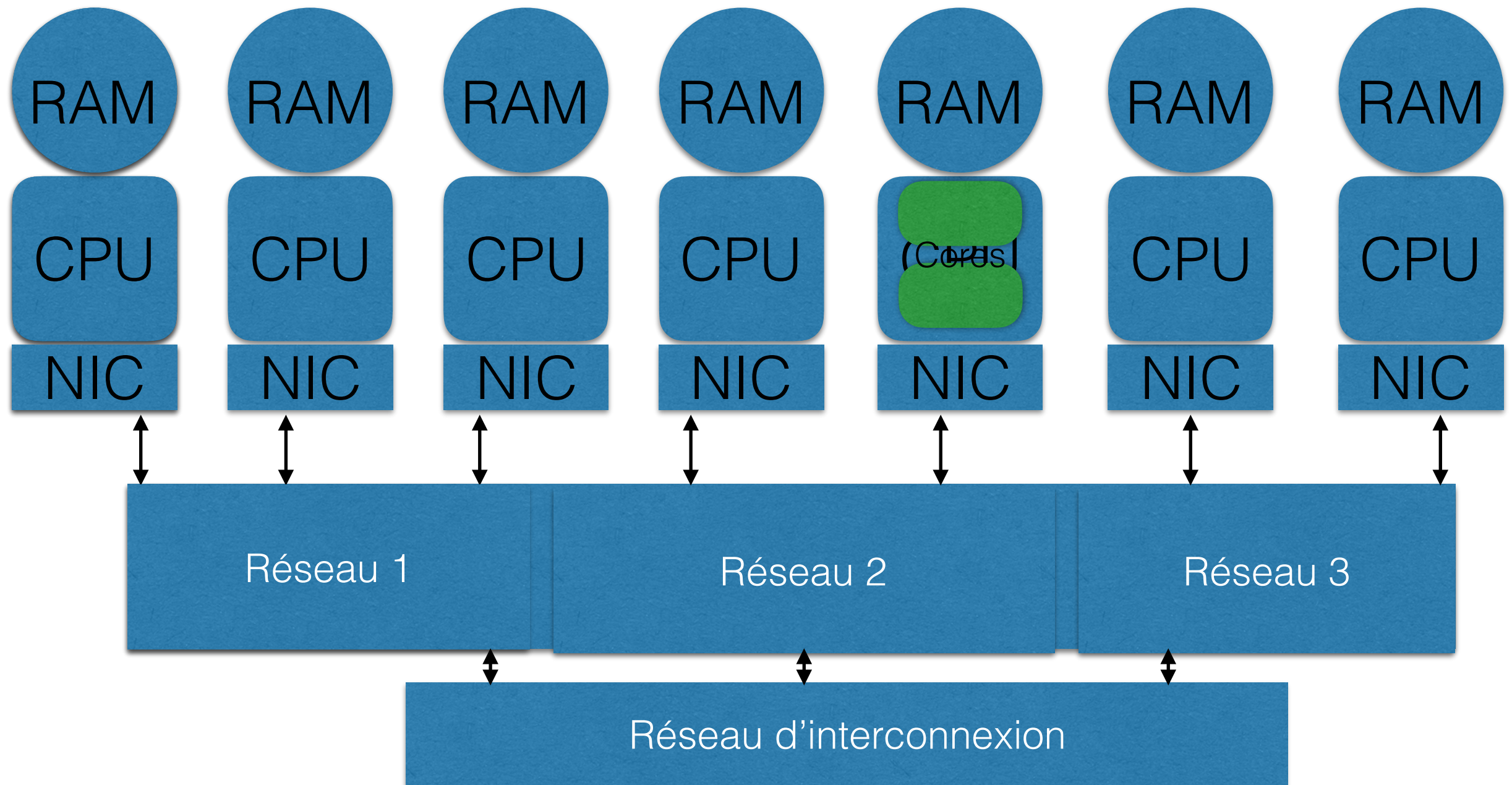


Programmation Parallèle par passage de messages

Master Informatique
Georges Da Costa, Jean-Marc Pierson
georges.da-costa@irit.fr

Contexte



Machine parallèle à mémoire distribuée

Objectifs

- Connaître les mécanismes de base de la programmation de machines parallèles à mémoire distribuée
- Apprendre à paralléliser un code
- Savoir évaluer la qualité de la parallélisation
- Connaître la bibliothèque MPI (Message Passing Interface)

