



HAL
open science

Prolégomène au calcul parallèle

Jalel Chergui

► **To cite this version:**

| Jalel Chergui. Prolégomène au calcul parallèle. Licence. Orsay., France. 2017, pp.63. hal-04452438

HAL Id: hal-04452438

<https://hal.science/hal-04452438>

Submitted on 12 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Prolégomène au calcul parallèle

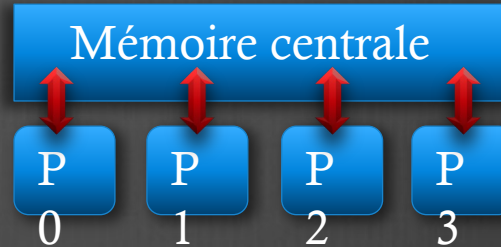
Jalel.Chergui@lisn.fr

Plan du cours

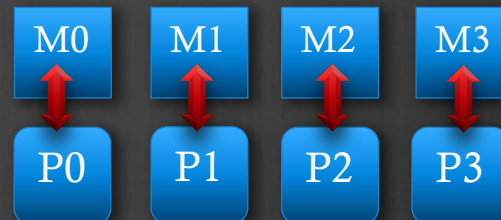
1. Notions d'architectures
2. Modèles de programmation
3. Parallélisation par échanges de messages
4. Application
5. Travaux pratiques
6. Mesure de performance
7. Exercices

Architecture mémoire des machines

① Mémoire partagée →

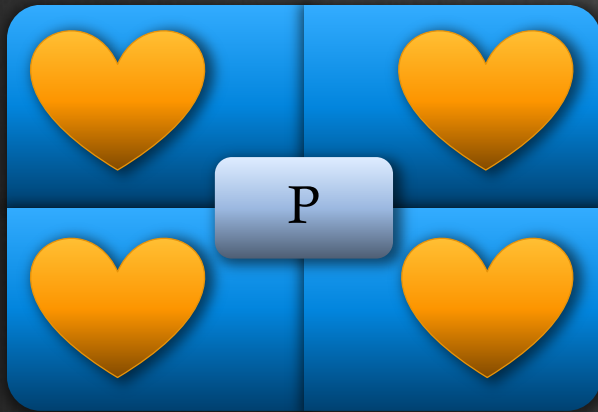


① Mémoire distribuée →



Architecture des processeurs

Aujourd'hui, les **P**rocesseurs sont équipés de plusieurs cœurs. Chaque cœur peut être composé de plusieurs *threads* (processeurs Intel, AMD, IBM).

