Calcul parallèle distribué

Xavier JUVIGNY

ONERA

14 Novembre 2019

Plan du cours

Calculateur à mémoire distribuée

Définitions

Nœud de calcul : Une ou plusieurs unités de calculs et une mémoire vive;

Calculateur à mémoire distribuée : Calculateur contenant plusieurs nœuds de calcul pouvant communiqués entre eux au travers d'un bus spécialisé ou un réseau éthernet.

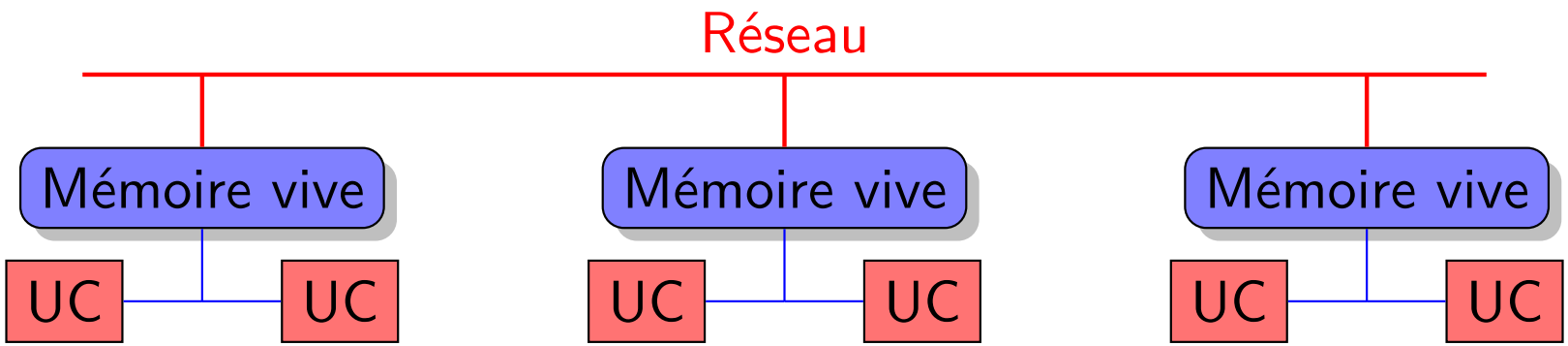


Figure: Architecture calculateur à mémoire distribuée

Modèle de programmation

Modèle de programmation

Exécute simultanément `nbp` processus;

Chaque processus a un numéro unique (de 0 à `nbp-1`);

Les processus échangent des données en envoyant/recevant des messages;

message envoyé : les données, quantité de données, types de données, destinataire, identifiant du message (et groupe de communication...)

message reçu : buffer réception, quantité de données attendues, type données, expéditeur, identifiant du message (groupe de communication...), status échange (ok, incomplète, ...).

Coût d'un envoi

$$T_{\text{Envoi}} = \text{Temps initialisation message} + N.\text{Taux de transfert}$$

N : Quantité de données à envoyer (Mb)
Si le nœud expéditeur n'est pas connecté directement au nœud destinataire : passe par des nœuds transitoires $\Rightarrow$ augmentation du temps d'envoi/réception

Topologie d'un réseau

Topologie d'une réseau : Manière dont sont interconnectés les nœuds de calcul

Définitions

Distance entre deux nœuds : Nombre minimal de nœuds intermédiaire par lesquels doit passer le message;

Diamètre d'un réseau : Distance maximale possible entre deux nœuds du calculateur parallèle.

Topologie linéaire

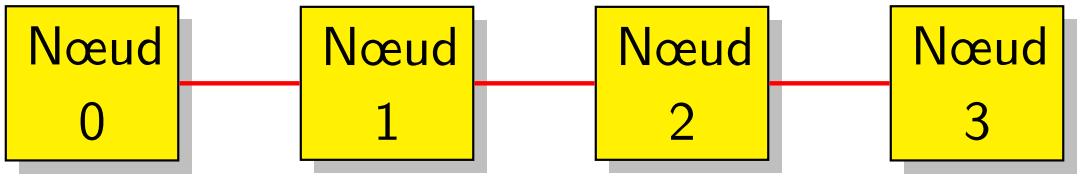


Figure: Réseau Linéaire

diamètre $= N - 1$.