Mécanismes de base

- Pas de mémoire commune :
 - toutes les communications entre les processus sont explicites : passage de messages
 - ou alors utilisation d'un système de fichiers distribués (par ex: NFS, Network File System), ce qui en fait est un passage de message

Propriétés générales du passage de messages

- Expliciter les processus parallèles
- Les exécutions sont asynchrones, il faut donc qu'ils se synchronisent pour s'envoyer des messages
- Chaque processus exécute des instructions potentiellement différentes
- Les processus peuvent être dynamiques (en nombre et donc en communications)
- Communications explicites par passage de messages (quoi ?, à qui ?)

Parallélisation

- Identification des morceaux de code pouvant être exécutés indépendamment les uns des autres

- exemple : (avec N processus)

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

- Identifier ce qui doit être communiqué entre les processus

- exemple : (avec N processus)

```
b[0] = 1
```

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

S'assurer d'un bon équilibrage de charge

Si on a P processeurs (numérotés de 0 à $P-1$), comment paralléliser le code suivant ?

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

Solution 1



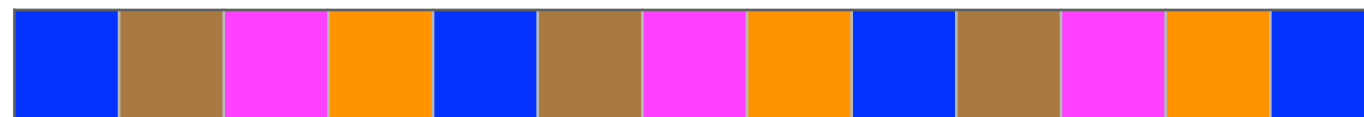
$$T1 = 4 \times 2$$

Solution 2



$$T2 = 10 \times 2$$

Solution 3



$$T3 = 4 \times 2$$

Quel temps pour chaque solution ? (en nombre de multiplications)

Et si $P > N$?

Prendre en compte les temps de communication

Si on a P processeurs, comment paralléliser le code suivant ? Combien de messages dans les trois cas précédents pour le découpage du tableau a (on considère que b est découpé selon la même politique ?

```
b[0] = 1;
```

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

M1=3 ; M2 = 3 ; M3 = 12

Plus formellement

- Temps d'un programme parallèle = Temps de calcul + temps de communication
- Réduire le temps de calcul : ajouter des processeurs, mais au risque d'augmenter les communications
- Réduire le temps de communication :
 - communications asynchrones,
 - recouvrement calcul-communication
 - éviter les attentes entre processeurs (barrières, synchro)

Accélération

- Soit T = le temps du programme en séquentiel
- Soit $T_2 = T/(N)$ le temps du programme en parallèle avec N processeurs
- Accélération = Speedup(N) = $S(N) = T / T_2$
- Exemple : Un programme en séquentiel tourne en 12s. Le même programme adapté pour tourner en parallèle tourne en 4s quand on le fait tourner en parallèle sur 5 processeurs : $S(N) = ?$

$S(5) = 12 / 4 = 3$. Il va donc 3 fois plus vite avec 5 processeurs que avec 1 processeur.