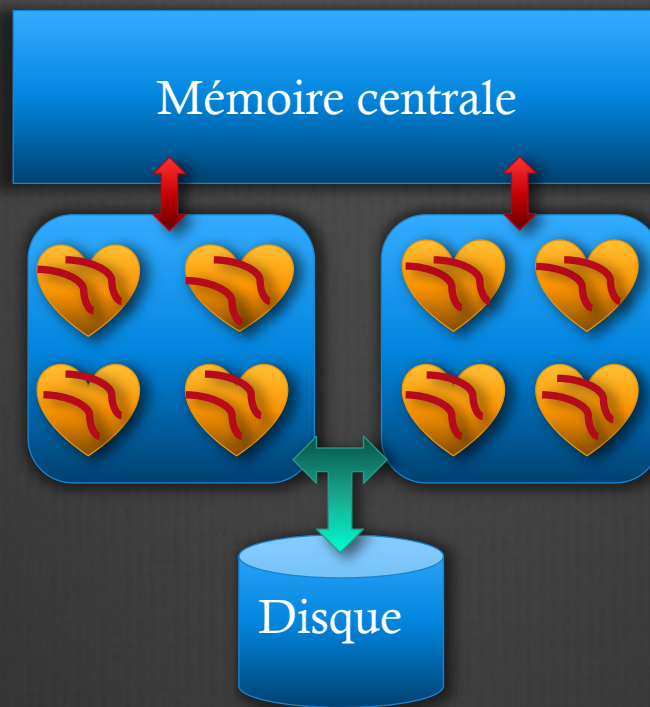


Un processeur équipé de 4 cœurs

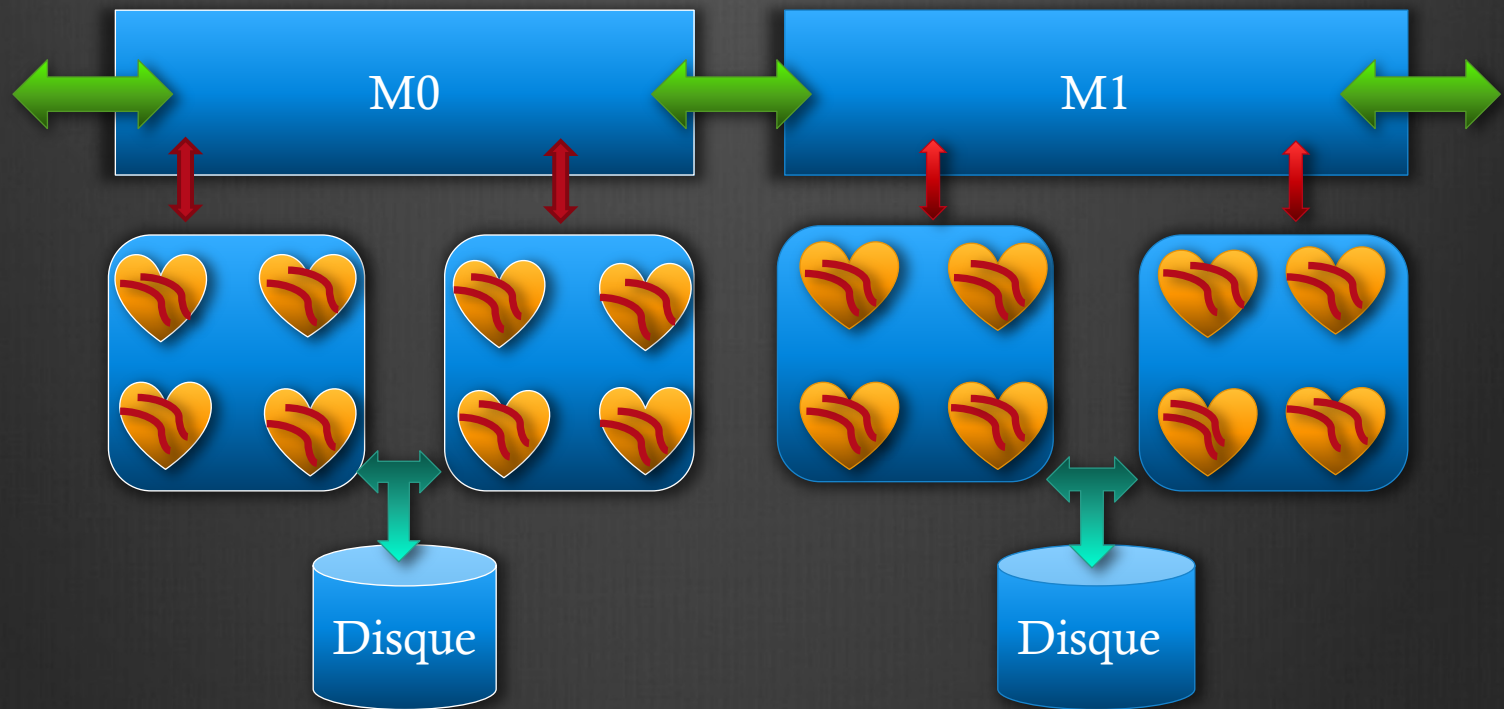


Un cœur équipé de 4 threads

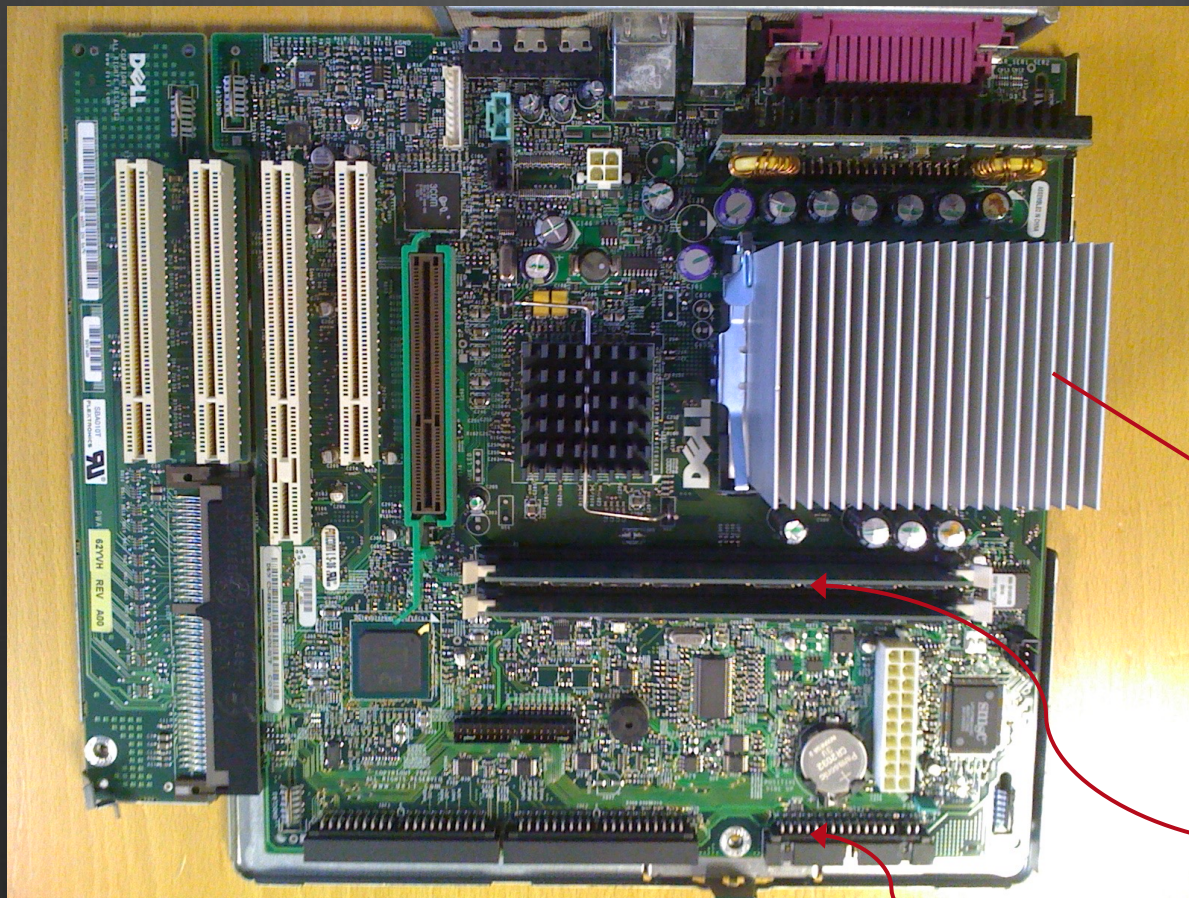
Un serveur de calcul



Un serveur de calcul parallèle



Une carte mère



Processeur

Mémoire

Connecteur disque

Performance liée au matériel

- ✓ Fréquence d'horloge du processeur.
- ✓ Nombre de processeurs (sockets) par serveur de calcul.
- ✓ Nombre de cœurs par processeur.
- ✓ Nombre de threads par cœurs (*hyper-threading*).
- ✓ Taille des caches mémoires.
- ✓ Taille de la mémoire centrale (partagée par les coeurs).
- ✓ Fréquence et bande-passante du bus mémoire sur un serveur de calcul (mémoire partagée).
- ✓ Nombre de serveurs de calcul interconnectés.
- ✓ Latence, bande-passante et topologie du réseau d'interconnexion sur un serveur de calcul parallèle (mémoire distribuée).

Quel model de programmation parallèle ?

Selon l'architecture mémoire du serveur de calcul parallèle.

- ✧ En général par échange de messages (*MPI*) quelque soit l'architecture mémoire.
 - ✧ Si la mémoire est partagée, la parallélisation du code peut se faire avec des processus légers (*multi-threadings*) : *OpenMP* ou directement avec la bibliothèque *threads* standard.
 - ✧ Les deux à la fois (parallélisation hybride à 2 niveaux): *MPI* et *OpenMP*.
- ⇒ En utilisant le langage informatique favori (*Fortran* , *C* , *C++*) !

Calcul d'une norme de matrice

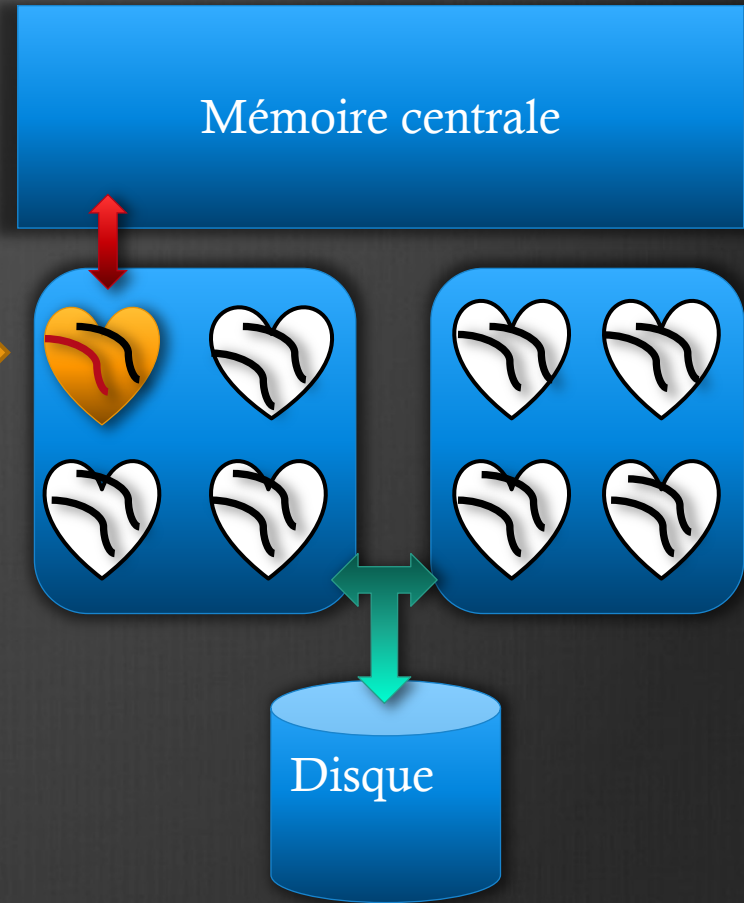
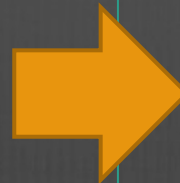
$$R = \sqrt{\sum_{ij} a_{ij}^2}$$

Programmation en Fortran ...

Programmation séquentielle

```
Program norme
implicit none
integer, parameter :: N=128,M=256
integer :: i, j
real, dimension(M,N) :: a
real :: S=0, R

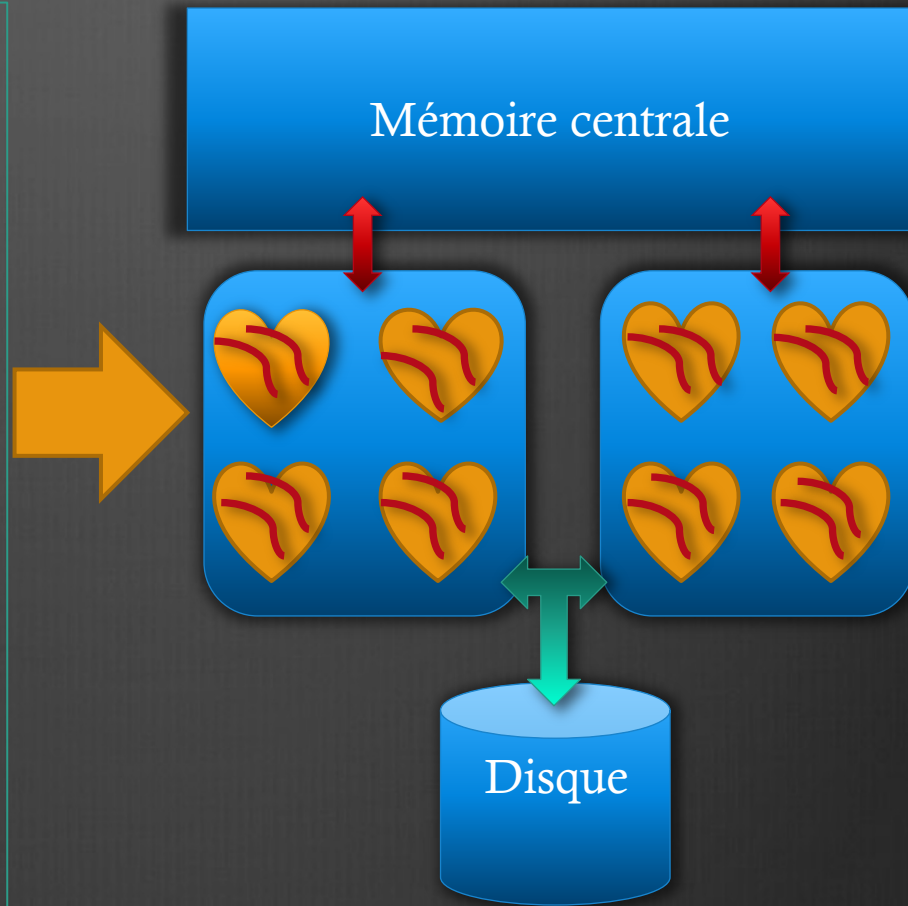
call random_number(a(:,,:))
do j=1, N
  do i=1,M
    S=S+a(i,j)*a(i,j)
  end do
end do
R=sqrt(S)
print *, 'Norme=' , R
End program norme
```