Topologie en anneau

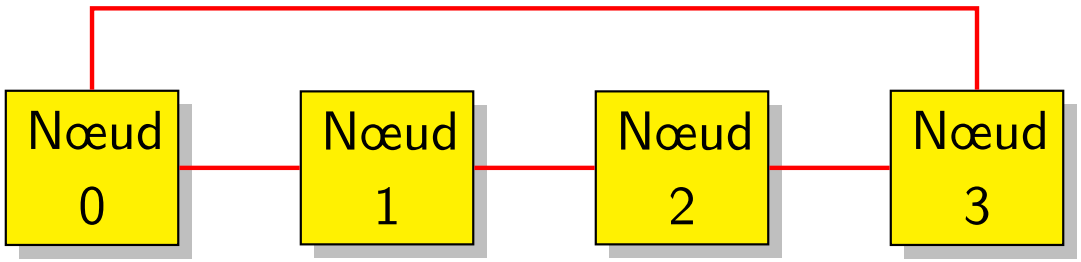


Figure: Réseau en anneau

diamètre $= \frac{N}{2}$ si N pair, $= \frac{N-1}{2}$ si N impair.

Topologie grille

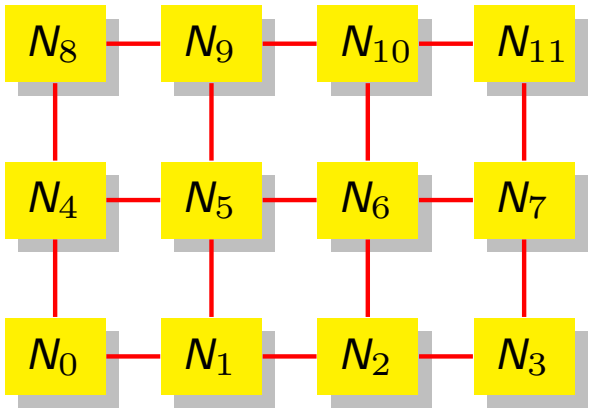
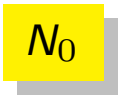