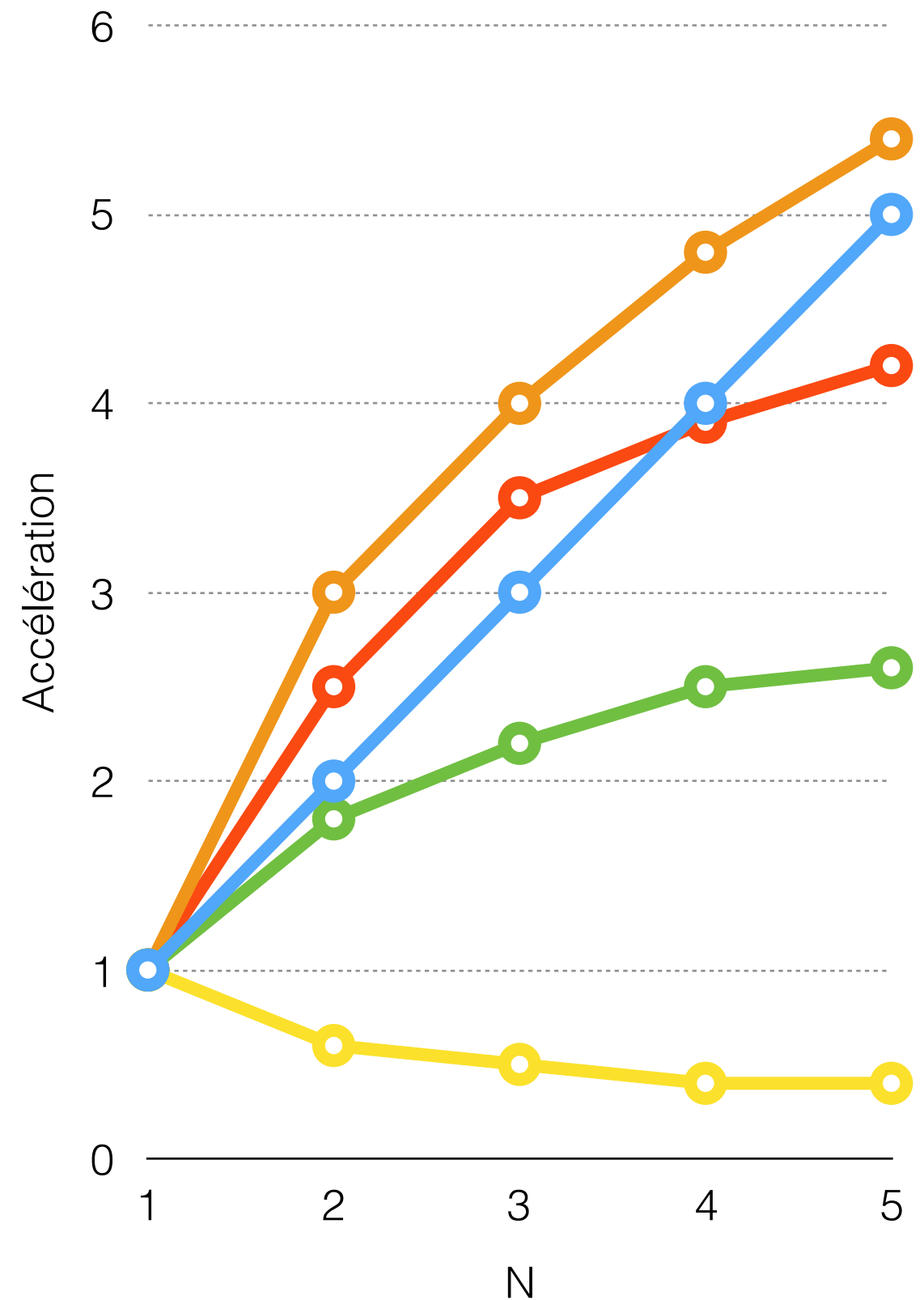
Accélération linéaire si le rapport entre T_s et $T/(N)$ est égal à N

Accélération sous-linéaire si le rapport est $< N$

Ralentissement si le rapport est < 1

Accélération sur-linéaire si le rapport est $> N$

Situation plus compliquée, la caractérisation dépend de N (exemple : surlinéaire puis sous-linéaire...)



Loi de Amdahl

- Tout le code ne peut pas être parallélisé, par exemple une lecture sur un fichier de paramètres au début du programme.
- Soit s le pourcentage de temps du programme purement séquentiel.

$$T = sT + (1-s)T$$

$$\longrightarrow T/(N) = sT + (1-s) T / N$$

(avec N le nombre de processeurs)

