

Faculté  
des Sciences  
& Techniques



Université  
de Limoges

Master 1<sup>ère</sup> année



---

Parallélisme & Applications

—

P-F. Bonnefoi

*Version du 3 mars 2022*

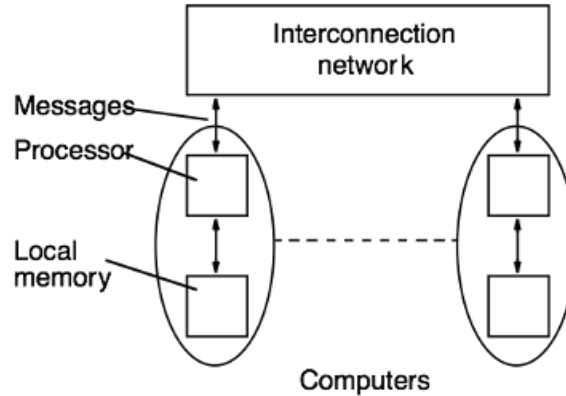
## Table des matières

1	L'architecture parallèle « <i>courante</i> » .....	3
2	Utilisation de bibliothèques « par passage de message » .....	6
3	Bibliothèque MPI : qu'est-ce qu'un message ? .....	10
4	Bibliothèque MPI : les primitives de communication .....	11
5	Bibliothèque MPI : les communications globales .....	19
6	MPI : compilation & Exécution .....	23
7	Des bibliothèques spécialisées .....	28



## Le réseau de station de travail, « NOW »

Un ensemble de machines autonomes connectées par un réseau d'interconnexion qui communiquent entre eux par « échange de messages ».

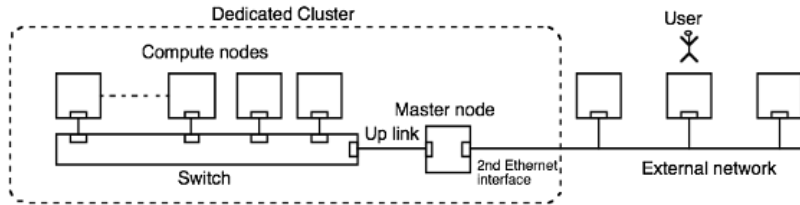


Ce modèle est caractérisé par :

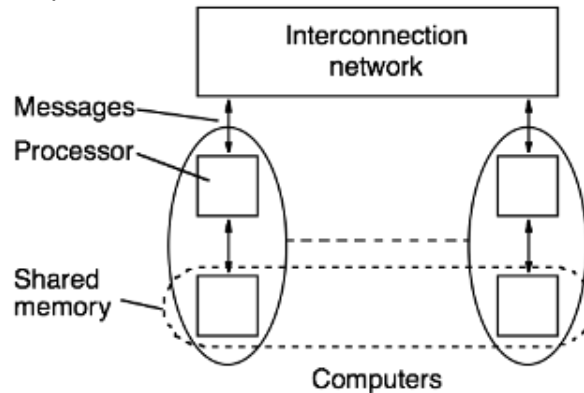
- ❑ une topographie réelle : le réseau d'interconnexion ;
- ❑ l'absence de synchronisation aux niveaux des instructions exécutées sur chaque processeur (machines indépendantes) ;
- ❑ l'organisation des échanges entre processeur :
  - ◇ les messages contrôlent le programme parallèle ;
  - ◇ le chemin des messages définissent la topographie de la machine parallèle (topographie virtuelle) !



## Organisation



Par exemple, il est possible à l'aide d'échanges de messages de simuler une machine à mémoire partagée au-dessus de la machine parallèle.



*Cela permet de simplifier la programmation en ne tenant pas compte des messages à échanger entre les différents processeurs, mais cela peut entraîner des surcoûts parce que le programmeur n'est pas informé des effets de sa programmation (exemple : accès par ligne à une matrice « réellement » répartie par colonnes entre les processeurs).*



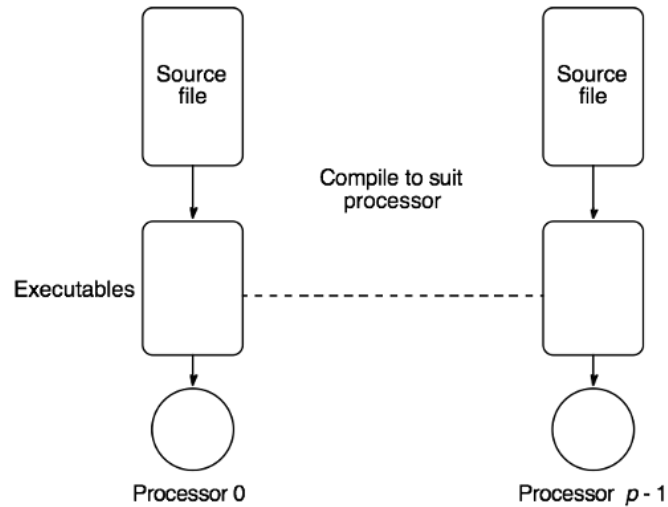
# Modèle du «*Message Passing*»