Figure: Réseau sur grille

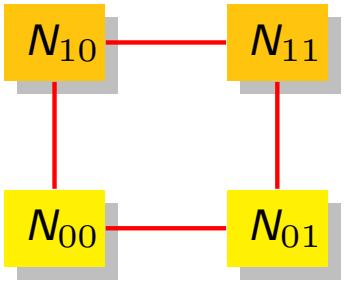
Diamètre $= W + H - 2$

Réseau hypercube

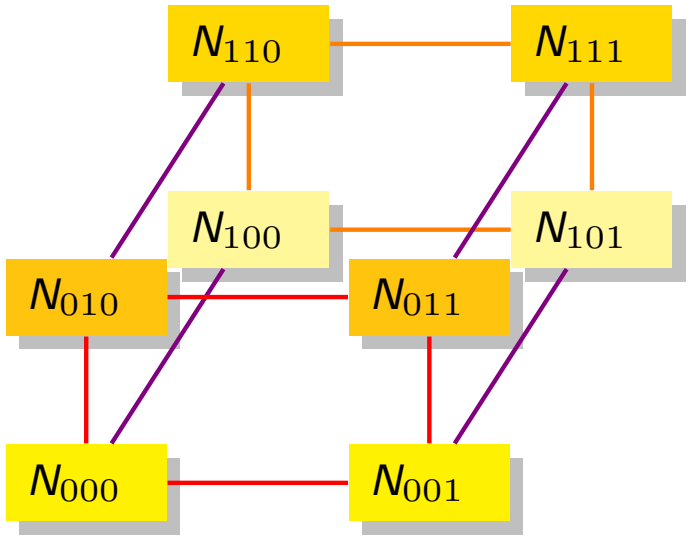


(a) $\mathcal{C}_0$: hypercube dimension 0

(b) $\mathcal{C}_1$: hypercube dimension 1



(c) $\mathcal{C}_2$: hypercube dimension 2

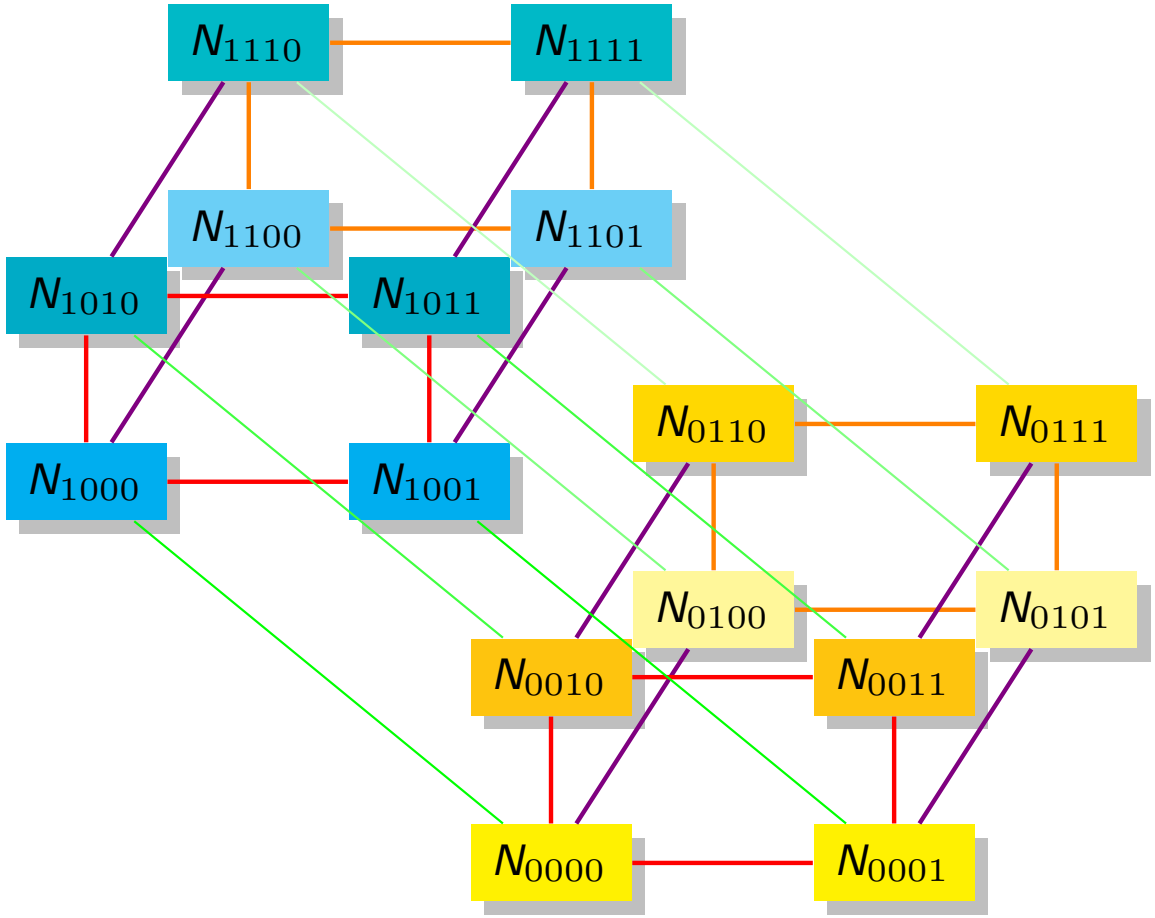


(d) $\mathcal{C}_3$: hypercube dimension 3

Figure: Réseau hyper cube de dimensions 0 à 3.

Nœuds numérotés en binaire selon numérotation de Gray.
Diamètre du réseau = dimension du cube.

Réseau hypercube



(a) $\mathcal{C}_4$: hypercube dimension 4

Figure: Réseau hyper cube de dimension 4.

Modèles de programmation

SPSD (**S**imple **P**rogram **S**imple **D**ata) : programmation séquentielle classique

SPMD (**S**imple **P**rogram **Multiple Data**) : Même programme exécuté sur chaque nœud, Données différentes

À ne pas confondre avec architecture SIMD !!!!!

