High Performance Computing et calcul parallèle

D. Cornu,
UTINAM / Observatoire de Besancon

Master 2 - P2N & PICS

Automne 2019

Crédits et remerciement à Benoit Semelin, LERMA (OBSPM)

(Open) MPI : Message Passing Interface

Méthodes de parallélisation

Architecture matérielle:

SISD	SIMD	MIMD
Single Instruction Single Data	Single Instruction Multiple Data	Multiple Instruction Multiple Data
PC monoprocesseur	Architecture **Vectorielle,** MMX,SSE, GPU	Architecture **parallèle** multiprocesseur

Modèle de programmation:

Le plus utilisé →

SPMD

Single Program Multiple Data

Un seul programme (n° processeur = variable)

MPMD

Multiple Program Multiple Data

Création dynamique de process

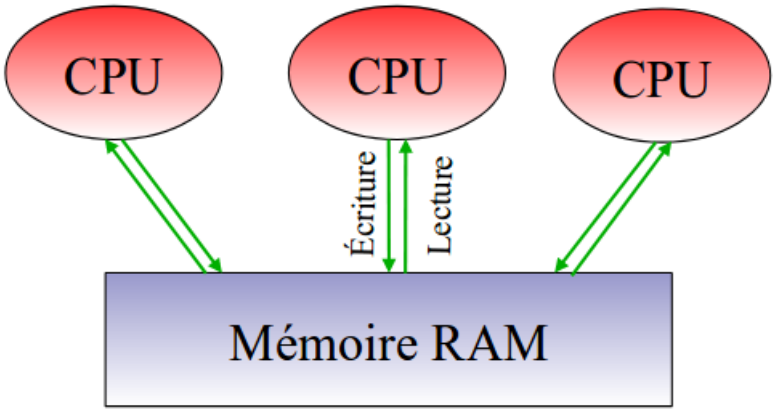
Modèle maître-esclave.

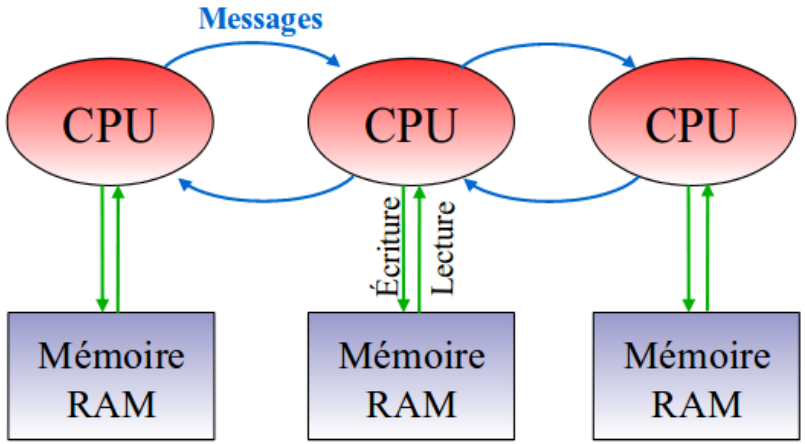
Outils de parallélisation:

OpenMP
ordinateur à **mémoire partagée**

MPI
ordinateur à **mémoire distribuée**

Mémoire Partagée Vs Distribuée





Les processeurs partagent la même mémoire

- Pas ou peu de surcoût lié à la parallélisation
- Possibles conflits d'accès à la mémoire
- Matériel coûteux. Une seule machine avec de nombreux coeurs
- Le plus souvent un nombre de coeurs < 128

Chaque processeur dispose de sa propre mémoire et ne peut pas accéder à celle des autres

- Surcoût pour l'échange des messages
- Matériel plus accessible, mais coût d'architecture
- Sensible à la performance du système de communication
- Nombre de coeurs $\approx$ illimité