Programmation parallèle avec OpenMP

Mémoire partagée

```
Program norme
  implicit none
  integer,          parameter
  :: N=128,M=256
  integer           :: i, j
  real, dimension(M,N) :: a
  real              :: R=0.
  call random_number(a(:,,:))
  !$OMP PARALLEL DO &
  !$OMP REDUCTION (+:R)
  do j=1, N
    do i=1,M
      R=R+a(i,j)*a(i,j)
    end do
  end do
  !$OMP END PARALLEL DO
  print *, 'Norme=', sqrt(R)
End program norme
```



Programmation parallèle avec MPI

Mémoire distribuée

Program norme

use **MPI**

implicit none

integer, parameter :: N=128, M=256

integer :: i, j, code

real, dimension(M,N) :: a

real :: S=0., R

call **MPI_Init**(code)

call random_number(a(:, :))

do j=1, N

do i=1, M

S=S+a(i,j)*a(i,j)

end do

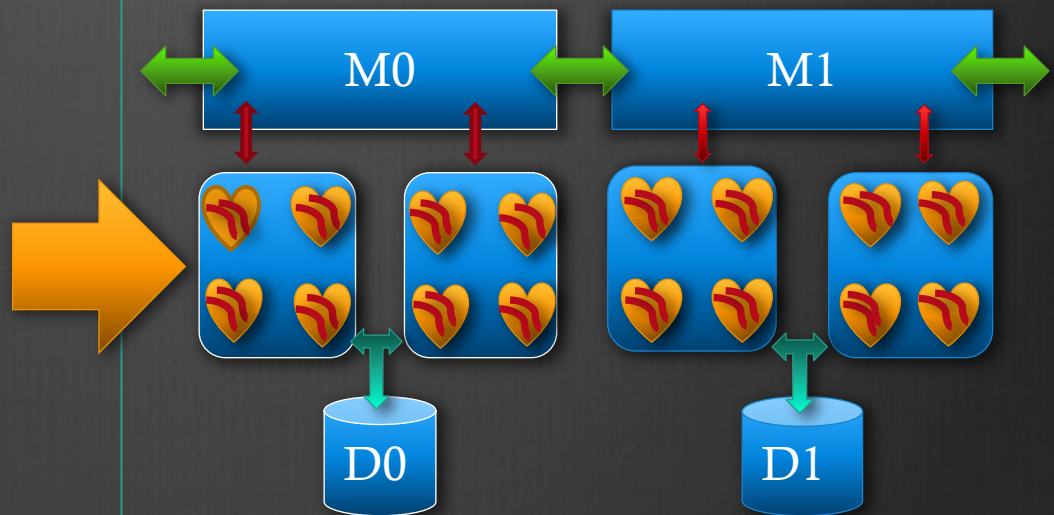
end do

call **MPI_Allreduce**(S, R, 1, ..., **MPI_SUM**, ...)

print *, 'Norme=', sqrt(R)

call **MPI_Finalize**(code)

End program norme



Programmation parallèle hybride

Mémoire mixte

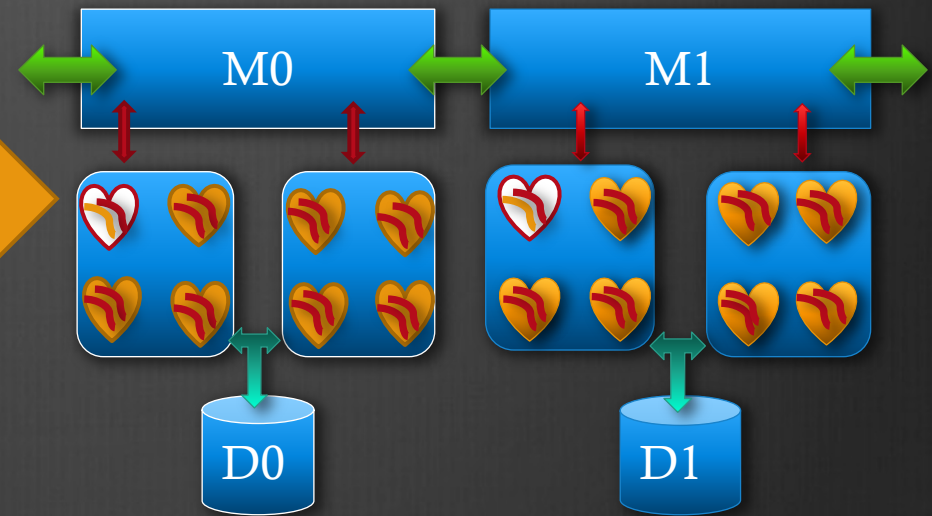
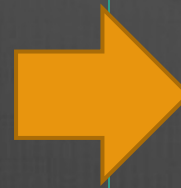
```
Program norme
use MPI
implicit none
integer, parameter :: N=128,M=256
integer           :: i, j, code
real, dimension(M,N) :: a
real              :: S=0., R

call MPI_Init(code)
call random_number(a(:,,:))

!$OMP PARALLEL DO &
!$OMP REDUCTION (+:S)
do j=1, N
  do i=1, M
    S=S+a(i,j)*a(i,j)
  end do
end do
!$OMP END PARALLEL DO

call MPI_Allreduce(S, R, ..., MPI_SUM, &
                  ..., code)
print *, 'Norme=', sqrt(R)

call MPI_Finalize(code)
End program norme
```



Performance liée au programme

- Optimiser le code séquentiel.
- Optimiser les accès à la mémoire.
- Choisir un algorithme parallèle.
- Paralléliser avec MPI.
- Ensuite ajouter OpenMP si programmation hybride.
- Minimiser les communications et les synchronisations inter-processus et *inter-threads*.
- Minimiser le coût des entrées/sorties sur les disques.

Pourquoi le calcul parallèle ?

- ✓ Codes de calcul de plus en plus gourmands en mémoire centrale (simulations 3D, enrichissement des modèles physiques, etc.).
 - ✓ Verrou sur la fréquence des processeurs (consommation énergétique accrue, miniaturisation des circuits intégrés, etc.)
- ⇒ calcul parallèle par échanges de messages sur une machine à mémoire distribuée.

Parallélisation avec MPI

Historique

- ✧ Initiative conjointe issue à la fois de constructeurs de machines et d'universitaires qui a véritablement démarrée au début des années 90.
- ✧ Volonté de définir un environnement parallèle standard de développement avec la bibliothèque MPI (*Message Passing Interface*). Premier formalisme établi en 1992.
- ✧ Point de départ d'un véritable foisonnement d'algorithmes et de bibliothèques d'algèbre linéaire et d'applications parallèles en particulier ...
- ✧ *MPI-4* : en cours d'élaboration

Bibliographie

- ❖ *Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.0, High Performance Computing Center Stuttgart (HLRS), 2012.* <https://fs.hlr.de/projects/par/mpi/mpi30/>
- ❖ William Gropp, Ewing Lusk et Rajeev Thakur : *Using MPI-2*, MIT Press, 1999.
- ❖ Peter S. Pacheco : *Parallel Programming with MPI*, Morgan Kaufman, Ed., 1997.