MPMD (**M**ultiple **P**rogram **M**ultiple **D**ata) :

Chaque nœud exécute un programme différent avec des données différentes

Remarque

Possibilité d'émuler le MPMD avec du SPMD

```
if ( processus == 0 )
  f1 ();
if ( processus == 1 )
  f2 ();
...
```

Contexte parallèle

Chaque processus se voit attribué un identifiant unique : un entier unique par exemple;

Chaque processus peut connaître le nombre de processus exécutés par l'application;

Permet d'atribuer les tâches selon l'identifiant et le nombre de processus;

Permet d'identifier le destinataire ou l'expéditeur d'un message.

Communication point à point

Constitution d'un envoi

Les données à envoyer (et leurs types et quantité);

Le numéro du destinataire;

Un identifiant pour le message;

Le groupe de communication utilisé.

Constitution d'une réception

Un buffer de réception (et sa taille);

Le type des données à recevoir;

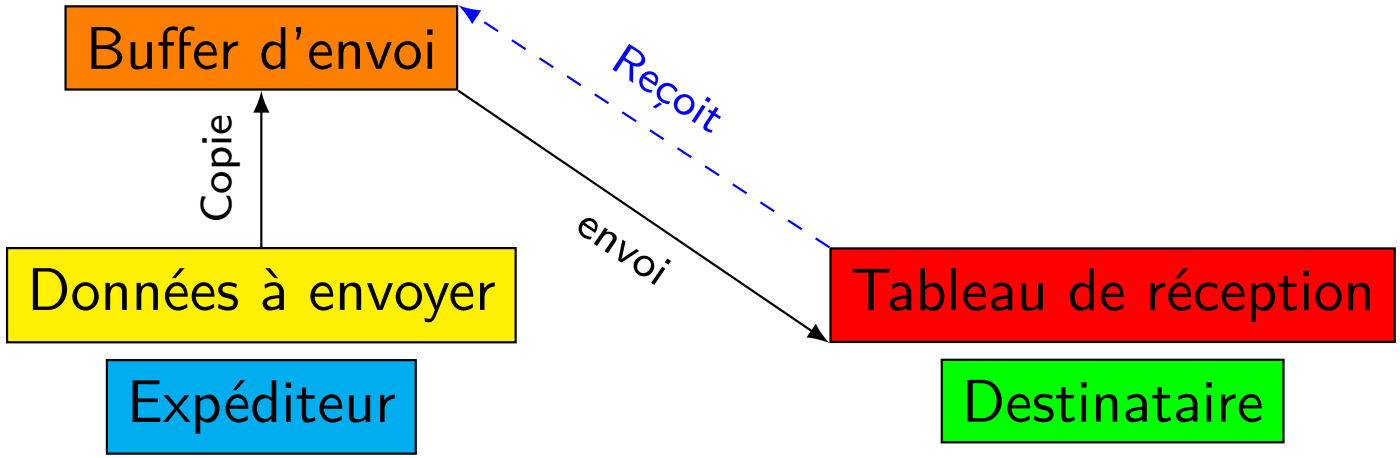
Le numéro de l'expéditeur;

L'identifiant du message;

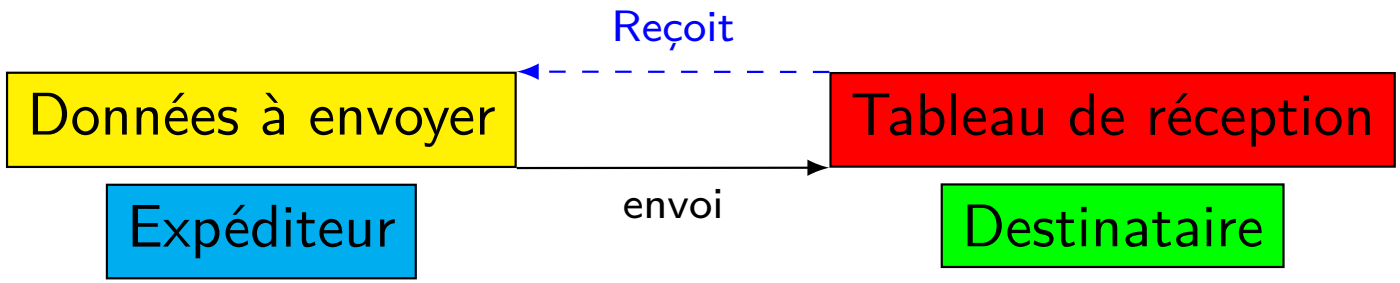
Le groupe de communication utilisé.