Comprendre les outils

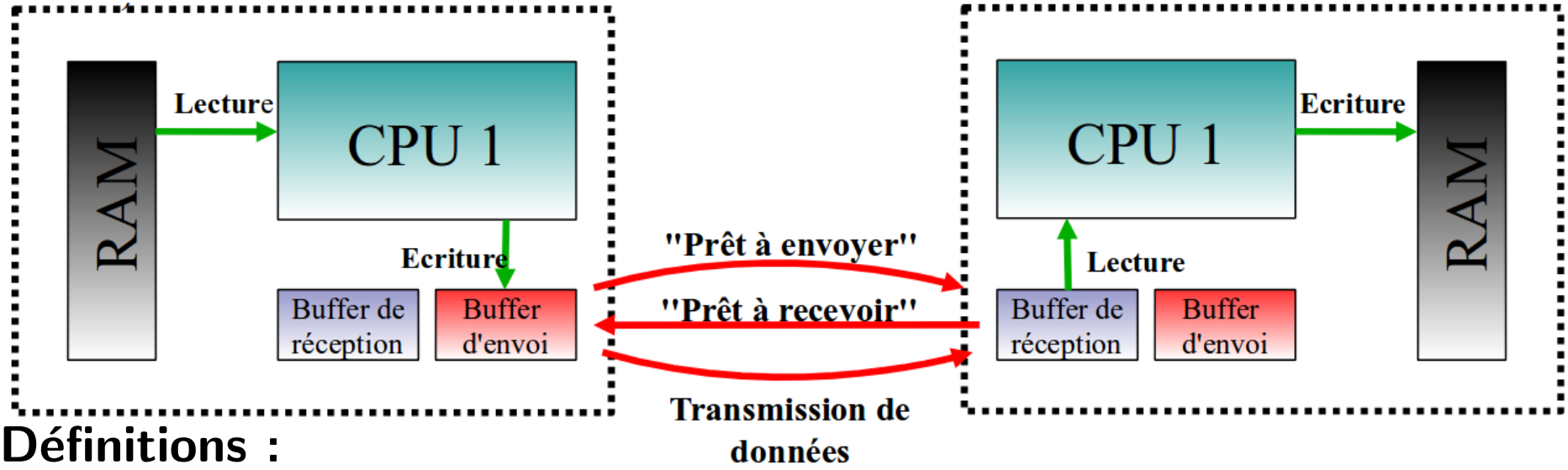
Plusieurs niveaux auquels effectuer la parallélisation

- **Langages ou extensions**
  - CUDA (GPU)*
  - OpenCL(divers)*
- **Bilbiothèques**
  - Message Passing Interface MPI*
  - PosixThreads
- **Directives de compilation**
  - OpenMP*
  - Accélération GPU
- **Compilateur autonome**
  - Fortran/C inclu dans le compilateur