### Mécanismes proposés

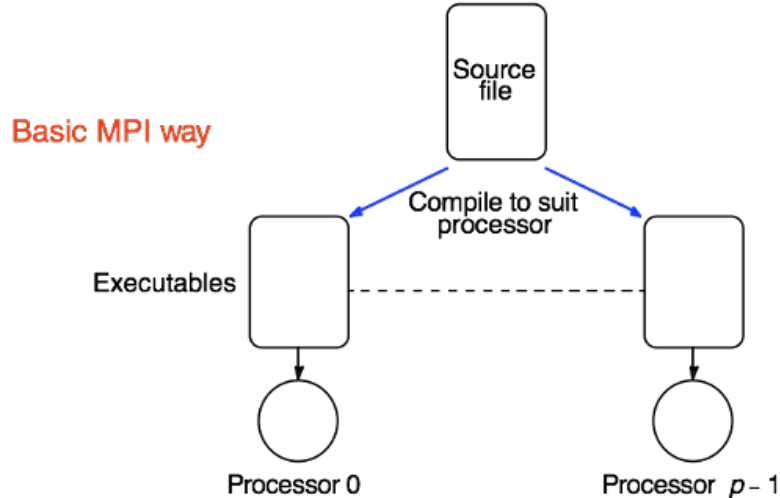
- possibilité de créer de processus séparés sur différents ordinateurs ;
- possibilité d'envoyer et de recevoir des messages.

Cela correspond au modèle «*Multiple program, multiple data (MPMD)*»



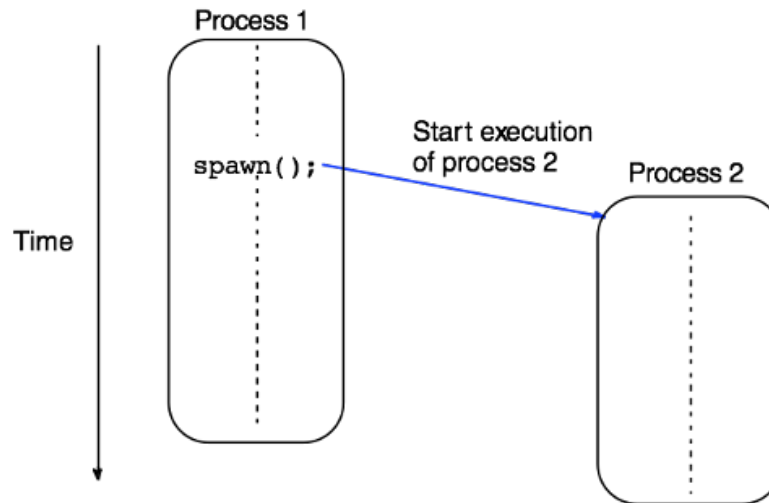
## Modèle SPMD

- différents processus se « mélangent » en un seul programme ;
- les instructions de contrôle vont exécuter différentes parties des programmes présents sur chaque processeur.
- tous les exécutables démarrent en même temps : on parle de création statique de processus



## Modèle MPMD

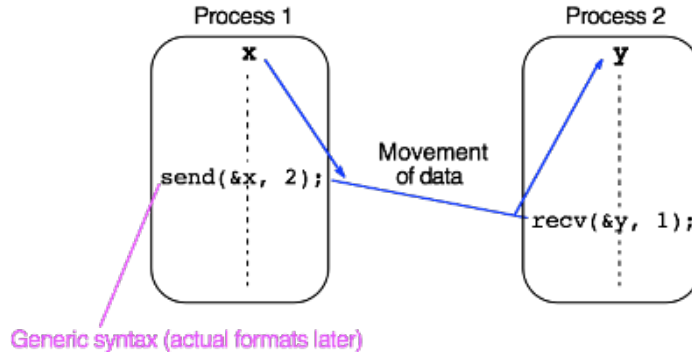
- ▷ des programmes séparés sur chaque processeur ;
- ▷ un processeur exécute le processus « maître » ;
- ▷ le processus maître exécute d'autres processus : on parle de création dynamique de processus.



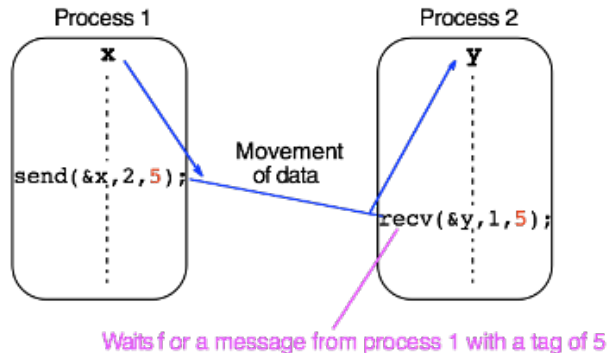


## Passage de messages

- un processus envoie un message contenant une donnée vers un autre processus ;



- le message peut contenir un étiquette ou « tag » pour différencier les messages (ne contiennent que des données).