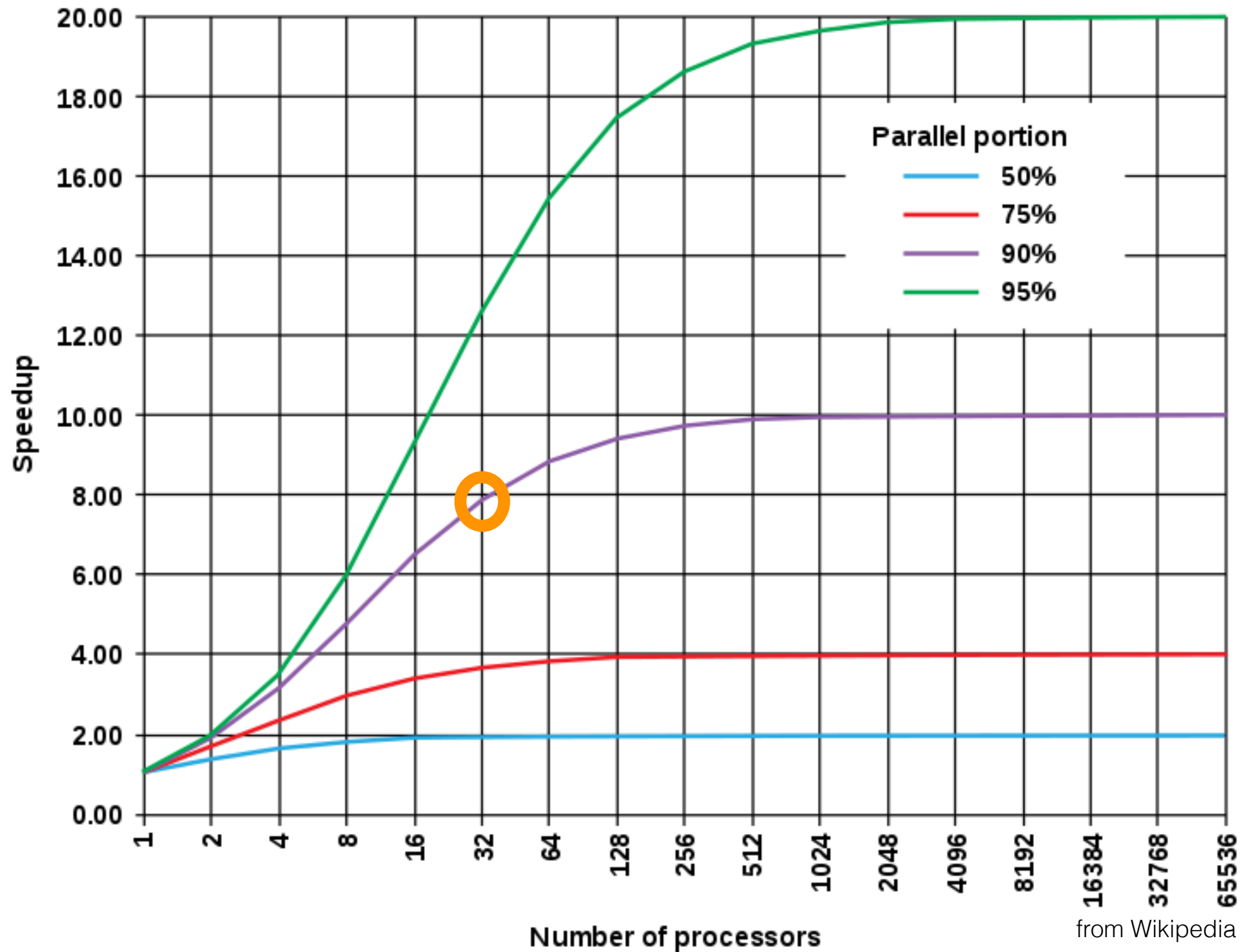
$$\longrightarrow S(N) = T / T/(N) = 1 / (s + (1-s) / N)$$

Exemple: Un programme en séquentiel tourne en 12s. Il commence par lire un fichier de paramètre pendant 3s, avant de faire tourner un code qui est parallélisable. La partie parallèle bénéficie linéairement de l'ajout de processeurs. L'accélération avec 5 processeurs est de ?

$$T = 3/12 T + 9/12 T \longrightarrow T/ = 3/12 T + 9/12 T / N$$

$$\longrightarrow S(5) = 1 / (3/12 + (9/12) / 5) = 2.5$$

Amdahl's Law



$$S(N) \leq 1/s$$

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

Exemple: $N = 32$. Si l'accélération est linéaire sur la partie parallèle, et si $s = 10\%$, alors $S(32) = 1 / (0.10 + 0.90 / 32) = 7.8$