*Méthodes usuelles pour le calcul

Mémoire distribuée : Communication

Exemple pour une communication : Two-Sided, Buffered, Synchrone

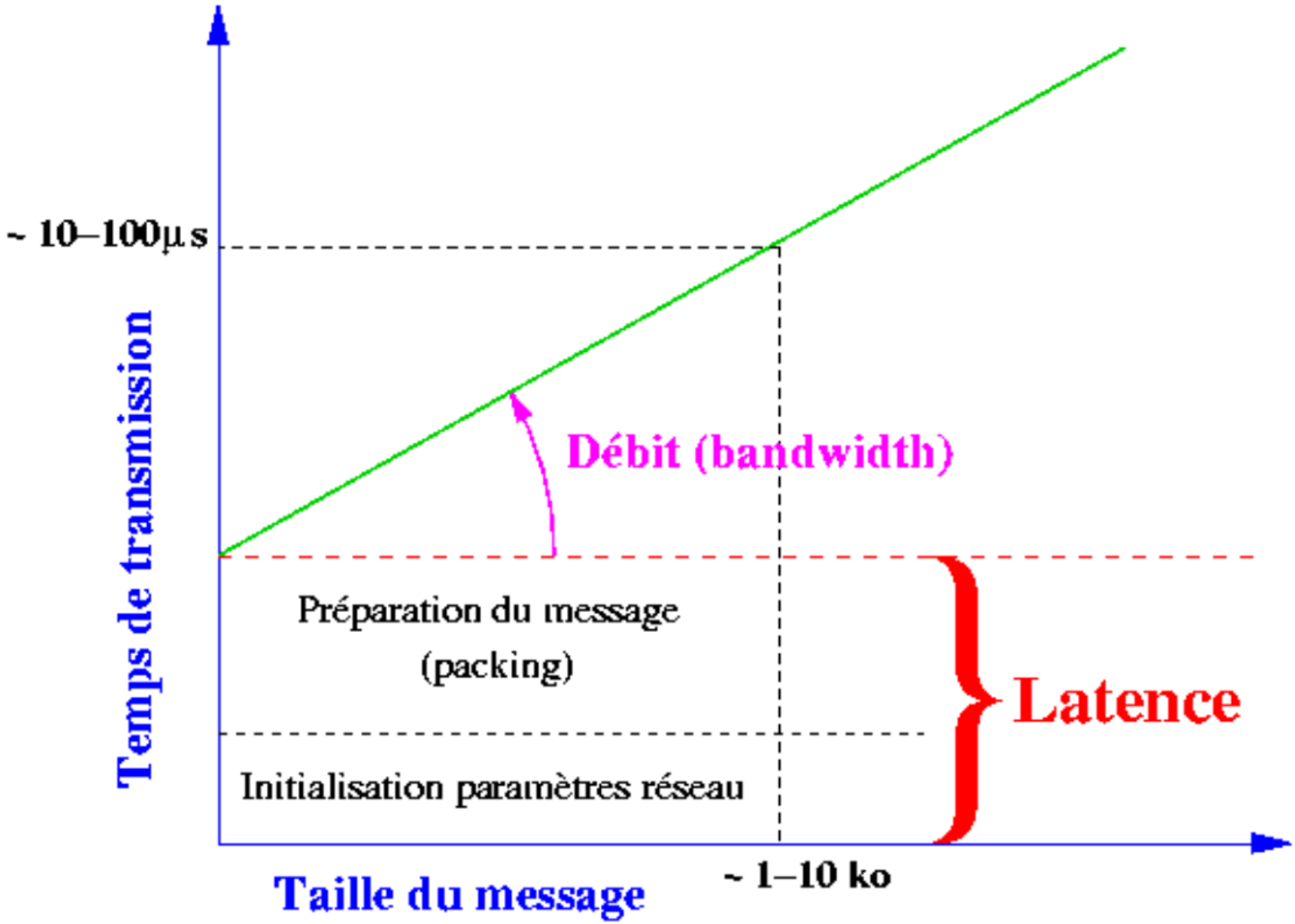


Définitions :

- **One-sided** : Un seul processeur gère tout seul la communication
- **Two-sided** : les deux processeurs se partagent la tache de communication

- **Synchrone** : Le processeur qui envoie attend que la lecture ait été validée
- **Asynchrone** : Le processeur envoie les données puis continue sans savoir si l'autre est prêt à la recevoir. Celui qui la réceptionne la lira dans son "buffer" de réception une fois l'instruction correspondante rencontrée

Coût de communication



Performance d'une application parallèle

Coût des communications :

- Un temps de communication trop important ralenti le programme. Si ce temps dépasse 10% du temps de calcul total, inutile d'utiliser plus de ≈ 10 processeurs.
- Exemples pour minimiser le temps de communication :
  - Recouvrement des domaines de calcul
  - Duplication des données critiques
  - Optimisation de la taille des communications ...

Equilibre de charge :

- Avoir un processus deux fois plus lent que les autres équivaut à calculer sur deux fois moins de processeurs...
- L'équilibrage dynamique des charge peut en partie régler le problème. Utile pour les problèmes qui évoluent dans le temps (ex : propagation). Il faut alors diffuser la charge de travail, et changer la taille des domaines de manière appropriée.