### 3 Bibliothèque MPI : qu'est-ce qu'un message ?

Un message MPI est un tableau d'éléments d'un certain *type* :



Où le *type*  
est :

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED.	

Le *type* des données doit correspondre entre envoi et réception.



### Les communications entre processus

- Communications **point à point** entre deux processeurs ;
- L'envoi de message nécessite 2 opérations :
  - ◇ émission de l'expéditeur
  - ◇ réception du destinataire

*Suivant le modèle de communication, le récepteur qui refuse de recevoir message peut bloquer l'expéditeur ou non.*

### Pour l'envoi

Forme: `MPI_SEND(&buffer, count, datatype, destination, tag, comm)`

- **buffer** pointeur en mémoire qui indique où se trouve le premier octet de l'info à envoyer ;
- **count** entier indiquant la taille du message en nombre d'éléments de type `datatype` ;
- **datatype** type d'un élément du message.
- **destination** identifiant du destinataire ;
- **tag** étiquette du message *Il permet de différencier les messages au sein de l'application*
- **comm** : indique le *communicator* à utiliser.

### Pour la réception

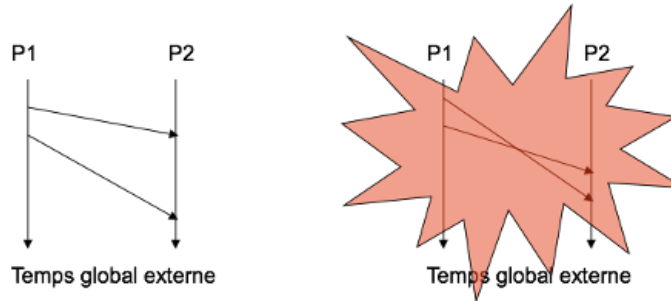
Forme: `MPI_RECV(&buffer, count, datatype, source, tag, comm, &status)`

- **buffer** pointeur en mémoire qui indique où devra être déposé le premier octet de l'info
- **count** entier indiquant la taille du message en nombre d'éléments de type `datatype` ;
- **datatype** type d'un élément du message ;
- **source** identifiant de la source ;
- **tag** étiquette du message ; *Elle permet de différencier les messages au sein de l'application*
- **comm** : communicator à utiliser ;
- **status** : pointeur vers une structure `MPI_Status` contenant des infos sur le message reçu (source, tag, error, count, cancelled). *On peut utiliser `MPI_STATUS_IGNORE` si on n'en a pas besoin.*



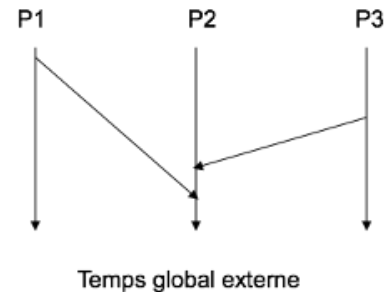
## Les communications entre processus

- chaque paire de processeurs est reliée par un lien ;  
*Ce lien « logique » peut être réalisé par plusieurs liens physiques et switches*
- les liens sont **fiables** : pas de perte de message
- les liens garantissent l'ordre FIFO, «*First In First Out*» ;



## Problème ? Absence de temps global

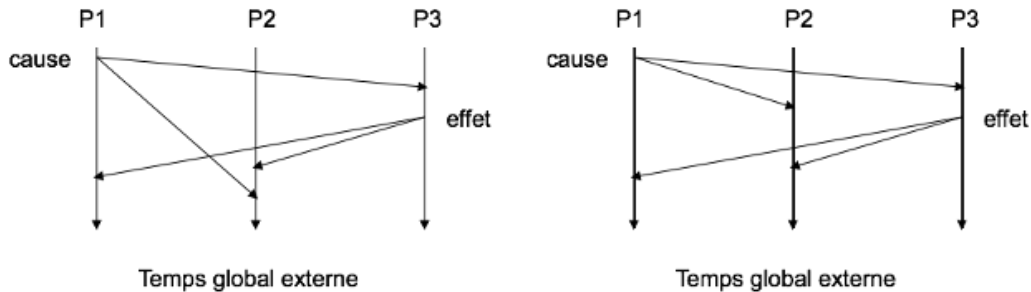
On ne peut pas garantir que deux messages envoyés par des processeurs distincts arrivent dans un ordre choisi. *si le message M1 est plus ancien que M2, ils ont nécessairement été envoyés dans cet ordre seulement s'ils proviennent du même processus.*



## L'ordre causal n'est pas respecté dans une machine parallèle

- différence des distances
- différence des disponibilités des liens ...

### Caractérisation : plusieurs possibilités



### Conséquence

Le non respect de l'ordre causal se traduit par du **non déterminisme**.