Le status de la réception (ok, incomplète, raté, ...)

Envoi/Réception bufférisés/non bufférisés



(a) Schéma d'envoi bufférisé



(b) Schéma d'envoi non bufférisé

Envoi/Réception synchrone

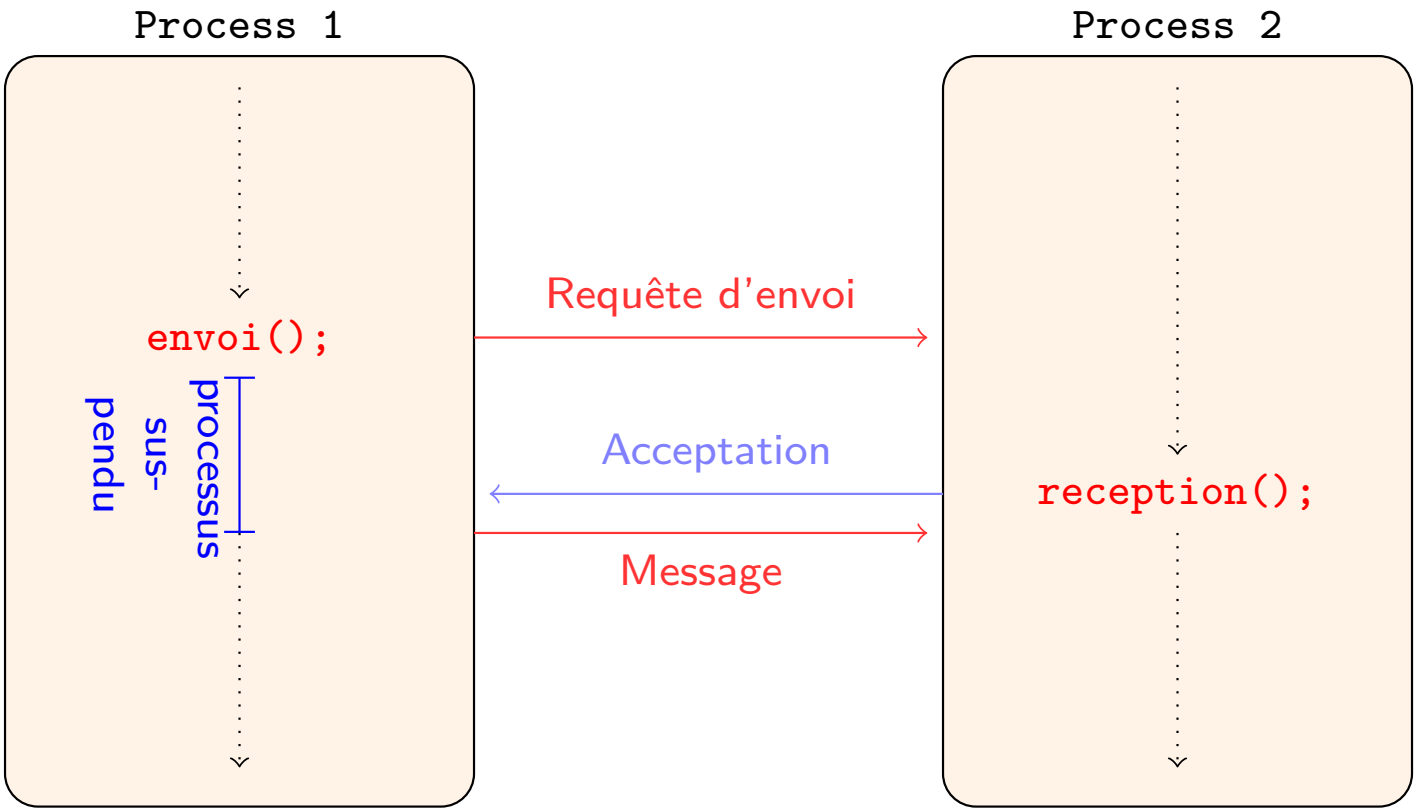


Figure: Exemple message synchrone quand la réception est exécutée après l'envoi

Envoi/Réception synchrone