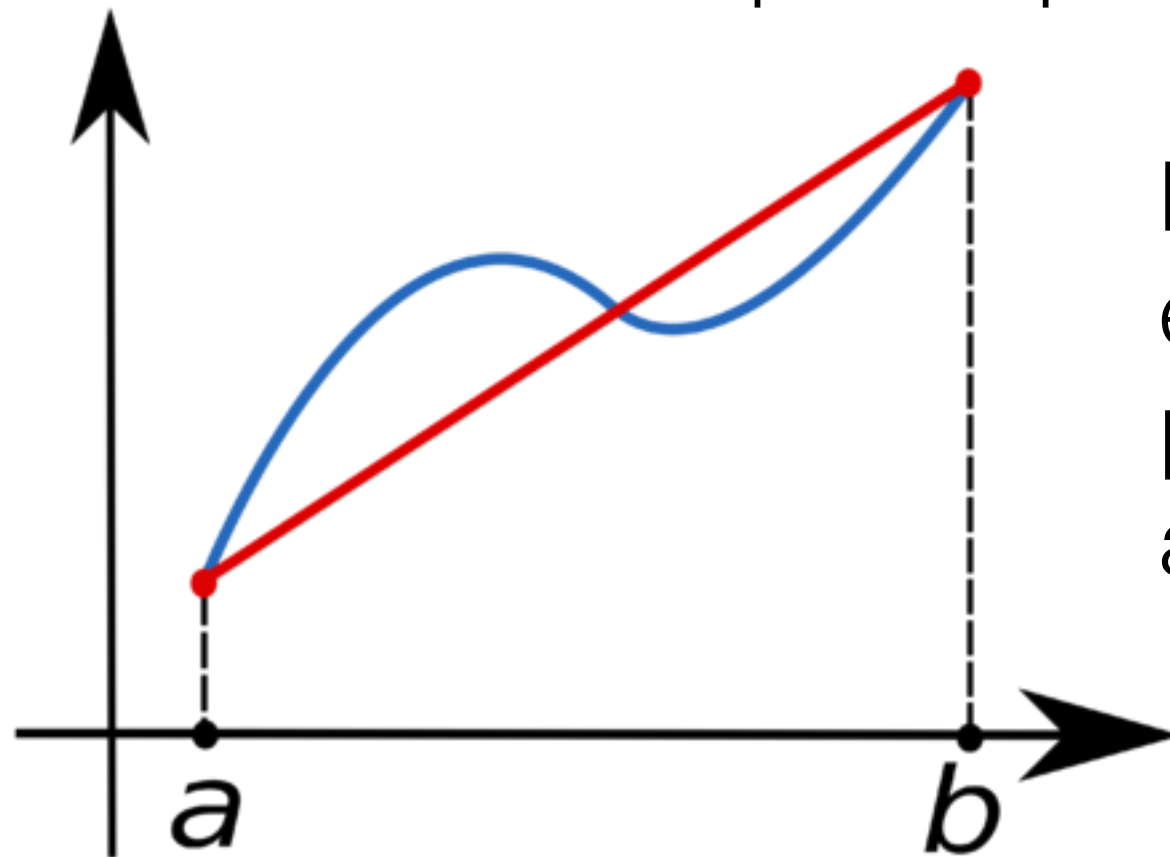
Parallélisme de données contre parallélisme de tâches

- Parallélisme de données : les données sont distribuées sur les différents processeurs (exemple du tableau a précédent)
- Parallélisme de tâches : les tâches sont distribuées sur les différents processeurs
 - exemple : à l'initialisation, un tableau a est recopié sur 2 machines P1 et P2.
 - P1 : `while (1) { res = calculer_somme(a); send_to(2, res); recv_from(2, a); }`
 - P2 : `while (1) { recv_from(1, res); a = transform(a, res); send_to(1, a); }`