Ce non déterminisme doit être pris en compte :

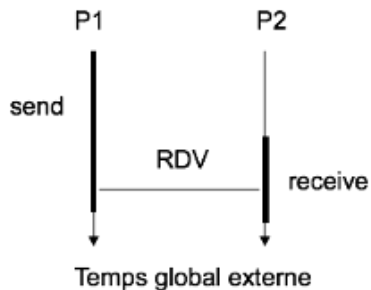
- Lors de l'écriture et de l'utilisation des primitives d'envoi et de réception ;
- Lors de la conception d'outils d'analyse de performance et/ou de débogage qui peuvent modifier l'ordre des messages en effectuant la collecte de traces d'exécution et des échanges de messages.



## Communication synchrone ou par rendez-vous

- ▷ les deux processus P1 et P2 sont prêts : P1 à émettre, P2 à recevoir ;
- ▷ P1 attend P2.

Évaluation : le temps d'attente peut être long, mais l'expéditeur est sûr que le destinataire a reçu le message



**Bloquant :** la fonction ne retourne que lorsque l'échange a fini

- ▷ côté réception : les données sont arrivées et prêtes à être traitées ;
- ▷ côté envoi : le buffer d'envoi peut être réutilisé sans créer d'erreurs ;

## Communication asynchrone

- l'expéditeur envoie son message dès qu'il est prêt ;
- la réception peut se faire plus tard ;

**Avantages :**

- l'émetteur n'attend jamais ;
- le récepteur peut éventuellement attendre si le message n'est pas encore arrivé.

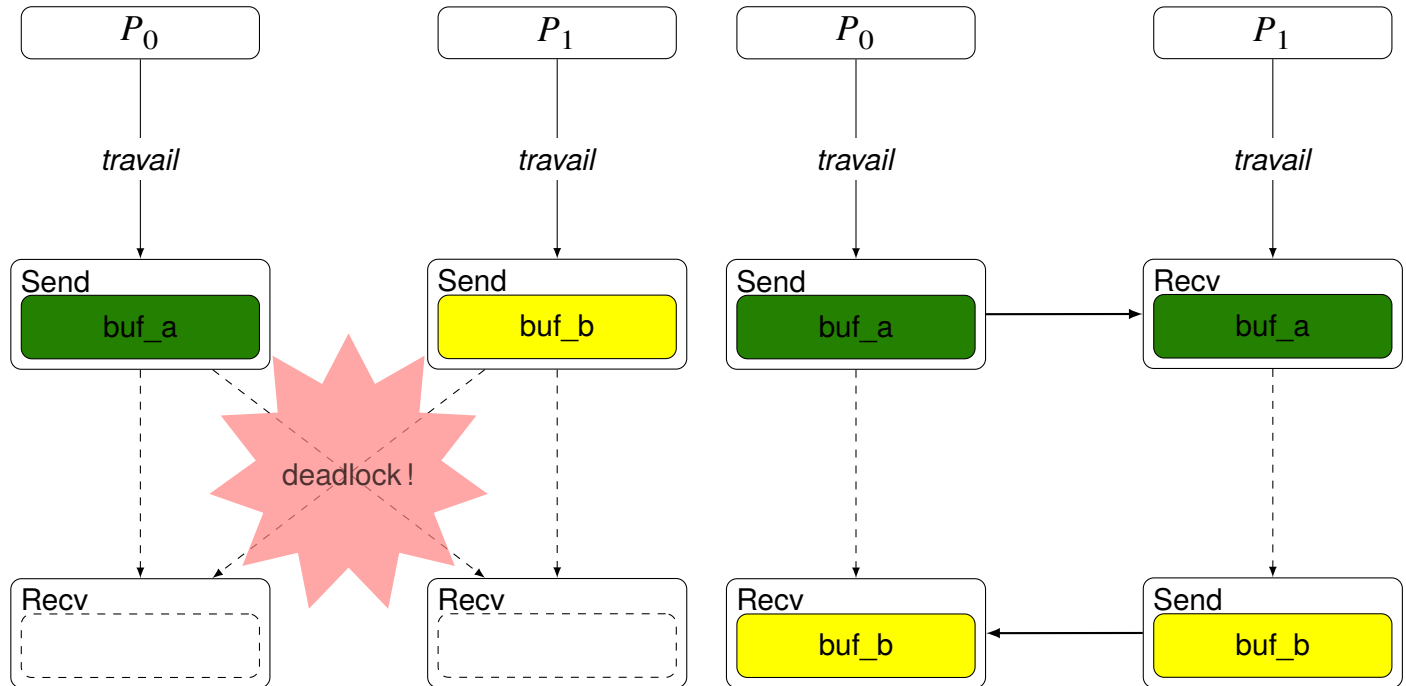
**Inconvénient :**

- moins de contrôle sur la réception.

*Il est facile de simuler du synchrone avec des communications asynchrones à l'aide d'un système d'accusé-réception.*



## L'utilisation d'appels bloquants peut conduire à un deadlock



### Deadlock :

- ▷ un processus attend un message qui n'arrivera jamais ;
- ▷ le seul moyen de terminer le programme MPI et de l'arrêter brutalement (ctrl-c/kill) ;
- ▷ **peut ne pas toujours bloquer** : cela dépend de la taille des buffers d'envoi.