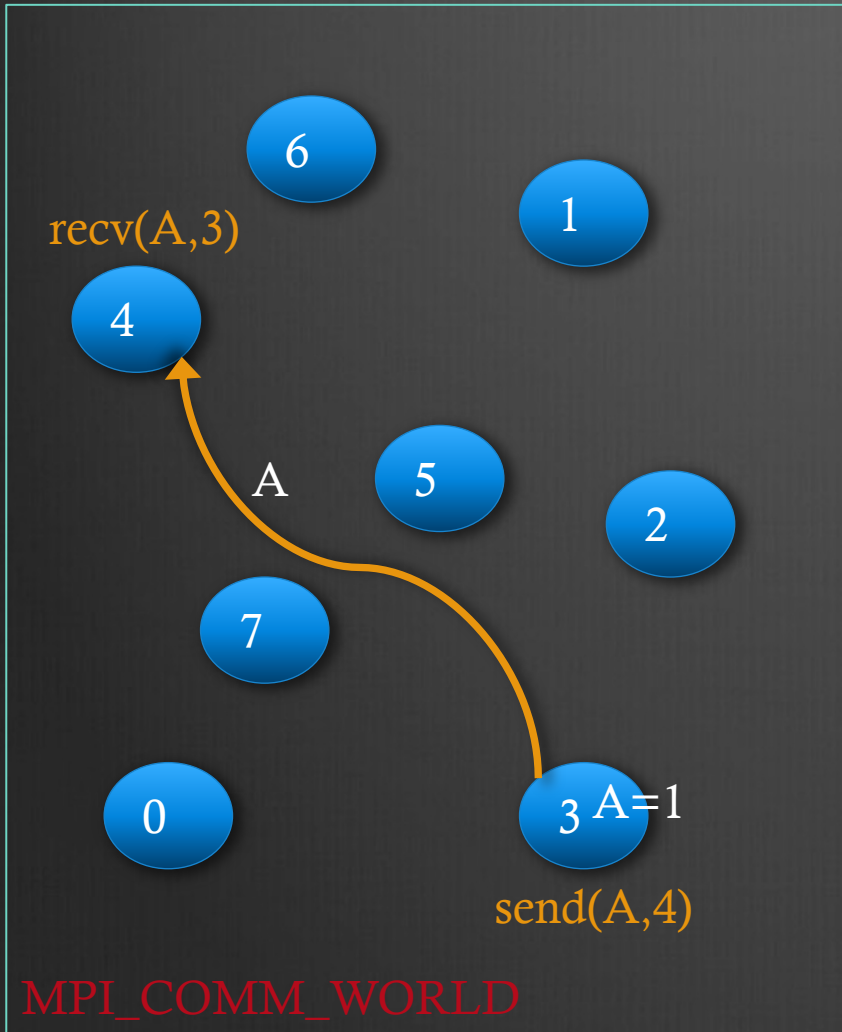
Implémentations Publiques

- ✧ MPICH2 : <http://mpich.org/>
- ✧ OpenMPI : <http://www.open-mpi.org/>

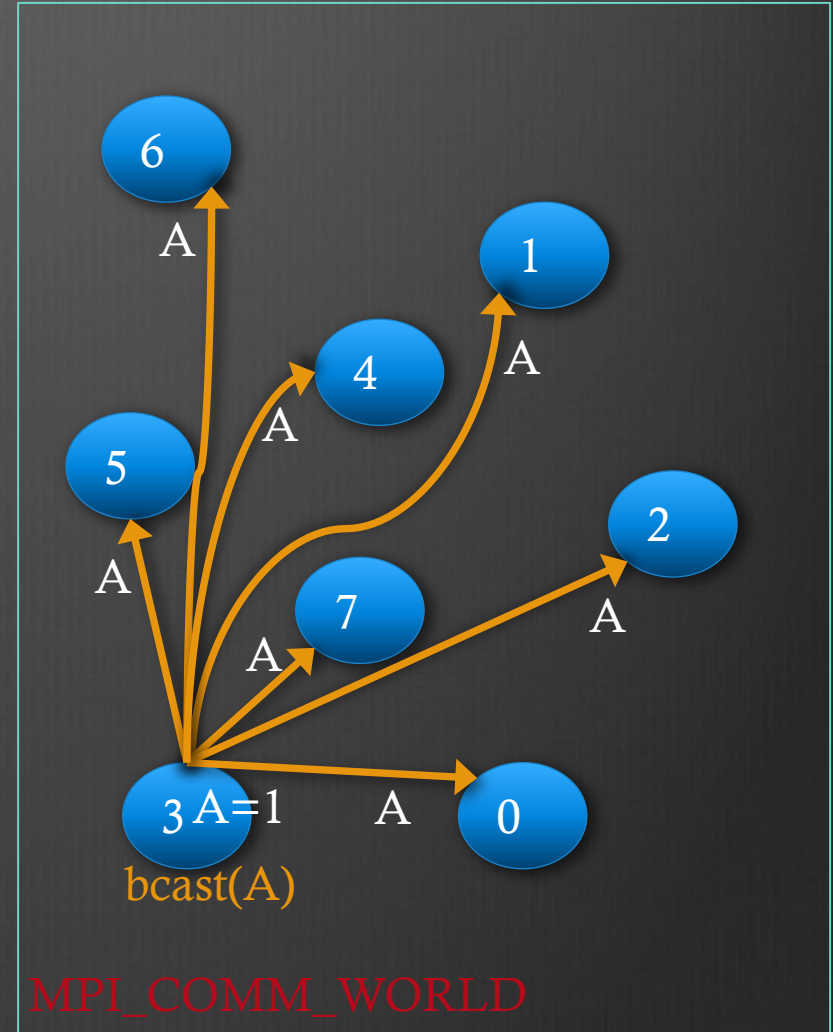
Principe de la parallélisation par échange de messages

- ✧ Le programme est répliqué sur N processus (**préférentiellement un processus par thread**).
- ✧ L'ensemble des processus appartient à un groupe par défaut désigné par **MPI_COMM_WORLD** (communicateur).
- ✧ Chaque processus est identifié par son rang (0, 1, 2, 3, ...).
- ✧ La mémoire allouée par chaque processus est privée.
- ✧ Une donnée privée à un processus doit être envoyée (**MPI_send**) pour être reçue (**MPI_recv**) par un autre processus (communication point à point).
- ✧ Une donnée privée peut être communiquée à un groupe de processus et inversement (communication collective : **MPI_bcast**, **MPI_scatter**, **MPI_gather**, **MPI_Allreduce**, etc.).

Communication point à point



Diffusion collective



pointapoint.f90

Program point_a_point

use **MPI**

implicit none

integer :: A, rang, tag=100, code

call **MPI_Init**(code)

call **MPI_Comm_rank**(**MPI_COMM_WORLD**, rang, code)

A=92290+rang

if (rang == 3) then

 A=1

 call **MPI_Send**(A, 1, **MPI_INTEGER**, 4, tag, **MPI_COMM_WORLD**, code)

else if (rang == 4) then

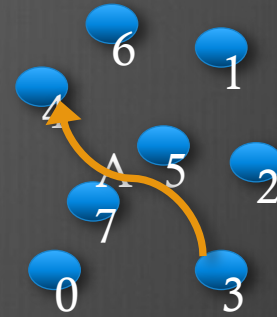
 call **MPI_Recv**(A, 1, **MPI_INTEGER**, 3, tag, **MPI_COMM_WORLD**, **MPI_STATUS_IGNORE**, code)

end if

print *,rang, A

call **MPI_Finalize**(code)

End program point_a_point



Compilation, exécution

```
mpif90 -o pointapoint.x pointapoint.f90
```

```
mpirun -np 8 pointapoint.x
```

Résultat

0	92290
1	92291
2	92292
5	92295
6	92296
7	92297
3	1
4	1

bcast.f90

Program bcast

use **MPI**

implicit none

integer :: A, rang, code

call **MPI_Init**(code)

call **MPI_Comm_rank**(**MPI_COMM_WORLD**, rang, code)

A=92290+rang

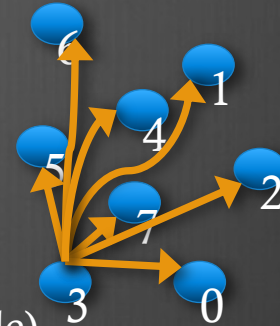
if (rang == 3) A=1

call **MPI_Bcast**(A, 1, **MPI_INTEGER**, 3, **MPI_COMM_WORLD**, code)

print *,rang, A

call **MPI_Finalize**(code)

End program bcast



Compilation, exécution

```
mpif90 -o bcast.x bcast.f90
```

```
mpirun -np 8 bcast.x
```

Résultat

7	1
3	1
0	1
1	1
4	1
5	1
6	1
2	1

Placement des processus

Géré par le système d'exploitation...

✧ En mode dédié :

- ✓ Un et un seul processus par *thread* de calcul.
- ✓ Autant de processus que de *threads* à disposition.
- ✓ Répartition des processus sur l'ensemble des *threads* d'un serveur avant de passer au serveur suivant.

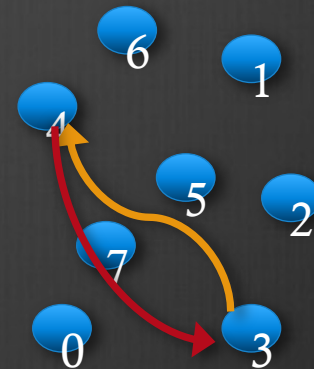
⇒ Placement optimal selon la topologie du réseau d'interconnexion.

✧ En mode non-dédié « temps partagé » :

- ✓ Plus de processus demandés que de *threads* à disposition.
- ✓ Plusieurs processus se partagent le temps sur un même thread.

⇒ Inefficace en mode production.