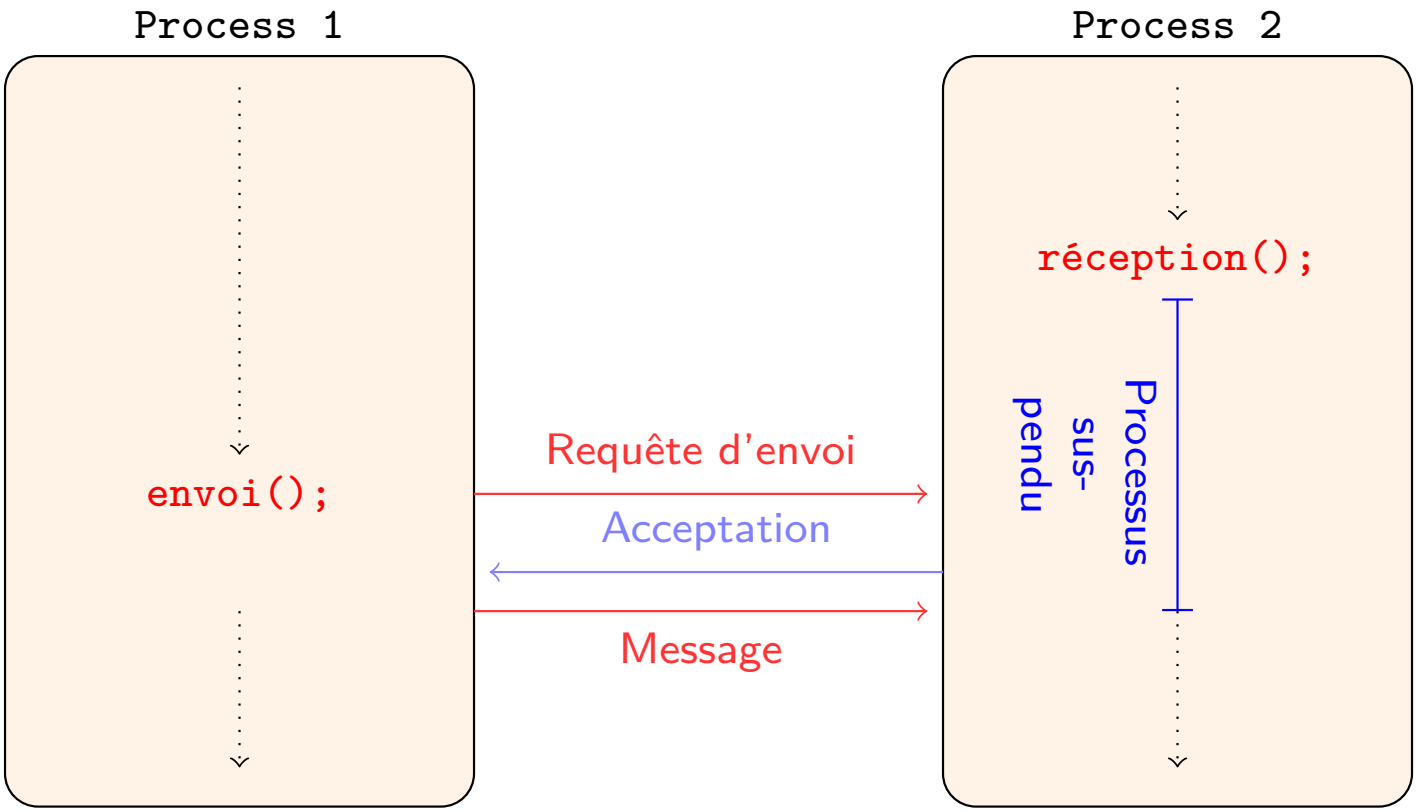


Figure: Exemple message synchrone quand la réception est exécutée avant l'envoi

Interblocage (deadlock)

Cas d'interblocage

tous les processus envoient message bloquant à un autre processus avant d'effectuer une réception;

Ou cas symétrique où tous les processus attendent un message en réception avant d'effectuer un envoi.