MPI

MPI est une **bibliothèque** de communication parallèle pour architecture à mémoire partagée. Elle fonctionne avec Fortran, C et C++.

Elle se définie par des standards qui ont étés établis en 1994 et qui évoluent encore aujourd'hui (derniers standards en 2015).

L'implémentation gratuite la plus courante est OpenMPI, dont nous nous servirons aujourd'hui.

MPI

MPI est une **bibliothèque** de communication parallèle pour architecture à mémoire partagée. Elle fonctionne avec Fortran, C et C++.

Elle se définie par des standards qui ont étés établis en 1994 et qui évoluent encore aujourd'hui (derniers standards en 2015).

L'implémentation gratuite la plus courante est OpenMPI, dont nous nous servirons aujourd'hui.

Exercice 1 :

- Ecrire un programme séquentiel "Hello World" qui se contente d'afficher un message sur la sortie standard.
- Le compiler en MPI avec la commande
  mpifort mon_programme.f90 -o mon_programme
- L'exécuter en MPI avec la commande (où n = nb_procs)
  mpirun -n 4 ./mon_programme

Premier programme MPI

Avec MPI une seul zone parallèle est définie.
C'est donc en général l'ensemble du programme, qui est executé plusieurs fois simultanément.
Les variables sont donc toutes locales !
Une zone parallèle représente ici une **zone de communication** entre les "instances" du même programme.

MPI étant une bibliothèque, il faut l'importer dans le programme pour en utiliser les **instructions** (routines). Ce qui se fait avec **USE** MPI avant le **implicit none**, ou bien avec **include** "mpif.h"

On définit la zone de communication avec :

call MPI_INIT(error)

Et on la termine avec :

call MPI_FINALIZE(error)

Remarque : La variable "error" est obligatoire et doit être déclarée comme un entier

Premier programme MPI

L'initialisation de MPI définit un communicateur par défaut : MPI_COMM_WORLD.

Le plus souvent il est le seul communicateur utilisé.
Néanmoins il est possible de définir des groupes de processus et de leur attribuer un communicateur (Intracommunicateurs), mais également de définir des communicateurs entre ces groupes (Intercommunicateurs).

Fonctions de manipulation utiles

- MPI_COMM_SIZE(comm, **size**, error) Renvoie **size**, le nombre de processus dans le communicateur comm
- MPI_COMM_RANK(comm, **rank**, error) Renvoie **rank**, le numéro du processeur appelant la fonction

Remarque : Ici comm sera remplacé par MPI_COMM_WORLD, et size et rank sont des entiers

Premier programme MPI

Exercice 2 :

- Ecrire un programme parallèle "Hello World" ou chaque processus affiche son rang et le nombre total de processus.
- Le compiler, puis l'executer plusieurs fois. Quel comportement en ressort ?

Rappel des procédures :
USE MPI
MPI_INIT(error) défini MPI_COMM_WORLD
MPI_FINALIZE(error)
MPI_COMM_SIZE(comm, **size**, error)
MPI_COMM_RANK(comm, **rank**, error)

Structure d'un message MPI

Un processus peut envoyer à un autre des données **via un message**.

En plus de contenu du message il faut constituer une **"enveloppe"** qui contient les informations suivantes :

- **Source** : rang du processeur qui envoie
- **Destination** : rang du processeur qui recoit
- **Etiquette** : valeur unique attribué à un message
- **Communicateur** : structure dans laquelle se fait l'échange

Pour ce faire on utilise des **procédures** définies dans le standard MPI

Modes de communication

Les communications se séparent en deux types :

- **Bloquante** : la procédure se termine quand les ressources engagées **peuvent être réutilisées** et que la fonction "complète"
- **Non-bloquante** : La procédure se termine **avant** que les ressources soient **libérées**, et le programme continue. Il faut alors surveiller que les ressources ne soient pas modifiées avant leur lecture. Puis **s'assurer plus tard que la communication à été effectuée** (la fonction "complète" à ce moment)

Puis on distingues différents modes d'envois :

- **Buffered** : Le message est **stocké dans un buffer local** avant d'être envoyé. La fonction se termine quand la copie est finie, avant l'envoi.
- **Synchrone** : L'envoi du message ne commence que quand *le processus qui va le réceptionner est prêt*. Se termine à ce moment la.
- **Ready** : Envoi le message **sans vérifier l'état du récepteur**. Peut donc perdre des données ! Se termine dès l'envoi terminé.
- **Standard** : Utilise le mode Buffered ou Synchrone en fonction des disponibilités de la mémoire.

