)
- In MPI, a process is executed on a single processor (or on a single core).

How to use MPI

MPI is used by running the mpirun command

- `mpirun -n <nb> <cmd>`
- <nb> is the number of processes
- <cmd> is the command to run

mpirun starts <nb> processes each running <cmd>

Reminder: processes do not share memory. Communication must be explicit !

Simplest mpi program

echo !

Execution:

```
mpirun -n 2 echo Hi!
```

Will print

```
Hi!
Hi!
```

mpirun -n 2 python3 demo.py

Rank = 0

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

if rank == 0:
     data = [24, 17]
else:
     data = None

l_data = comm.scatter(data, root=0)

print(rank, "of", size, ":", l_data)
```

Rank = 1

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

if rank == 0:
     data = [24, 17]
else:
     data = None

l_data = comm.scatter(data, root=0)

print(rank, "of", size, ":", l_data)
```

1 of 2 : 17
0 of 2 : 24

MPI : Communicator

- Communicator: connects a group of processes together.
- They are linked by a certain topology.
- Communication operations are done:
  - within a group (intracommunicator) or
  - between groups (intercommunicator)

At the beginning, the processes are all in the same group (`MPI.COMM_WORLD`)

MPI: Point-to-point communications

Sending and receiving data (`com.send`, `com.receive`)

Blocking and non-blocking

and ready-send (sending is only done if the associated receiving has also been done)

Rarely used

MPI: Collective operations

Communication for all the group

1. Diffusion (broadcast): 1 to all
2. Reduction (reduce): all to 1
3. Full communication: all to all

Example:

- Scatter: takes elements from one root and scatter them on all servers

Remark:

- All processes must execute the command

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

if rank == 0:
     data = [24, 17]
else:
     data = None

l_data = comm.scatter(data, root=0)
l_data = 3 * l_data
res = comm.reduce(l_data, op=MPI.SUM,
root=0)

if rank == 0:
     print(res)
```

MPI: Actions on other processors

Capability to directly change memory on remote processes

- Write on a remote memory
  - Put
- Read on a remote memory
  - Get
- 3 methods to synchronize these communications
  - global, peer to peer, or with remote locks

MPI: Data types

Interoperable MPI type (can be used on all languages):

- Basic types: integers, real (single or double precision, characters, ...): `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`, …
- The derived types (`MPI_Datatype`):
  - contiguous type (`MPI_Type_contiguous`) : homogeneous and contiguous data in memory
  - vector type (`MPI_Type_vector`) : homogeneous data spaced by a constant space in memory
  - indexed type (`MPI_Type_indexed`) : same but variable space in memory
  - structure type (`MPI_Type_create_struct`) : for heterogeneous data

Python-specific type (easier and the type we will use during this lesson)

MPI and Python

```
> mpirun -n 4 python3 hello.py
```

Starts 4 processes, each being a python interpreter

hello.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print ("hello world from process ", rank)
```

`mpirun` can start processes on several computers at the same time. From inside the code, being on one computer or multiple one is transparent

MPI and Python: Communications

2 types of communication primitives :

- Generic python object communication: lowercase primitives are used (ex: comm.send()). Simple use. The one we will use

- Communication of objects organized in an array, for example from the Numpy digital library. Primitives used have their first letter in uppercase (ex: comm.Send()). Very fast, optimized! Not used during the lessons.