Parallélisation d'un code

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

Méthode des trapèzes pour estimer une intégrale



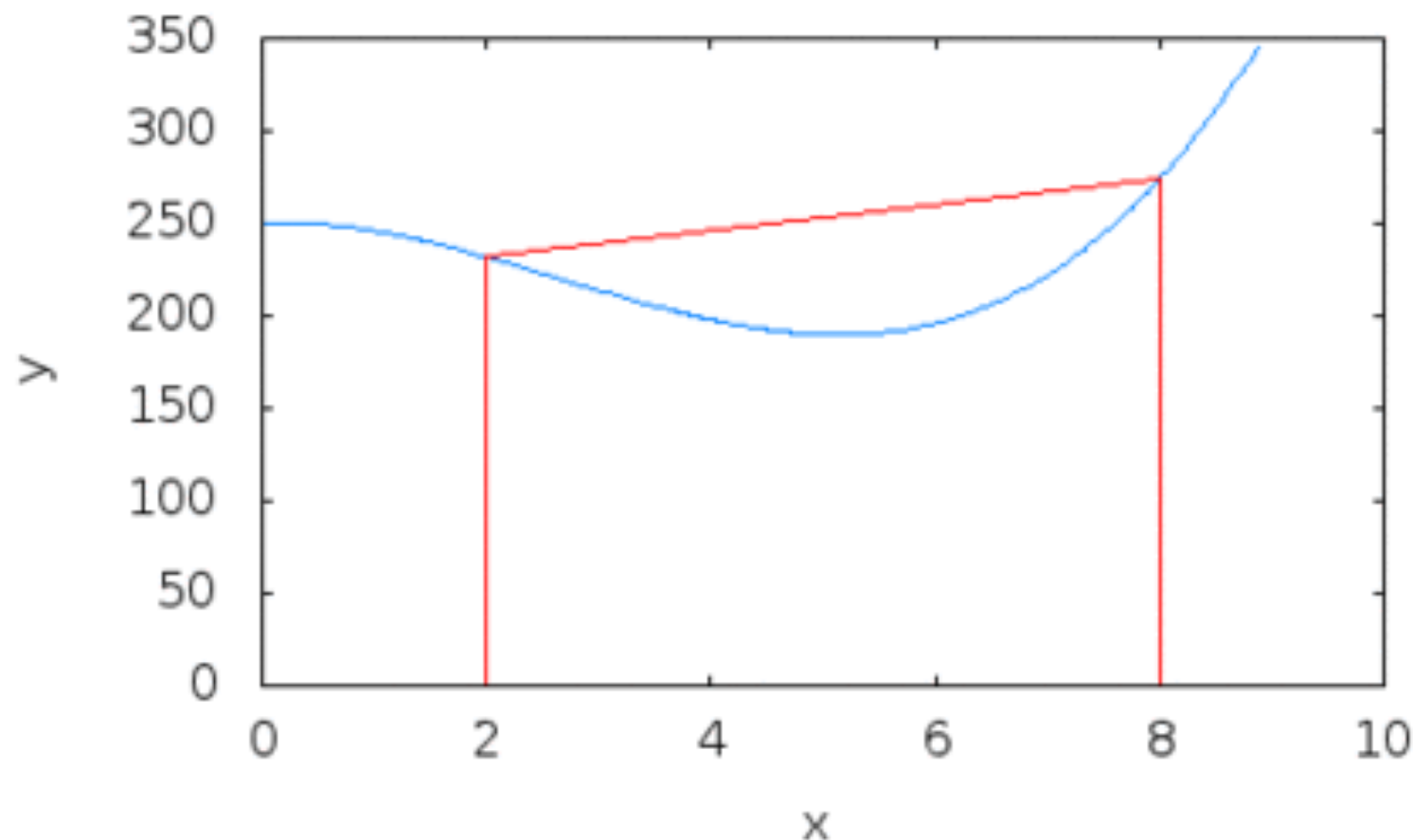
L'intégrale de la fonction f en bleue est approchée par l'aire sous la droite affine en rouge

Parralélisation d'un code

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

avec $h = (b-a) / N$

Méthode des trapèzes pour estimer une intégrale



Pour améliorer l'estimation, on ajoute des points intermédiaires, pour un total de $N+1$ points

Parallélisation du code

- On suppose que l'on a P processeurs.
- On suppose que l'on a N trapèzes.
- On découpe les N trapèzes sur les P processeurs. On suppose que P divise N .
- Algorithme 1 :
 - Le processus P_0 calcule l'intervalle $[a_i, b_i]$ que chaque processus $[0 \dots P-1]$ devra calculer, et le nombre de trapèzes T_i dans cet intervalle $[a_i, b_i]$
 - P_0 envoie l'intervalle $[a_i, b_i]$ et T_i à chaque processus P_i , avec $P > i > 0$
 - Chaque P_i , $P > i > 0$ reçoit son intervalle $[a_i, b_i]$ et T_i
 - Chaque P_i , $P > i \geq 0$, calcule la fonction sur son intervalle
 - Chaque P_i , $P > i \geq 0$, envoie son résultat local à P_0
 - P_0 fait la somme finale

- Algorithme 2 : peut-on se passer de l'étape initiale où P0 envoie les intervalles à tous les autres ?
- Algorithme 3 : que se passerait-il si P ne divise pas N (exemple: $N=109$, $P=10$) ? Comment modifier le programme pour que la charge soit équilibrée entre les processeurs (load-balancing) ?