## Et l'indéterminisme dans tout ça ?

Pour tenir compte de l'indéterminisme, il faut tenir compte de :

- l'émission doit indiquer le destinataire et le type de message ;

*C'est la primitive de réception qui détermine donc l'ordre dans lequel les messages doivent être traités*

- les réceptions sont indépendantes de l'ordre d'arrivée des messages.

On obtient ainsi une exécution déterministe.

*Attention au niveau des algorithmes qui n'imposent pas d'expéditeurs ou de type de messages.*

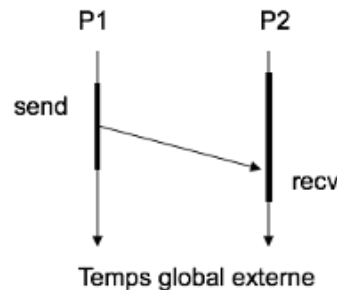
## Modèle asynchrone bloquant

- l'appel à la primitive d'envoi se termine lorsque le message a quitté l'expéditeur

Mais, cela ne signifie pas que le destinataire l'ait bien reçu.

- l'appel à la primitive de réception se termine lorsque le message est :

- ◇ arrivé ;
- ◇ copié dans le buffer de réception.

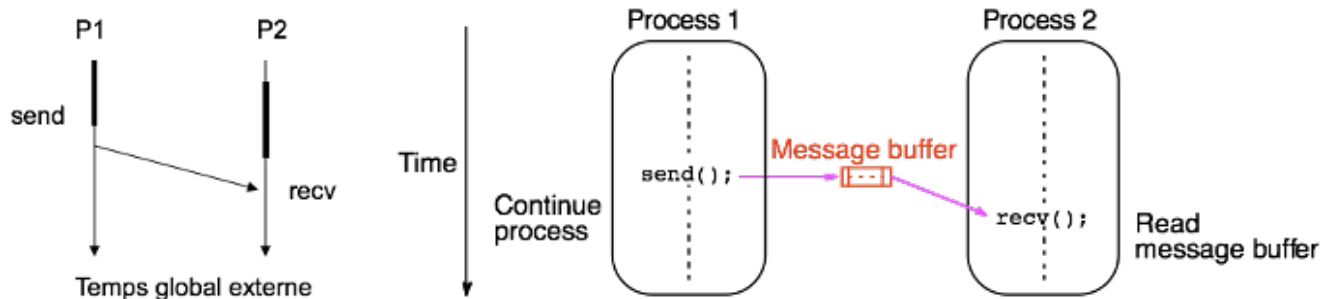




## Modèle asynchrone non bloquant

On demande l'exécution d'une communication sans attendre qu'elle soit finie.

- la couche logiciel expédie ou reçoit dès que possible ;
- recouvrement Calcul/Communication :  
 permet de **masquer le temps des communications** : des calculs sont effectués pendant les communications.



## Contraintes

Le programmeur doit veiller à ce que les opérations soient terminées en temps voulu :

- une primitive `msgwait` qui bloque l'exécution jusqu'à ce qu'une communication soit terminée ;
- une primitive `msgdone` qui permet de savoir si la communication a été effectuée.

Les primitives `msgwait` et `msgdone` se réfèrent à un numéro de transaction retourné par les primitives `send` et `recv`.

## Avantage

Il est possible de faire du calcul et parallèlement des communications, mais il faut alors **gérer** la terminaison de ces communications pour **effectuer des calculs** sur les **bonnes données** !



**Exemple : Opération sur un vecteur de taille  $n$** 

- ▷ `blsrecv/blsend` : réception et envoi bloquant (en MPI : `MPI_Recv` et `MPI_Send`);
- ▷ `nbsend/nbrecev` : réception et envoi non bloquant (en MPI : `MPI_Irecv` et `MPI_Isend`);
- ▷ `msgwait` : attente de l'envoi/réception effective (en MPI : `MPI_Wait`).

**Version Bloquante**

1	<b>pour</b> $j = 1$ à $n$ <b>faire</b>
2	<b>blrecv</b> ( $P_{i-1}$ , donnée)
3	résultat=calcul (donnée)
4	<b>bsend</b> ( $P_{i+1}$ , résultat)

Dans la version **non bloquante**, le résultat2 peut être calculé pendant que la communication est réalisée.

⇒ **recouvrement calcul/communication.**

**Version non bloquante**

1	<b>blrecv</b> (donnée1)
2	<code>id_recv</code> = <b>nbrecev</b> (donnée2)
3	résultat1=calcul (donnée1)
4	<b>msgwait</b> ( <code>id_recv</code> )
5	donnée1 = donnée2
6	
7	<b>pour</b> $j = 2$ à $n-1$ <b>faire</b>
8	<code>id_recv</code> = <b>nbrecev</b> (donnée2)
9	<code>id_send</code> = <b>nbsend</b> (résultat1)
10	résultat2 = calcul (donnée1)
11	<b>msgwait</b> ( <code>id_recv</code> )
12	<b>msgwait</b> ( <code>id_send</code> )
13	donnée1 = donnée2
14	résultat1 = résultat2
15	
16	<code>id_send</code> = <b>nbsend</b> (résultat1)
17	résultat2 = calcul (donnée1)
18	<b>msgwait</b> ( <code>id_send</code> )
19	<b>bsend</b> (résultat2)



Ce sont des communications :

- qui concernent l'**ensemble** ou un **sous-ensemble** des processus d'une application parallèle ;
- qui reprennent des **schémas** de communication **souvent employés**.