Communication point-à-point avec MPI_Sendrecv



sendrecv.f90

Program sendrecv

use **MPI**

implicit none

integer :: A, B, rang, tag=100, code

call **MPI_Init**(code)

call **MPI_Comm_rank**(**MPI_COMM_WORLD**, rang, code)

A=0 ; B=0

if (rang == 3) A=290

if (rang == 4) B=92

if (rang == 3) &

call **MPI_Sendrecv**(A, 1, **MPI_INTEGER**, 4, tag, B, 1, **MPI_INTEGER**, 4, tag, **MPI_COMM_WORLD**, &
MPI_STATUS_IGNORE, code)

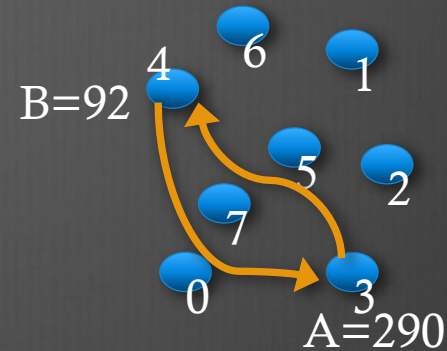
if (rang == 4) &

call **MPI_Sendrecv**(B, 1, **MPI_INTEGER**, 3, tag, A, 1, **MPI_INTEGER**, 3, tag, **MPI_COMM_WORLD**, &
MPI_STATUS_IGNORE, code)

print *,rang, A, B

call **MPI_Finalize**(code)

End program sendrecv



Compilation, exécution

```
mpif90 -o sendrecv.x sendrecv.f90  
mpirun -np 8 sendrecv.x
```

Résultat

0	0	0
1	0	0
2	0	0
5	0	0
6	0	0
7	0	0
4	290	92
3	290	92

Opération de réduction avec MPI_Reduce

- ✓ Par défaut, réalise en parallèle une opération commutative du type produit (\times), somme ($+$), min, max, etc.
- ✓ Met à contribution l'ensemble des processus d'un communicateur.
- ✓ Dépose le résultat final dans l'espace mémoire du processus désigné (**MPI_Reduce**) ou de l'ensemble des processus (**MPI_Allreduce**) d'un communicateur (groupe).

reduce.f90

Program reduce

```
use MPI
```

```
implicit none
```

```
integer :: A, B, rang, code
```

```
call MPI_Init(code)
```

```
call MPI_Comm_rank(MPI_COMM_WORLD, rang, code)
```

```
A = 92290+rang
```

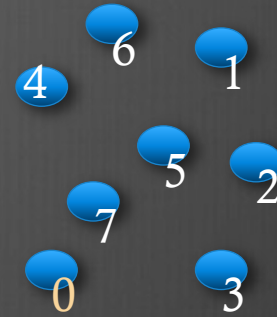
```
B = -1
```

```
call MPI_Reduce(A, B, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD, code)
```

```
print *,rang, A, B
```

```
call MPI_Finalize(code)
```

End program reduce



Compilation, exécution

```
mpif90 -o reduce.x reduce.f90
```

```
mpirun -np 8 reduce.x
```

Résultat

7	92297	-1
1	92291	-1
3	92293	-1
5	92295	-1
2	92292	-1
0	92290	92297
4	92294	-1
6	92296	-1

Allreduce.f90

Program Allreduce

use **MPI**

implicit none

integer :: A, B, rang, code

call **MPI_Init**(code)

call **MPI_Comm_rank**(**MPI_COMM_WORLD**, rang, code)

A = 92290+rang

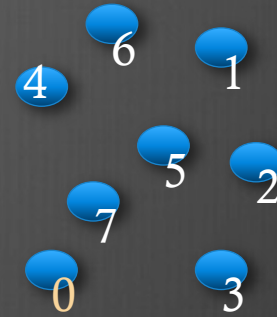
B = -1

call **MPI_Allreduce**(A, B, 1, **MPI_INTEGER**, **MPI_MAX**, **MPI_COMM_WORLD**, code)

print *,rang, A, B

call **MPI_Finalize**(code)

End program Allreduce



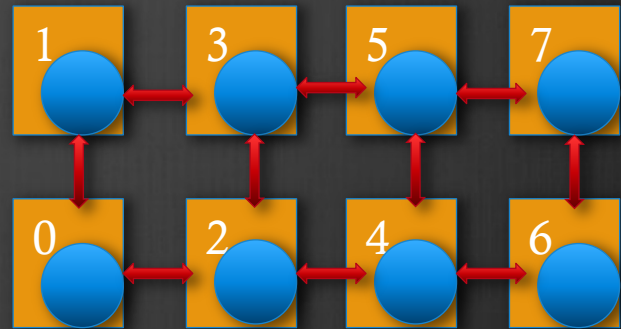
Compilation, exécution

```
mpif90 -o Allreduce.x Allreduce.f90
mpirun -np 8 Allreduce.x
```

Résultat

7	92297	92297
1	92291	92297
3	92293	92297
5	92295	92297
2	92292	92297
0	92290	92297
4	92294	92297
6	92296	92297

Topologie cartésienne (2D)



- ✓ Organiser les processus selon leur rang dans une grille cartésienne.
- ✓ Identifier le rang des processus voisins afin d'échanger des données.
- ✓ Extrêmement utile dans les méthodes de décomposition de domaine où chaque processus est affecté à un sous-domaine physique.

topo.f90

Program topo

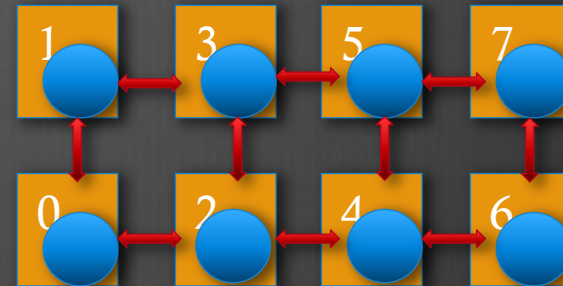
```
use MPI
implicit none
integer, parameter      :: Ndims = 2, OUEST=1, EST=2, SUD=3, NORD=4
integer, dimension(Ndims) :: dims  = [4, 2]
logical, dimension(Ndims) :: periods= [.FALSE., .FALSE.]
logical                 :: reorder = .TRUE.
integer, dimension(4)    :: voisin=MPI_PROC_NULL
integer                 :: rang, comm2D, code
```

```
call MPI_Init(code)
call MPI_comm_rank(MPI_COMM_WORLD, rang, code)
```

```
call MPI_Cart_create(MPI_COMM_WORLD, Ndims, dims, periods, reorder, comm2D, code)
call MPI_Cart_shift(comm2D, 0, 1, voisin(OUEST), voisin(EST), code)
call MPI_Cart_shift(comm2D, 1, 1, voisin(SUD), voisin(NORD), code)
```

```
print *, rang, voisin(:)
call MPI_Finalize(code)
```

End program topo



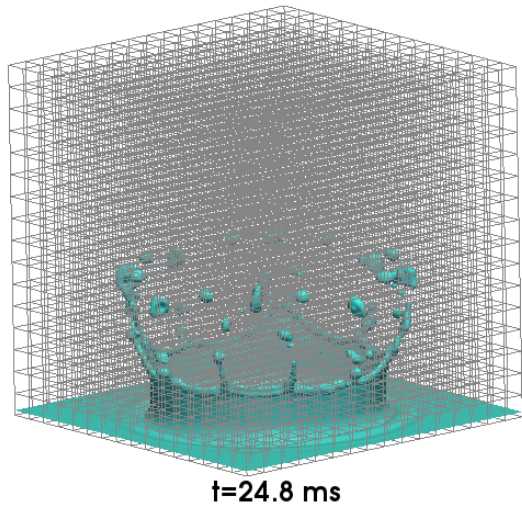
Compilation, exécution

```
mpif90 -o topo.x topo.f90
mpirun -np 8 topo.x
```

Résultat

5	3	7	4	-2
6	4	-2	-2	7
7	5	-2	6	-2
0	-2	2	-2	1
1	-2	3	0	-2
2	0	4	-2	3
3	1	5	2	-2
4	2	6	-2	5

Splash d'une goutte



Impact d'une goutte sur un film d'eau calculé sur 4096 threads de la machine BlueGene/Q de l'IDRIS (Centre de calcul du CNRS). $We=436$, $Fr=66$, $Oh=0.0016$, $\Delta=0.1$.
S. Shin, J. Chergui, D. Damir. <http://arxiv.org/abs/1410.8568>.

Communiquer dans une topologie cartésienne

```
...  
integer :: A, B, C, D, E, F, G, H, tag=100
```

```
...
```

```
A= 92290 + rang ; B=92290 - rang
```

```
C= 92 + rang ; D=92 - rang
```

```
call MPI_Sendrecv(A, 1, MPI_INTEGER, voisin(EST), tag, B, 1, MPI_INTEGER, voisin(OUEST),&  
tag, comm2D, MPI_STATUS_IGNORE, code)
```

```
call MPI_Sendrecv(C, 1, MPI_INTEGER, voisin(OUEST), tag, D, 1, MPI_INTEGER, voisin(EST),&  
tag, comm2D, MPI_STATUS_IGNORE, code)
```

```
...
```

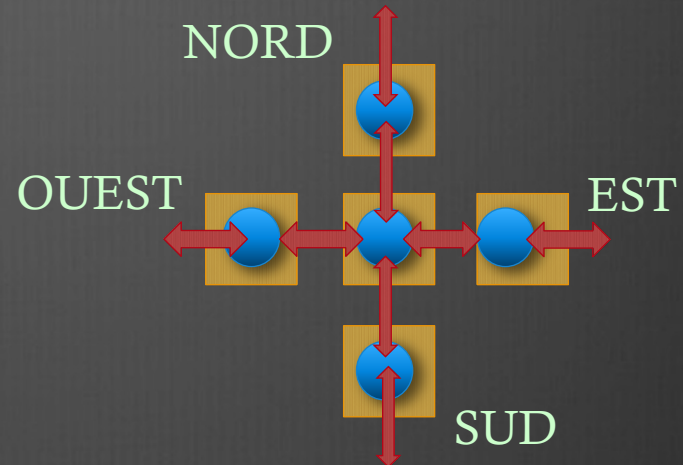
```
E= 2 * 92290+rang ; F=2*92290 - rang
```

```
G= 2*92 + rang ; H=2*92 -rang
```

```
call MPI_Sendrecv(E, 1, MPI_INTEGER, voisin(NORD), tag, F, 1, MPI_INTEGER, voisin(SUD),&  
tag, comm2D, MPI_STATUS_IGNORE, code)
```

```
call MPI_Sendrecv(G, 1, MPI_INTEGER, voisin(NORD), tag, H, 1, MPI_INTEGER, voisin(SUD),&  
tag, comm2D, MPI_STATUS_IGNORE, code)
```

```
...
```