MPI : Message Passing Interface

- MPI : une norme définissant une librairie de fonctions pour le passage de message (y compris sur machine à mémoire partagée) et les entrées/sorties parallèles
- Des implémentations spécialisées et optimisées pour les machines parallèles (OpenMPI, MPICHv2)
- Des interfaces C, C++, Fortran, Java, Perl, OCaml, Python (mpi4py)
- Dans MPI, un processus est exécuté sur un seul processeur (ou sur un seul coeur).

MPI : Communicator

- Communicator : connecte un groupe de processus ensemble. Ils sont liés par une certaine topologie. Les opérations de communication sont faites dans un groupe (intracommunicator) ou entre groupes (intercommunicator)
- Au départ, les processus sont tous dans le même groupe (`MPI_COMM_WORLD`)

MPI : Communications point à point

- Envoi et réception de données (`MPI_Send`, `MPI_Recv`)
- Bloquante et non-bloquante
- et ready-send (l'envoi n'est fait que si la réception associée a été aussi faite)

MPI : Communications collectives

- Communications qui implique tout le groupe
- Diffusion (`MPI_BCast`, 1 vers N)
- Réduction (`MPI_Reduce`, N vers 1),
- `MPI_AllToAll` (N vers N)
- ...

MPI : Actions sur un autre processeur

- Ecrire dans une mémoire distante : `MPI_Put`
- Lire dans une mémoire distante : `MPI_Get`
- 3 méthodes pour synchroniser ces communications (globale, pair à pair ou avec des verrous distants)

MPI : les données

- Les types de base : entiers, réels (simple ou double précision, caractères, ...) : `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`, ...
- Les types dérivés (`MPI_Datatype`):
 - type contigu (`MPI_Type_contiguous`) : données homogènes et contiguës en mémoire
 - type vecteur (`MPI_Type_vector`) : des données homogènes espacées par un espace constant en mémoire
 - type indexé (`MPI_Type_indexed`) : pareil mais espace variable en mémoire
 - type structure (`MPI_Type_create_struct`) : pour des données hétérogènes

MPI et python

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

On lance avec 4 processus, chaque processus est une instance de python3

hello.py

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

mpirun, mpiexec sont des synonymes
mpirun possède de nombreuses options : spécifier les programmes à lancer, les machines/coeurs où les lancer, les entrées/sorties, ... —> `man mpirun`

Python et les communications

2 types de primitives de communications :

- communication d'objets python générique : on utilise les primitives en minuscules (ex: `comm.send()`). Utilisation simple.
- communication d'objets organisés en tableau, par exemple de la bibliothèque numérique Numpy. Les primitives utilisées ont leur première lettre en majuscule (ex: `comm.Send()`). Très rapide, optimisé !

Exemple n°1: objets python

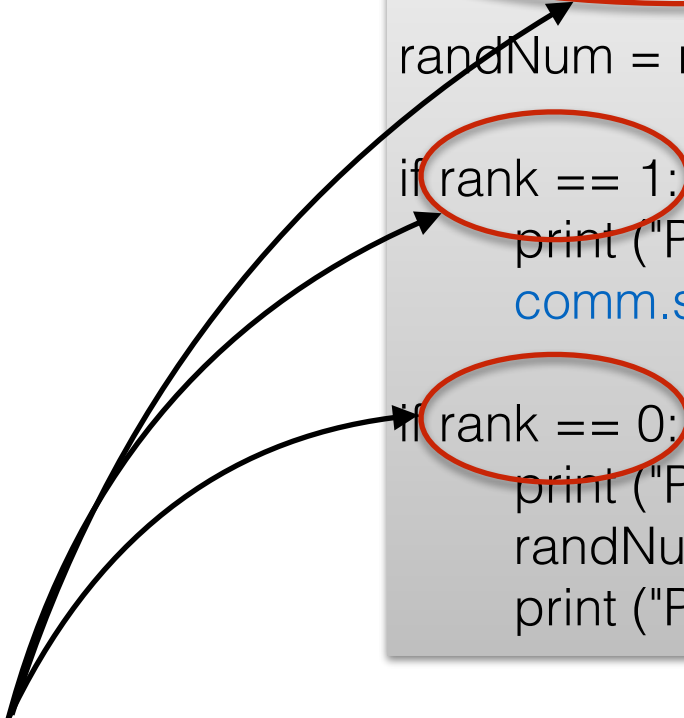
1stMPI.py

```
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)
```



On distingue le traitement
sur chaque processus
grâce à son **rank**

Primitives d'envoi et de réception
bloquantes

Envoi : l'objet à envoyer, où ?