Fonctionnement général d'une communication globale :

- l'ensemble des processus réalise une séquence d'instructions prédéterminées ;
- ils se synchronisent avant d'exécuter l'algorithme ;
- ils se synchronisent à la fin ;

ces synchronisations permettent de voir la communication globale comme une opération **atomique** et **déterministe**.

### Différents types de communication globale

- la **barrière de synchronisation** : permet de synchroniser un ensemble de processus ;
- la **diffusion** ou broadcast : un processus envoie un même message à l'ensemble des autres ;
- la **distribution** : un processus donné envoie un message distinct à tous les autres ;
- le **rassemblement** : opération inverse de la distribution ;
- la **transposition** : chaque processus effectue simultanément une distribution ;
- le **comméragé** : chacun des  $n$  processus possède une information spécifique, à la fin de l'algorithme, tous les processus connaissent les  $n$  informations.



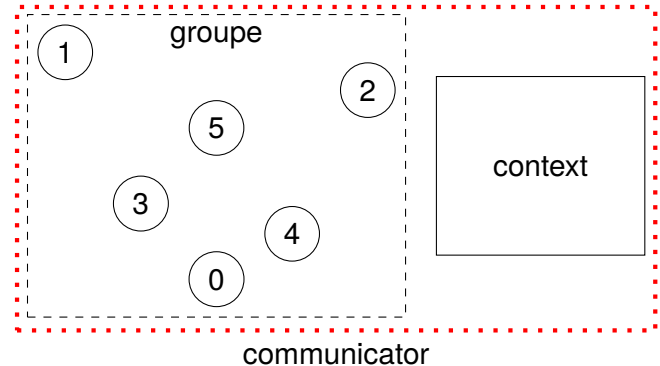
# Comment organiser les nœuds dans une machine parallèle avec MPI ? 20

Un nœud doit :

- ▷ rejoindre l'application parallèle : c'est une machine indépendante au lancement ;
- ▷ connaître les membres de son groupe et leur nombre : pour pouvoir organiser le travail parallèle ;
- ▷ son rang dans le groupe : pour pouvoir s'identifier, déterminer son rôle et communiquer avec les autres nœuds.

## Groupe, «Communicator» et «context»

- **Groupe** : un ensemble fixe de  $k$  nœuds numérotés de 0 à  $k - 1$
- «**Communicator**» : spécifier le «scope» ou la portée d'une communication :
  - ◇ entre les nœuds dans un groupe (intra) ;
  - ◇ entre deux groupes disjoints (inter) ;*Par défaut c'est `MPI_COMM_WORLD`*
- «**Context**» : partition de l'espace de communication :
  - ▷ un message envoyé dans un contexte ne peut pas être reçu dans un autre contexte : utile pour des bibliothèques indépendantes utilisant des étiquettes qui leur sont propres.



MPI_INIT	rejoindre la machine parallèle
MPI_FINALIZE	quitter la machine parallèle
MPI_COMM_SIZE	obtenir la taille du groupe
MPI_COMM_RANK	obtenir son rang dans le groupe
MPI_SEND	envoyer un message
MPI_RECV	recevoir un message

Le «hello world» :

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```

*Le printf sera local au nœud.*

Connaitre la machine parallèle :

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    /* rank of this process in the communicator */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* get the size of the group associates to the communicator */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```



## Les communications

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    /* Find out rank, size */
    int world_rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int number;

    if (world_rank == 0)
        number = -1;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (world_rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }
}
```

Diagram annotations:

- nbre élément**: points to the value `1` in `MPI_Send`.
- rang dest.**: points to the value `1` in `MPI_Send`.
- étiquette/tag**: points to the value `MPI_INT` in `MPI_Send`.
- communicator par défaut**: points to the value `MPI_COMM_WORLD` in `MPI_Send`.
- rang source**: points to the value `0` in `MPI_Recv`.
- status**: points to the value `MPI_STATUS_IGNORE` in `MPI_Recv`.



**Compilation**