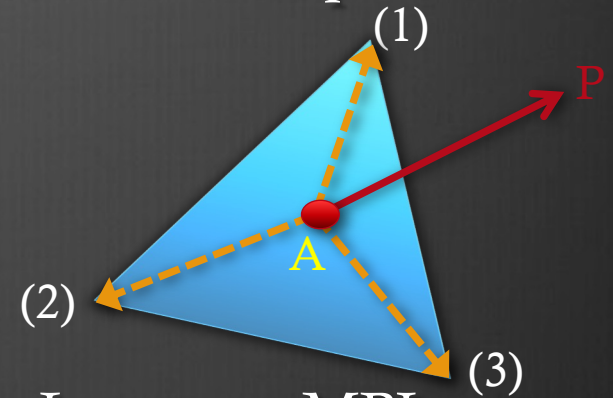
A découvrir ...

- ✓ Plusieurs protocoles de communications (bloquantes, non-bloquantes, copie mémoire).
- ✓ Types de données dérivées.
- ✓ Communications collectives (non-bloquantes).
- ✓ Topologie graphe.
- ✓ Constructeurs de communicateurs (groupes).
- ✓ Gestion dynamique des processus.
- ✓ Entrées/sorties parallèles.

Conclusion

Les **P**erformances d'une **A**pplication reposent sur trois piliers fondamentaux.

1. Architecture matériel.
2. Environnement logiciel (Compilateurs, Langages, MPI, OpenMP, Bibliothèques annexes, etc.).
3. Algorithmique (parallèle).



Application

Résolution parallèle de l'équation de la chaleur (1D)

1. Considérer l'équation de la chaleur munie de ses conditions limites (type Dirichlet) sur une tige de longueur L .
2. Discrétiser la dérivée spatiale par une méthode aux différences finies d'ordre (2).
3. Discrétiser la dérivée en temps à l'aide d'un schéma d'Euler retardé d'ordre (1).
4. Adopter un schéma explicite pour le calcul du terme de diffusion thermique.
5. Décomposer la tige de longueur L en sous-segments et résoudre en parallèle.

(1) Considérons l'équation de la chaleur munie de ses conditions initiale et aux limites sur une tige de longueur L .

$$\frac{\partial T}{\partial t} - \lambda \frac{\partial^2 T}{\partial x^2} = S(x)$$

$$T = T(x, t) ; x \in [0, L] ; t \geq 0$$

$$T(x, 0) = 0 ; x \in [0, L]$$

$$T(0, t) = T(L, t) = 0 ; t \geq 0$$

$$S(x) = \lambda \left(\frac{k\pi}{L} \right)^2 \sin\left(\frac{k\pi x}{L} \right)$$

$$\text{Solution: } T(x, t) = \sin\left(\frac{k\pi x}{L} \right) ; k = 1, 2, \dots, K$$

(2-4) Discrétisation du problème posé

$$T_i^{n+1} = \alpha T_{i-1}^n + (1 - 2\alpha)T_i^n + \alpha T_{i+1}^n + \delta t S_i^{n+1}$$

$$T_i^0 = 0.$$

$$T_1^n = T_N^n = 0.$$

où

$$\delta x = \frac{L}{N-1} ; \alpha = \lambda \frac{\delta t}{\delta x^2} \implies \delta t = \frac{\alpha \delta x^2}{\lambda} ; \alpha \leq 0.5$$

$$t^n = n\delta t ; n = 0, 1, 2, 3, \dots, N_t$$

$$x_i = i\delta x ; i = 0, 2, \dots, N - 1$$

(5) Décomposition du domaine et résolution en parallèle

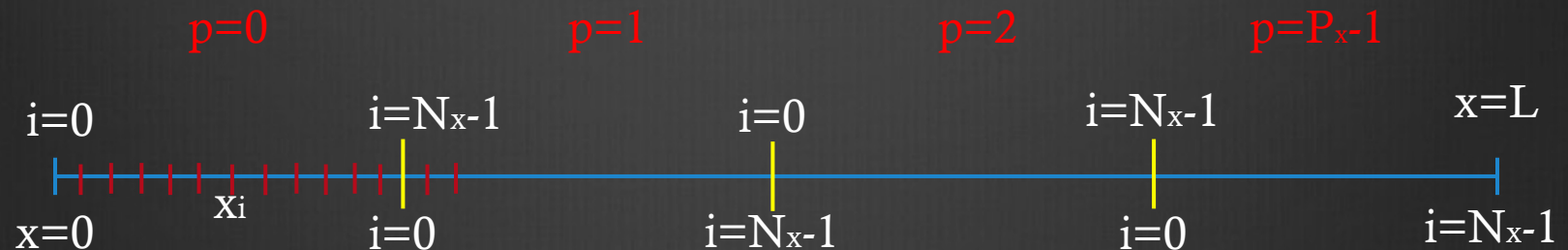
P_x : nombre de sous-segments.

$L_x = \frac{L}{P_x}$: longueur d'un sous-segment.

N_x : nombre de points du maillage par sous-segment.

$\delta x = \frac{L}{P_x(N_x-1)}$: pas d'espace du maillage.

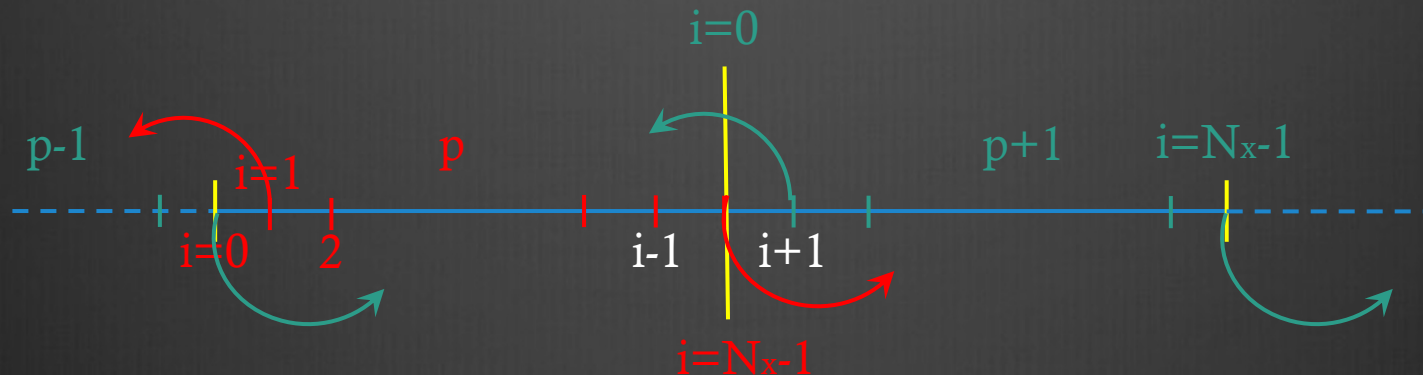
$x_i = pL_x + i\delta x$: coordonnée du nœud $i = 0, 1, \dots, N_x - 1$ du maillage sur le sous-segment $p = 0, 1, \dots, P_x - 1$.



Pour tout sous-segment $p = 0, 1, \dots, P_x - 1$:

$$T_i^{n+1} = \alpha T_{i-1}^n + (1 - 2\alpha)T_i^n + \alpha T_{i+1}^n + \delta t S_i^{n+1}$$

$$i = 1, 2, \dots, N_x - 1 ; n = 0, 1, \dots, N_t$$



Chaque sous-segment p **communiqué** ses valeurs $T_{i=1}^n$ et $T_{i=N_x-1}^n$ à ses voisins OUEST et EST.