Réception : de qui ? retourne l'objet reçu

Exemple n°2: objets numpy

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])
```

Primitives d'envoi et de réception bloquantes.

Envoi : l'objet à envoyer, où ?

Réception : l'objet à recevoir, de qui ?

Exemple n°1bis: objets python (plus compliqué)

1stBis.py

Si source
=MPI.ANY_SOURCE
—> reçoit
n'importe qui

```
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])
```

data est un tableau indexé (en python, index avec ce que l'on veut - ou presque)

tag: permet d'identifier les messages (optionnel)
tag=MPI.ANY_TAG

Exemple n°2bis: objets numpy plus compliqué

2ndBis.py

```
from mpi4py import MPI
import numpy

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

# VERSION 1 : On explicite les types dans le tableau
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
    print(data[1:100])

print("***** VERSION 2 *****")
# On laisse découvrir automatiquement les types
if rank == 0:
    data = numpy.arange(1000, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(1000, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
    print(data[1:100])
```

Primitives asynchrones

1stTer.py

```
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)
    // ce processus peut faire quelque chose ici pendant la communication
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    // ce processus peut faire quelque chose ici pendant la communication
    data = req.wait()
    print(data['a'], " ", data[2])
```

Primitives d'envoi et de réception non bloquantes.
wait(): attend la fin de l'envoi/réception

Méthode des trapèzes (séquentiel)

trapezeSerial.py

```
import sys
```

```
#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
```

```
integral = integrateRange(a, b, n)
```

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

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

avec $h = (b-a) / N$

```
$ 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
```


Méthode des trapèzes (parallèle v1)

```
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
```

Méthode des trapèzes (parallèle v2)

```
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-2.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
```

```
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)
```

Opérations collectives MPI: Reduction

reduce.py

```
$ 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
```

```
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)
```

Opérations possibles: MPI.SUM, MPI.MAX, MPI.MIN, ...

Opérations collectives MPI:

Diffusion

bcast.py

Ici, seul le processus 0 connaît A ↗

```
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)
```

Après broadcast, tous connaissent 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]
```

Opérations collectives MPI: distribution à plusieurs

scatter.py

Ici, seul le processus 0 connaît 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
```

```
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)
```

Après scatter, tous connaissent leur morceau de A
Attention: le nombre d'éléments dans A doit être
égal au nombre de processus du communicator

Opérations collectives MPI: rassemblement

gather.py

```
$ 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]
```

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

rank = comm.Get_rank()

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

comm.barrier()

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

comm.barrier()

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

barrier(): tous les processus du groupe s'attendent

Mesurer les performances

time.py

```
$ mpirun -n 3 python3 time.py
process 0 has [1.0, 2.0, 3.0]
Durée sur 0 : 0.000112000000000000099
process 1 has [4.0, 5.0, 6.0]
Durée sur 1 : 0.00012699999999999998823
process 2 has [7.0, 8.0, 9.0]
Durée sur 2 : 0.0001109999999999999999
```

```
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()
```

```
comm.barrier()
```

```
print ("Durée sur", rank, ": ", end - start)
```

Exercices

- On suppose que le processus 0 possède deux vecteurs X et Y , dont il veut faire le produit scalaire $X.Y$. Soit N la taille des vecteurs, connu de lui seul. On suppose que l'on a la fonction de base `produitScalaire(X,Y)` qui renvoie le résultat du produit scalaire de X par Y . Donner l'algorithme parallèle MPI pour paralléliser le produit scalaire sur P processeurs.
- Reprendre l'exemple de l'exercice sur les N-bodies en programmation MPI. (exercice 10, TD3 de OpenMP).
- Reprendre l'exemple avec la convergence infinie (exercice 8, TD3 de OpenMP).

Credits

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