Example 1: Python objects

1stMPI.py

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

randNum = random.randint(1,10)

if rank == 1:
        print ("Process", rank, "drew the number", randNum)
        comm.send(randNum, dest=0)

if rank == 0:
        print ("Process", rank, "before receiving has the number", randNum)
        randNum = comm.recv(source=1)
        print ("Process", rank, "received the number", randNum)
```

The treatment on each process is distinguished by its rank

Blocking sending and receiving primitives
Send : the object to send, where ?
Receiving : from whom ? returns the received object

Example 2: Numpy objects

2ndMPI.py

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

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print ("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print ("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print ("Process", rank, "received the number", randNum[0])
```

Blocking sending and receiving primitives
Send : the object to send, where ?
Receiving : from whom ? returns the received object

Example 1bis: Python objects (more complex)

1stBis.py

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

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 2: 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data['a'], " ", data[2])
```

If source = `MPI.ANY_SOURCE`
—> receives from any source

Here data is a Dictionary. Can be any type of Python data.

tag: allows to identify the messages (optional)

Asynchronous primitives

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 2: 3.14}
    req = comm.isend(data, dest=1, tag=11)
    // This process can continue working during the communication
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    // This process can continue working during the communication
    data = req.wait()
    print(data['a'], " ", data[2])
```

Non-blocking send and receive primitives.
wait(): waits for the end of the sending/receiving

Trapezoid method (sequential)

trapezeSerial.py

```
import sys
a, b, n = float(sys.argv[1]), float(sys.argv[2]),  int(sys.argv[3])

def f(x):
       return x*x

def integrateRange(a, b, n):
      integral = -(f(a) + f(b))/2.0
      x, h = a, (b-a)/n
      while (x<b):
            integral = integral + f(x)
            x = x + h
       integral = integral * h
       return integral

integral = integrateRange(a, b, n)

print ("With n =", n, "trapezoids, our estimate of the integral
from", a, "to", b, "is", integral)
```

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{k=1}^{N}\left(f(x_{k+1}) + f(x_k)\right)$$

with h = (b-a) / N

```
$ python3 trapezeSerial.py 0.0 10.0 100000

With n = 100000 trapezoids, our estimate of the integral from
0.0 to 10.0 is 333.333333349691
```

Trapezoid method (parallel)

```
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

def f(x):
        return x*x

def integrateRange(a, b, n):
       integral = -(f(a) + f(b))/2.0
       x = a
       h = (b-a)/n
      while (x<b):
              integral = integral + f(x)
              x = x + h
       integral = integral * h
       return integral
```

trapezeParallel-1.py

```
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local_a is the starting point and local_b is the endpoint
local_a = a + rank*local_n*h
local_b = local_a + local_n*h

# perform local computation. Each process integrates its own interval
integral = integrateRange(local_a, local_b, local_n)

# communication
# root node receives results from all processes and sums them
if rank == 0:
        total = integral
        for i in range(1, size):
                recv_buffer = comm.recv(source=ANY_SOURCE)
                total += recv_buffer
else:
        # all other process send their result
        comm.send(integral, dest=0)

# root process prints results
if rank == 0:
        print ("With n =", n, "trapezoids, our estimate of the integral from" ,
a, "to", b, "is", total)
```

```
$ mpirun -n 4 python3 trapezeParallel-1.py 0.0 10.0 100000
With n = 100000 trapezoids, our estimate of the integral from 0.0 to 10.0 is 333.3302083498043
```

Trapezoid method (parallel V2)

trapezeParallel-2.py

```
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

def f(x):
        return x*x

def integrateRange(a, b, n):
        integral = -(f(a) + f(b))/2.0
        x = a
        h = (b-a)/n
       while (x<b):
                integral = integral + f(x)
                x = x + h
        integral = integral * h
        return integral
```

```
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local_a is the starting point and local_b is the endpoint
local_a = a + rank*local_n*h
local_b = local_a + local_n*h

# perform local computation. Each process integrates its own interval
integral = integrateRange(local_a, local_b, local_n)

# communication
total = comm.reduce(integral, op=MPI.SUM, root=0)

# root process prints results
if rank == 0:
        print ("With n =", n, "trapezoids, our estimate of the integral from" , a, "to",
b, "is", total)
```

Collective operation: Reduce

reduce.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

rank = comm.Get_rank()
somme = 0

somme = comm.reduce(rank, op=MPI.SUM, root=1)
print ("Reduced on 1 : rank=", rank, "somme=", somme)

maxi = comm.allreduce(rank, op=MPI.MAX)
print ("Allreduce on ", rank, "maximum=", maxi)
```