Travaux pratiques

```

$ mpif90 -O3 chaleur.f90 -o chaleur.x
$ mpirun -np 4 ./chaleur.x
***
*** Nx      :      8
*** Px      :      4
*** N°dt    :    3580
*** dt      :  6.37755E-04
*** dx      :  3.57143E-02
*** Temps   :  2.28253E+00
*** Erreur  :  1.04973E-03
*** Increment : 9.93872E-13
***

```

```

$ ls *.dat
Tn_p=0_t=0.637117.dat  Tn_p=1_t=1.912628.dat
Tn_p=3_t=1.274872.dat  Tn_p=3_t=1.912628.dat
Tn_p=0_t=1.274872.dat  Tn_p=2_t=0.637117.dat
Tn_p=0_t=1.912628.dat  Tn_p=2_t=1.274872.dat
Tn_p=1_t=0.637117.dat  Tn_p=2_t=1.912628.dat
Tn_p=1_t=1.274872.dat  Tn_p=3_t=0.637117.dat
Ta_..., ..., E.dat

```

chaleur.f90

```

program chaleur
  use MODULE_CHALEUR

  implicit none
  integer :: n

  call topologie()
  call initialiser()
  TEMPS: do n= 1, NT

    call calculer()
    call communiquer()
    call imprimer()
    if (arreter()) exit TEMPS

  end do TEMPS
  call terminer()
end program chaleur

```



```

module MODULE_CHALEUR
  use MPI

  implicit none
  integer, parameter      :: Nx=8, Nt=10000, k=3
  integer, parameter      :: IMPRESSION=10, OUEST=1, EST=2
  real(kind=8), parameter :: L=1.0d0, alpha=0.5d0, lambda=1.0d0
  real(kind=8), parameter :: tolerance=2.D-13, pi=3.14159265359d0

  real(kind=8)            :: dx, dt, time=0.0d0
  real(kind=8), dimension(0:Nx) :: T, To, Ta, S, x
  real(kind=8), dimension(2)  :: erreur
  integer                  :: nstep=0, p, Px, comm
  integer, dimension(2)     :: voisin=MPI_PROC_NULL

  public :: topologie, initialiser, communiquer, calculer, arreter, imprimer, terminer
contains
  subroutine topologie()
    ...
  end subroutine topologie

  subroutine initialiser()
    ...
  end subroutine initialiser
  ...
end module MODULE_CHALEUR

```

```

program chaleur
  use MODULE_CHALEUR

  implicit none
  integer :: n

  call topologie()
  call initialiser()
  TEMPS: do n= 1, NT

    call calculer()
    call communiquer()
    call imprimer()
    if (arreter()) exit TEMPS

  end do TEMPS
  call terminer()
end program chaleur

```

subroutine topologie()

implicit none

integer, parameter :: ndims=1

integer, dimension(ndims) :: dims

logical :: reorder=.TRUE.

logical, dimension(ndims) :: periods=.FALSE.

integer :: code

call MPI_Init(code)

call MPI_Comm_size(MPI_COMM_WORLD, Px, code)

dims(:)=Px

call MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, &
periods, reorder, comm, code)

call MPI_Comm_rank(comm, p, code)

call MPI_Cart_shift(comm, 0, 1, voisin(OUEST), voisin(EST), code)

end subroutine topologie

```
program chaleur
use MODULE_CHALEUR
```

```
implicit none
integer :: n
```

```
call topologie()
```

```
call initialiser()
```

```
TEMPS: do n= 1, NT
```

```
call calculer()
```

```
call communiquer()
```

```
call imprimer()
```

```
if (arreter()) exit TEMPS
```

```
end do TEMPS
```

```
call terminer()
```

```
end program chaleur
```

subroutine initialiser()

implicit none

integer :: i

real(kind=8) :: Lx

$Lx = L / \text{real}(Px, \text{kind}=8)$ *! Longueur d'un sous-segment*

$dx = L / \text{real}(Px * (Nx - 1), \text{kind}=8)$ *! Pas d'espace*

$dt = \alpha * dx ** 2 / \text{lambd}$ *! Pas de temps*

! Coordonnee des noeuds de maillage

$x(0:Nx) = \text{real}(p, \text{kind}=8) * Lx + [(\text{real}(i, \text{kind}=8), i=0, Nx)] * dx$

! Condition Initiale

$T(0:Nx) = 0.0d0$; $To(0:Nx) = 0.0d0$

! Solution analytique

$Ta(0:Nx) = \sin(\text{real}(k, \text{kind}=8) * \pi * x(0:Nx) / L)$

! Terme source

$S(1:Nx-1) = \text{lambd} * (\text{real}(k, \text{kind}=8) * \pi / L) ** 2 * Ta(1:Nx-1)$

end subroutine initialiser

```
program chaleur
use MODULE_CHALEUR
```

```
implicit none
integer :: n
```

```
call topologie()
```

```
call initialiser()
```

```
TEMPS: do n= 1, NT
```

```
call calculer()
```

```
call communiquer()
```

```
call imprimer()
```

```
if (arreter()) exit TEMPS
```

```
end do TEMPS
```

```
call terminer()
```

```
end program chaleur
```

```
Subroutine calculer()
```

```
implicit none
```

```
! Conditions limites
```

```
if ( p == 0 ) T(0) =0.0d0
```

```
if ( p == Px-1 ) T(Nx-1)=0.0d0
```

```
T(1:Nx-1) = alpha * T(0:Nx-2) + (1.0d0 - 2.0d0 * alpha) * T(1:Nx-1) + &  
alpha * T(2:Nx) + dt * S(1:Nx-1)
```

```
if ( p == Px-1 ) T(Nx-1)=0.0d0
```

```
end subroutine calculer
```

```
program chaleur  
use MODULE_CHALEUR
```

```
implicit none  
integer :: n
```

```
call topologie()
```

```
call initialiser()
```

```
TEMPS: do n= 1, NT
```

```
call calculer()
```

```
call communiquer()
```

```
call imprimer()
```

```
if (arreter()) exit TEMPS
```

```
end do TEMPS
```

```
call terminer()
```

```
end program chaleur
```



```
subroutine communiquer()
```

```
implicit none
```

```
integer, parameter :: tag=100
```

```
integer           :: code
```

```
call MPI_Sendrecv(T(1), 1, MPI_DOUBLE_PRECISION, voisin(OUEST), tag, &  
                  T(Nx), 1, MPI_DOUBLE_PRECISION, voisin(EST), tag, &  
                  comm, MPI_STATUS_IGNORE, code)
```

```
call MPI_Sendrecv(T(Nx-1), 1, MPI_DOUBLE_PRECISION, voisin(EST), tag, &  
                  T(0), 1, MPI_DOUBLE_PRECISION, voisin(OUEST), tag, &  
                  comm, MPI_STATUS_IGNORE, code)
```

```
end subroutine communiquer
```

```
program chaleur  
use MODULE_CHALEUR
```

```
implicit none
```

```
integer :: n
```

```
call topologie()
```

```
call initialiser()
```

```
TEMPS: do n= 1, NT
```

```
call calculer()
```

```
call communiquer()
```

```
call imprimer()
```

```
if (arreter()) exit TEMPS
```

```
end do TEMPS
```

```
call terminer()
```

```
end program chaleur
```



```

logical function arreter() result(arret)
  implicit none
  integer :: code
  real(kind=8), dimension(2) :: erreur_local ! Erreur et increment

  erreur_local(1)= maxval(abs(T(0:Nx-1)-Ta(0:Nx-1)))
  erreur_local(2)= maxval(abs(T(0:Nx-1)-To(0:Nx-1)))

  call MPI_Allreduce(erreur_local, erreur, 2, MPI_DOUBLE_PRECISION, &
    MPI_MAX, comm, code)

  To(0:Nx-1) = T(0:Nx-1)

  arret=.false.
  if (erreur(2) <= tolerance) arret=.true.
end function arreter

```

```

program chaleur
  use MODULE_CHALEUR

  implicit none
  integer :: n

  call topologie()
  call initialiser()
  TEMPS: do n= 1, NT

    call calculer()
    call communiquer()
    call imprimer()
    if (arreter()) exit TEMPS

  end do TEMPS
  call terminer()
end program chaleur

```

```

subroutine terminer()
  implicit none
  integer          :: i, code
  character(len=80) :: pid, filename

  if (p == 0) then
    print '("****")'
    print '("**** ",a,1X,I8)',    "Nx      :",Nx
    print '("**** ",a,1X,I8)',    "Px      :",Px
    print '("**** ",a,1X,I8)',    "N°dt   :", nstep
    print '("**** ",a,1X,F10.6)', "dt      :", dt
    print '("**** ",a,1X,F10.6)', "dx      :", dx
    print '("**** ",a,1X,ES12.6)', "Temps   :", time
    print '("**** ",a,1X,ES12.3)', "Erreur  :", erreur(1)
    print '("**** ",a,1X,ES12.3)', "increment :", erreur(2)
    print '("****")'
  end if

  ! Ecriture solution analytique
  write(pid, '(I8)') p
  filename="Ta_p="//trim(adjustl(pid))//".dat"
  open(unit=9, file=trim(adjustl(filename)), FORM="formatted", &
        STATUS="replace", ACTION="write")
  write(9, '(2(ES12.5,1X))') (x(i), Ta(i), i=0, Nx-1)
  close(9)
  close(8)
  call MPI_Comm_free(comm, code)
  call MPI_Finalize(code)
end subroutine terminer