Procedures de communication

En pratique on utilise le mode standard et l'attribut bloquant ou non en fonction des circonstances.

	Bloquant	Non-bloquant
Standard	MPI_SEND	MPI_ISEND
Synchrone	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Ces procédures suivent toutes le même prototype :
MPI_SEND(**data**, **count**, datatype, dest, tag, comm, error)
MPI_ISEND(**data**, **count**, datatype, dest, tag, comm, request, error)
MPI_RECV(**data**, **count**, datatype, source, tag, comm, **status**, error)
MPI_IRECV(**data**, **count**, datatype, source, tag, comm, request, error)

status de type **integer**, **dimension**(MPI_STATUS_SIZE)
datatype : MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LOGICAL, MPI_CHARACTER

Autres notions

- Il est possible de **creer ses propres types** ou bien d'envoyer des structures (MPI_PACKED)
- Il est possible de remplacer la source par MPI_ANY_SOURCE et le tag par MPI_ANY_TAG
- Vérifier les transferts de messages pour les communications non bloquantes avec "requete"
  - MPI_WAIT(requete, **status**, error) : attend que la communication associée soit completé
  - MPI_WAITALL(**count**, requete, **status**, error) : identique mais avec un vecteur de taille count
  - Diverses fonctions de test non bloquantes à voir dans la documention
- Il est possible de **forcer la synchronisation des processus** avec MPI_BARRIER(comm, error) qui est une fonction collective.
  La synchronisation deux à deux des processus peut se faire via une communication bloquante synchrone.

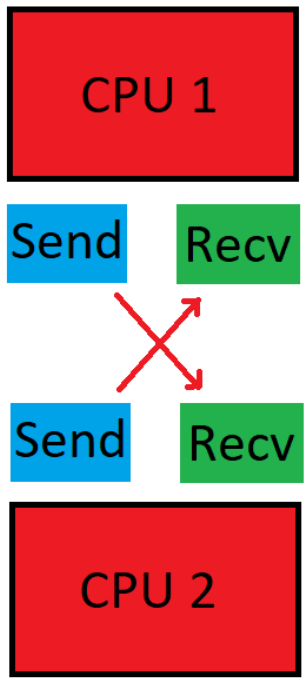
Communication point à point

Exercice 3 :

- Ecrire un programme parallèle avec 2 processus ou chacun envoie à l'autre un vecteur de nombres alétoires.
- Vérifier l'absence "d'interblocage", et vérifier que les données sont bien échangées en demandant à chaque processus d'afficher ce qu'il envoie et ce qu'il recoit.

Générateur aléatoire

```
 1 program main
 2   real, dimension(4) :: send
 3   integer, dimension(:), allocatable ::
         seed
 4   integer :: n, rank, clock
 5   call SYSTEM_CLOCK(count=clock)
 6   call RANDOM_SEED(size = n)
 7   allocate(seed(n)); seed = clock
 8   call RANDOM_SEED(PUT = seed)
 9   call RANDOM_NUMBER(send)
10   deallocate(seed)
11   print *, send
12 end program main
```



Script SGE - MPI

Pour executer vos programmes MPI sur le mésocentre vous devez suivre la même procédure que pour le cours OpenMP. L'execution se fait via un **script SGE** lancé avec la commande **qsub**. Quelques différences sont à noter par rapport au script OpenMP.