```
$ mpirun -n 4 python3 reduce.py
Reduced on 1 : rank= 3 somme= None
Reduced on 1 : rank= 0 somme= None
Reduced on 1 : rank= 1 somme= 6
Reduced on 1 : rank= 2 somme= None

Allreduce on  0 maximum= 3
Allreduce on  1 maximum= 3
Allreduce on  2 maximum= 3
Allreduce on  3 maximum= 3
```

Possible operations: MPI.SUM, MPI.MAX, MPI.MIN, …

Collective operation: Broadcast

bcast.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

rank = comm.Get_rank()

if rank == 0:
        A = [1.,2.,3., 4.,5.,6., 7.,8.,9.]
else:
        A = None

local_A = comm.bcast(A, root=0)
print ("Local_A on ", rank, "=", local_A)
```

Only the processus 0 knows A

After the Broadcast all nodes know A

```
$ mpirun -n 9 python3 bcast.py
Local_A on  0 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  2 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  1 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  3 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  5 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  6 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  8 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  4 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
Local_A on  7 = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Collective operation: Scatter

At first, only process 0 know A

scatter.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

rank = comm.Get_rank()

if rank == 0:
        A = [1.,2.,3., 4.,5.,6., 7.,8.,9.]
else:
        A = None

local_A = comm.scatter(A, root=0)
print ("Local_A on ", rank, "=", local_A)
```

```
$ mpirun -n 9 python3 scatter.py
Local_A on  0 = 1.0
Local_A on  3 = 4.0
Local_A on  4 = 5.0
Local_A on  5 = 6.0
Local_A on  6 = 7.0
Local_A on  8 = 9.0
Local_A on  7 = 8.0
Local_A on  1 = 2.0
Local_A on  2 = 3.0
```

After Scatter, all processes know they element from A

Important: The number of elements of A **must** be the same as the number of processes

Collective operation: Gather

```
$ mpirun -n 4 python3 gather.py
Local_A on  0 = 0
Local_A on  2 = 2
Local_A on  3 = 3
Local_A on  1 = 1

Global_A After Gather on  1 = None
Global_A After Gather on  2 = None
Global_A After Gather on  3 = None
Global_A After Gather on  0 = [0, 1, 2, 3]

Global_A After AllGather on  0 = [0, 1, 2, 3]
Global_A After AllGather on  1 = [0, 1, 2, 3]
Global_A After AllGather on  2 = [0, 1, 2, 3]
Global_A After AllGather on  3 = [0, 1, 2, 3]
```

gather.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

rank = comm.Get_rank()

local_A = rank
print ("Local_A on ", rank, "=", local_A)

global_A = comm.gather(local_A, root=0)
print ("Global_A After Gather on ", rank, "=", global_A)

global_A = comm.allgather(local_A)
print ("Global_A After AllGather on ", rank, "=", global_A)
```

Mesurer les performances

```
$ mpirun -n 3 python3 time.py
process 0 has [1.0, 2.0, 3.0]
Time on 0 :  0.00011200000000000099
process 1 has [4.0, 5.0, 6.0]
Time on 1 :  0.00012699999999998823
process 2 has [7.0, 8.0, 9.0]
Time on 2 :  0.00011099999999999999
```

time.py

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

comm.barrier()
start = MPI.Wtime()

A = [[1.,2.,3.],[4.,5.,6.],[7.,8.,9.]]
local_a = [0, 0, 0]
local_a = comm.scatter(A, root=0)
print ("process", rank, "has", local_a)

end = MPI.Wtime()

print ("Time on", rank, ": ", end - start)
```

Credits

- https://pythonhosted.org/mpi4py/usrman/tutorial.html
- http://materials.jeremybejarano.com/MPIwithPython/