```

```

program chaleur
  use MODULE_CHALEUR

  implicit none
  integer :: n

  call topologie()
  call initialiser()
  TEMPS: do n= 1, NT

    call calculer()
    call communiquer()
    call imprimer()
    if (arreter()) exit TEMPS

  end do TEMPS
  call terminer()
end program chaleur

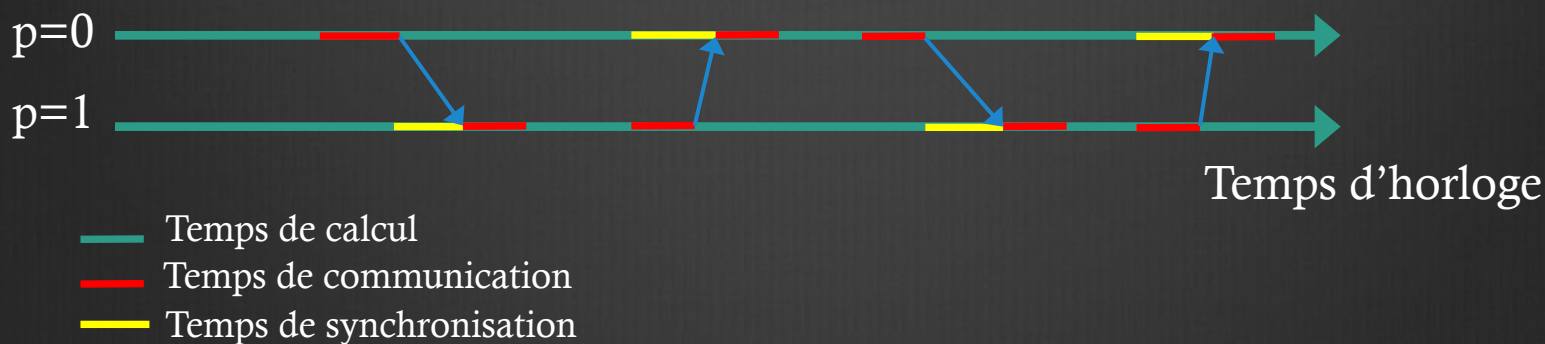
```

Mesure de performance

En cours d'exécution, un programme parallèle suit essentiellement une succession de 3 types d'évènements :

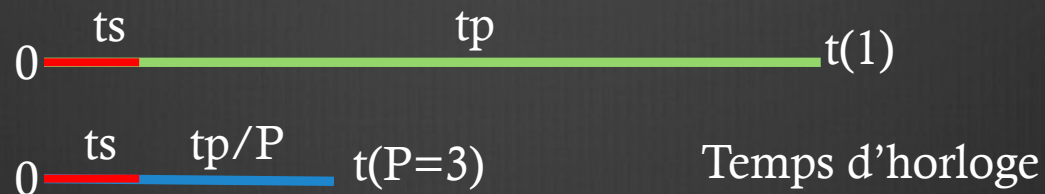
1. calcul (charge de travail utile),
2. synchronisation (attente d'un évènement),
3. communication (échange de données).

Les 2 derniers points pénalisent les performances parallèles quand leur coût cumulé devient relativement important par rapport à la charge de calcul.



Loi d'Amdahl (1967)

« Cette loi (optimiste) indique que si un programme s'exécute sur 1 *thread*, le temps $t(1)$ qu'il prendra sera la somme d'un temps t_s pour exécuter la partie non-parallèle et d'un temps t_p pour exécuter la partie parallélisable. Sur P threads, le temps d'exécution $t(P)$ sera au moins égale à $t_s + \frac{t_p}{P}$ »



Accélération et Efficacité

L'accélération A est définie par :

$$A(P) = \frac{t(1)}{t(P)} \in]0, P]$$

L'efficacité E est définie par :

$$E(P) = \frac{A(P)}{P} \in]0, 1]$$

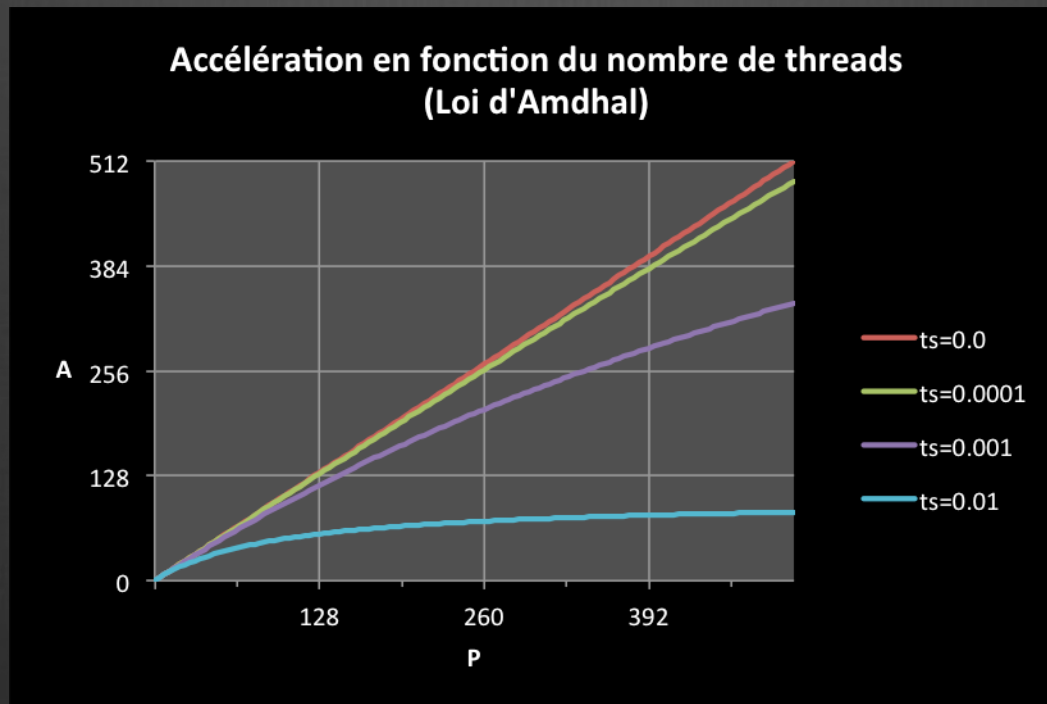
Selon la loi d'Amdahl et en fixant $t_s + t_p = 1$,

$$A(P) = \frac{1}{t_s(1 - \frac{1}{P}) + \frac{1}{P}}$$

l'accélération A est majorée ($P \rightarrow \infty$) par la fraction $\frac{1}{t_s}$;

l'accélération A est idéal ($A(P) = P$) quand $t_s = 0$.

L'accélération et l'efficacité expriment la qualité de la parallélisation du code et de son passage à l'échelle (extensibilité).



En pratique ...

- ✓ Mesurer le temps d'horloge avec la fonction `MPI_Wtime`.
- ✓ Mesurer le temps d'horloge $t(1)$ sur un *thread*.
- ✓ Mesurer le temps d'horloge $t(P)$ pour différentes valeurs de $P > 1$.
- ✓ Pour chaque P , calculer l'accélération $A(P) = \frac{t(1)}{t(P)}$
- ✓ Tracer les courbes $A(P)$ et $E(P)$.

chaleur.f90

```
program chaleur
  use MODULE_CHALEUR
  implicit none
  integer      :: n
  real(kind=8) :: t0, t1, tf

  call topologie()
  call initialiser()

  t0 = MPI_Wtime()
  TEMPS: do n= 1, NT
    call calculer()
    call communiquer()
    call imprimer()
    if (arreter()) exit TEMPS
  end do TEMPS
  t1 = MPI_Wtime() - t0
  call MPI_Allreduce(t1, tf, 1, MPI_DOUBLE_PRECISION, MPI_MAX, comm, code)
  if (p == 0) print *, "Px : ",Px, "; Temps : ", tf

  call terminer()
end program chaleur
```

Dépendance avec la charge de calcul

Quand cela est possible, mesurer le temps d'exécution :

- ✓ à charge de calcul constante par *thread* (**échelle faible**) ou
- ✓ à charge de calcul global constante (**échelle forte**).

Dans notre programme « chaleur », le nombre de points de maillage est constant par sous-segment (*thread*) => charge de calcul constante par *thread* => mesure selon une échelle faible.

Tant que le temps de calcul t_{calcul} par *thread* domine le temps lié aux communications t_{comm} ($\frac{t_{calcul}}{t_{comm}} \gg 1$), alors :

- ✧ selon l'échelle faible, le temps d'exécution du code devrait peu évoluer avec le nombre de threads P ;
- ✧ selon l'échelle forte, le temps d'exécution du code devrait décroître à nombre de threads P croissant.

Exercices

Reprenons le programme « chaleur » :

1. Fixer le nombre local de nœuds de maillage à $N_x=16$.
2. Exécuter le code avec $P_x=1, 2$ et 4 et mesurer le temps total.
3. Tracer les courbes $A(P_x)$ et $E(P_x)$ selon une échelle faible.
4. Justifier en mesurant le temps t_{calcul} et le temps t_{comm} .
5. Fixer $N_x=512$ et reprendre de (2) à (4).
6. Etendre la résolution du problème 1D à celle d'un problème 2D.