```
xterm
$ mpicc -o hello_world_mpi hello_world_mpi.c
```

**Exécution**

```
xterm
$ mpirun -np 1 ./hello_world_mpi
```

*lancement du travail MPI*

*exécutable*

*nombre de processus*

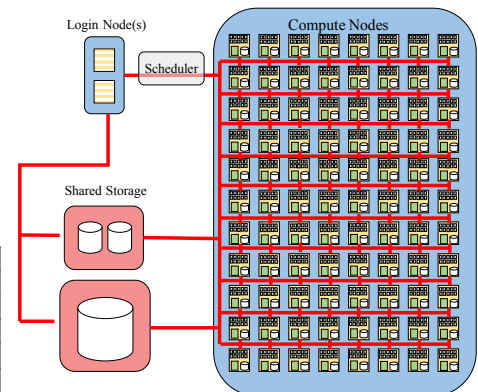
**Utilisation du scheduler SLURM**

- ▷ indiquer les ressources nécessaires et le temps d'utilisation ;
- ▷ donner le travail :
  - ◊ avec `srun` : lancer une application MPI sur un cluster SLURM ;
  - ◊ il exécute `mpirun -np n` mais en tenant compte de l'occupation du cluster.

Le script shell pour SLURM :

```
#!/bin/bash
#SBATCH --ntasks 4 # 4 mpi tasks
#SBATCH -t 00:05:00 # Time in HH:MM:SS

#Launch job with srun not mpirun/mpiexec!
srun ./hello_world_mpi
```



MPI_Init	Initialize MPI
MPI_Finalize	Clean up MPI
MPI_Comm_size	Get size of MPI communicator
MPI_Comm_Rank	Get rank of MPI Communicator
MPI_Reduce	Min Max Sum <i>etc</i>
MPI_Bcast	Send message to everyone
MPI_Allreduce	Reduce but store result everywhere
MPI_Barrier	Synchronize all tasks by blocking
MPI_Send	Send a message (blocking)
MPI_Recv	Receive a message (blocking)
MPI_Isend	Send a message (non-blocking)
MPI_Irecv	Receive a message (non-blocking)
MPI_Wait	Blocks until message is completed
MPI_Wtime	Donne le temps courant

Exemple de mesure du temps d'exécution :

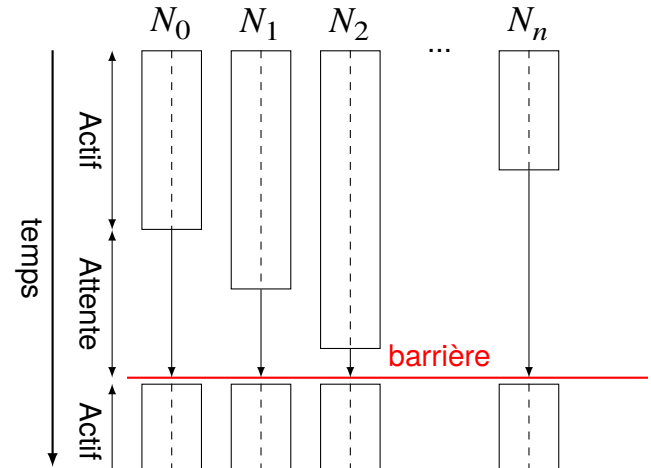
```
double t1 = MPI_Wtime();  
  
//do something expensive...  
  
double t2 = MPI_Wtime();  
  
if (my_rank == 0) {  
    printf("Total runtime = %g s\n",  
          (t2-t1));  
}
```





`MPI_Barrier` : réalise une synchronisation de tous les nœuds.

`MPI_Barrier(comm)`



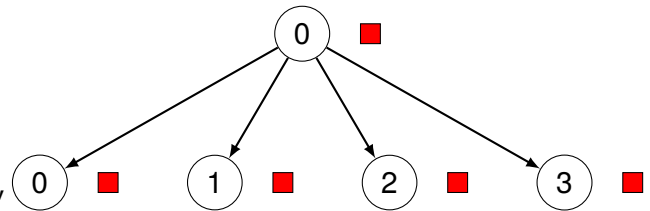
## Synchronisation des processus :

- ▷ instruction forcément bloquante ;
- ▷ tous les processus sont forcés de s'attendre mutuellement ;
- ▷ **À utiliser uniquement si nécessaire** ⇒ **réduit le parallélisme !**



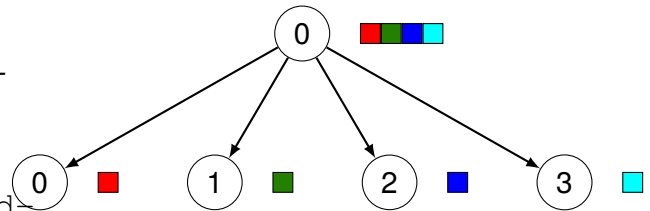
`MPI_Bcast`: envoie la même donnée vers tous les processus du groupe, y compris lui-même.

`MPI_Bcast(&buffer, count, datatype, root, comm)`



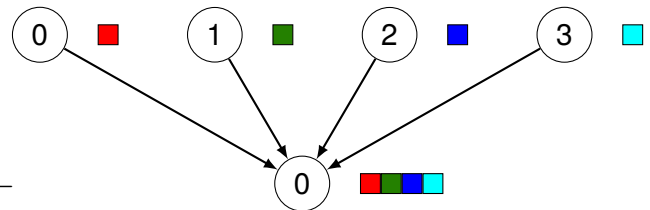
`MPI_Scatter`: envoie différentes données d'un tableau vers différents nœuds, y compris lui-même.

`MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`



`MPI_Gather`: récupère différentes données depuis différents nœuds dans un tableau, y compris lui-même.

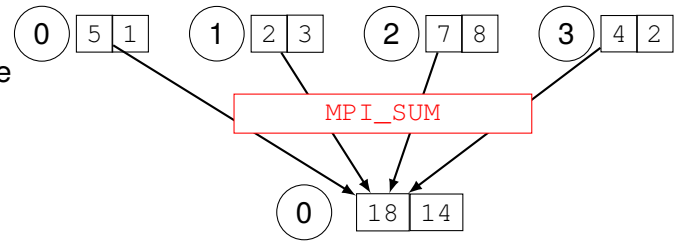
`MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`



`MPI_Reduce`: Prends un tableau de données en entrée sur chaque nœud et retourne un tableau de données en sortie dans le nœud racine en réalisant l'opération indiquée.

*Ici, on réalise une somme.*

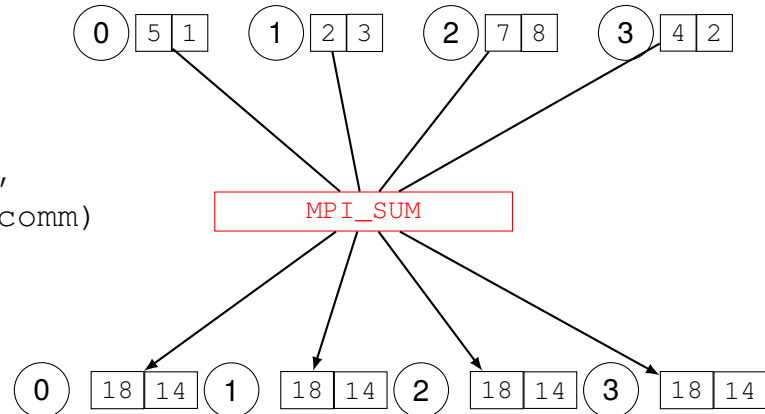
⇒ opération de **réduction**