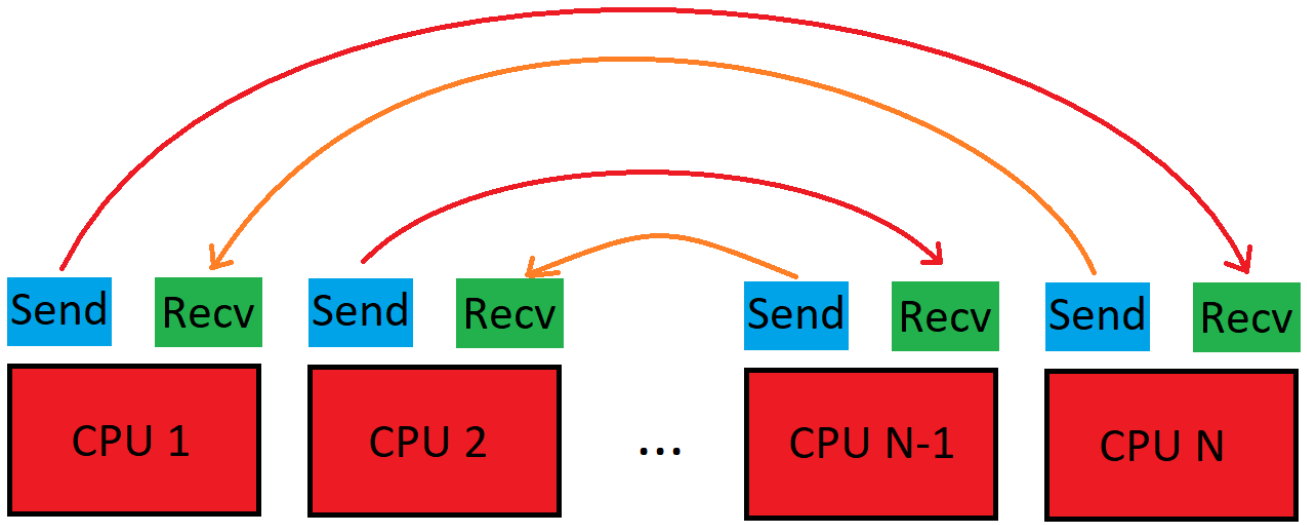
```
#!/bin/bash
#$ -q formation.q     ## on demande la file formation.q
#$ -N tp_myname       ## le nom de votre job
#$ -pe mpi 6           ## on demande 6 coeurs
#$ -l h_vmem=4G             ## on demande 4G/coeur
#$ -o results.out  ## le nom de fichier output/err

### Charge le module open mpi

module load mpi/openmpi/icc/1.7.5

## on lance l'application
time mpirun -np 6 ./appli_mpi
```

Le reste des fonctions de gestion des jobs restent identiques à OpenMP
Ce script est fourni sur le GitHub

Communication point à point > 2 procsessus

Exercice 4 :

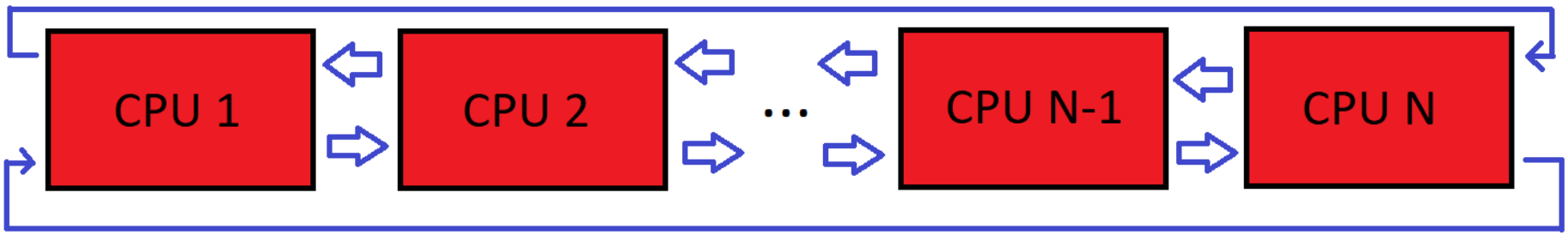
- Ecrire un programme parallèle avec $n > 2$ processus ou le premier envoie au dernier et réciproquement, le suivant à l'avant dernier et réciproquement, ainsi de suite ...
- Le programme doit fonctionner indépendamment du nombre de processus.
- Vérifier que les données sont bien échangées en demandant à chaque processus d'afficher ce qu'il envoie et ce qu'il reçoit.