```
MPI_Reduce(&sendbuf, &recvbuf, count,  
datatype, mpi_operation, root, comm)
```

`MPI_Allreduce`: Similaire à `MPI_Reduce` mais distribue le résultat à tous les nœuds.

```
MPI_Allreduce(&sendbuf, &recvbuf,  
count, datatype, mpi_operation, comm)
```



Une des difficultés d'utilisation des machines MIMD résidait dans le manque de logiciels adaptés.

- Solution :**
- ▷ des **bibliothèques spécialisées** de calcul disponible pour des machines parallèles de tout genre (NOW, machine pipeline, vectoriel *etc.*)
    - ▷ portabilité ;
    - ▷ efficacité ;
  - ▷ certains **algorithmes fondamentaux** de calcul reviennent régulièrement dans les programmes de calcul scientifique ;
  - ▷ routines de base optimisées permettant de construire des programmes plus simples, plus portables et plus efficaces.

### Exemples pour le calcul scientifique

- **BLAS**, BLAS Parallèles : «*Basic Linear Algebra Subroutines*»
  - ◇ BLAS de niveau 1 (les premières de l'algèbre linéaire) :
    - \* opérations de base sur les vecteurs
    - \* mise à jour et produit scalaire sur des vecteurs de même dimension
  - ◇ BLAS de niveau 2 : opérations entre matrices et vecteurs
  - ◇ BLAS de niveau 3 : opérations matrice-matrice
- **LAPACK** : «*Linear Algebra PACKage*»
  - ◇ résolution de systèmes linéaires, résolution des moindres carrés, valeur propre, valeur singulière factorisation de matrice LU...
  - ◇ conçue pour les machines vectorielles et les machines parallèles à mémoire partagée
  - ◇ LAPACK utilise les BLAS de niveau 3
- **ScaLAPACK** : Bibliothèque LAPACK pour machine parallèle à mémoire distribuée (scalable)
  - ◇ BLACS pour les communications PBLAS pour les calculs
  - ◇ PBLAS : BLAS parallèles, travaille sur des matrices distribuées selon un schéma cyclique par bloc



## Pour les communications

- base de la programmation des machines parallèles ;
- amélioration des performances ;
- confort d'utilisation (les premières machines à mémoire distribuée étaient livrées avec des primitives de trop bas niveau et souvent uniquement bloquantes).

## Exemples

**BLACS (Basic Linear Algebra Communication Subroutines)** : dédiée aux opérations de communications pour la parallélisation des BLAS ou pour LAPACK

- bibliothèque spécifique au calcul matriciel ;
- routines de bas niveau :
  - ◇ SD : envoyer un message
  - ◇ RV : recevoir un message
- routines de haut niveau :
  - ◇ BS : diffuser un message
  - ◇ BR : recevoir un message diffusé
- opérateurs globaux : MAX, MIN, SUM
- structures de données échangées : matrices rectangulaires et trapézoïdales.