Communication circulaire

Exercice 5 :

- Ecrire un programme parallèle avec $n > 2$ processus ou chaque processus transmets son identifiant (ou une donnée identifiable) à ses voisins, et réciproquement. (les bords transmettent à l'autre extrémité, condition aux bords périodique)
- Le programme doit fonctionner indépendamment du nombre de processus.
- Vérifier que les données sont bien échangées en demandant à chaque processus d'afficher son identifiant et LES informations qu'il reçoit.



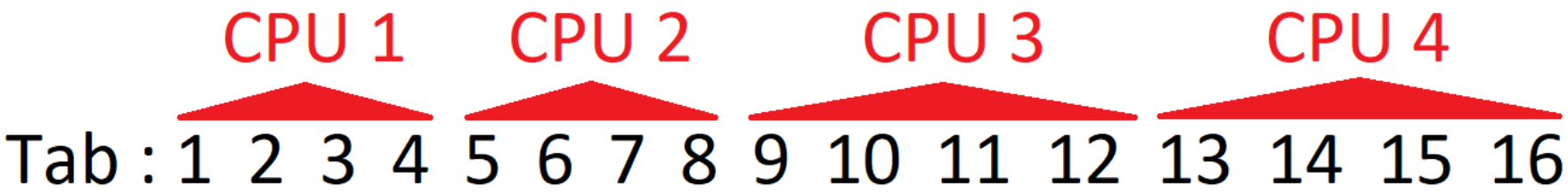
Communications Globales / Collectives

Il s'agit d'opérations de communication qui ont **un comportement identique sur l'ensemble des processus** du communicateur. Elle doit donc etre appelée **par tous les processus**.

- MPI_BCAST(adresse, **count**, datatype, root, comm, error) Procédure qui diffuse une valeur depuis root sur l'ensemble des processus.
- MPI_GATHER(s_add, s_count, s_type, r_add, r_count, r_type, root, comm, error) Procédure qui récupère une donnée sur chaque processus et la place dans un tableau sur root.
- MPI_REDUCE(s_add, r_add, **count**, datatype, op, root, comm, error) Procédure qui effectue l'opération op (MPI_SUM, MPI_PROD, ...) sur la variable s_add de chaque processus, et place le résultat dans r_add sur root.
- MPI_SCATTER(s_add, s_count, s_type, r_add, r_count, r_type, root, comm, error) Procédure qui sépare un tableau s_add, et en envoie une partie sur chaque processus.
- Autres fonctions de distribution MPI_ALLGATHER, MPI_ALLTOALL, MPI_ALLREDUCE

Exercice 6 :

- Ecrire un programme parallèle avec $n > 2$, où l'un des processus défini un tableau contenant les n premiers entiers. Il décompose ensuite ce tableau et le distribue sur les autres processus. Les parties transférées doivent êtres des portions continues de ce tableau et contenir plusieurs cases.
- Le programme doit fonctionner indépendamment du nombre de processus.
- Vérifier que les données sont bien échangées en demandant à chaque processus d'afficher son identifiant et les informations qu'il recoit.



Aller plus loins

Pour aller plus loin avec MPI :

- Communications non-prédictibles
- Communications one-sided (un processus va écrire dans une mémoire distante)
- Types dérivés, et structure
- Vecteurs spécifiques et sections de tableau dans les transferts
- Communications "one-sided" (PUT, GET, ...)

Pour aller plus loin sur sur la Parallélisation :

- Topologie de processus
- Entrées/sorties parallèles
- Créaction dynamique de processus
- Autres support de parallélisation (Vectoriel, GPU, ...)