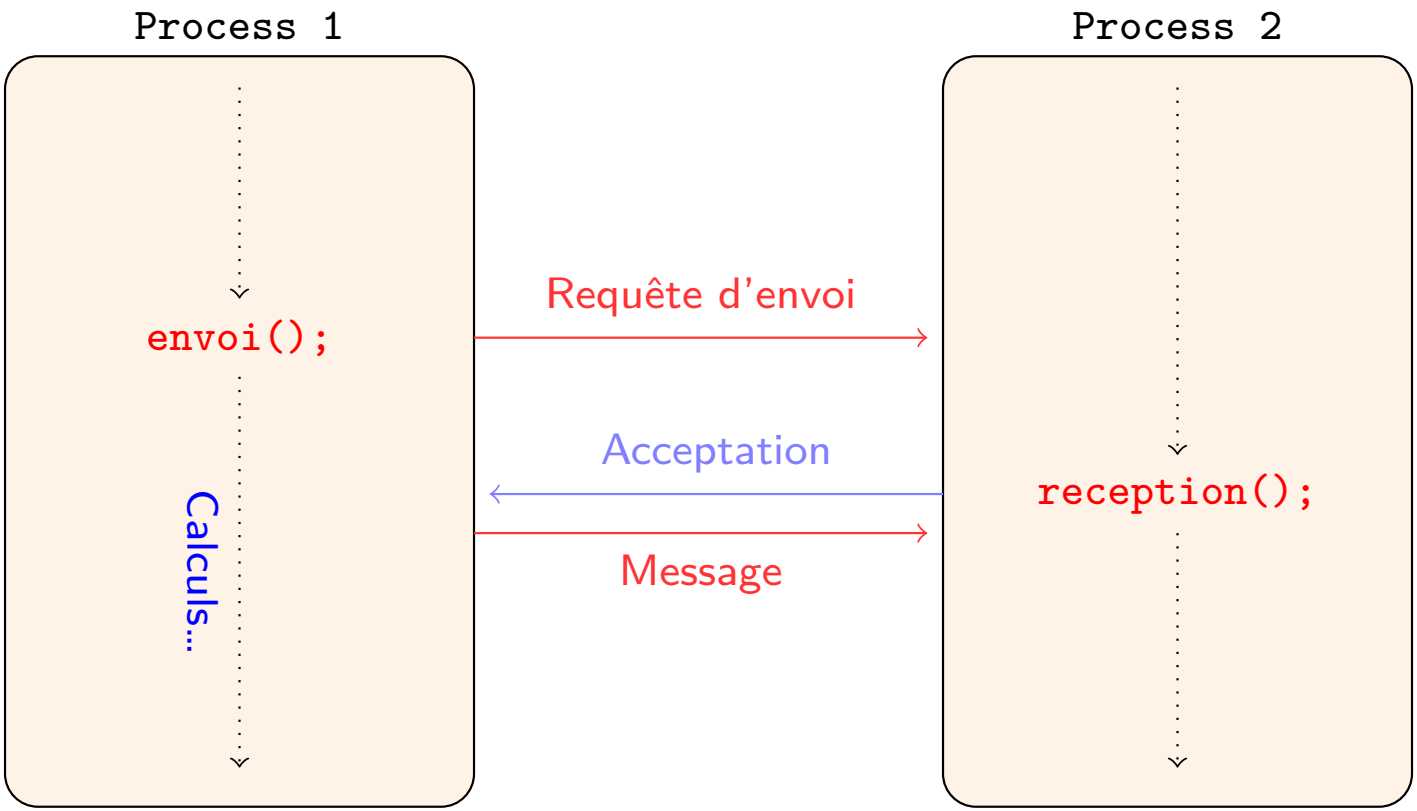
Example

```
if (rank == 0)
{
    Recoit_synchrone( recvdata, count, tag, 1 );
    Envoi_synchrone ( senddata, count, tag, 1 );
}
else
{
    Recoit_synchrone( recvdata, count, tag, 0 );
    Envoi_synchrone ( senddata, count, tag, 0 );
}
```

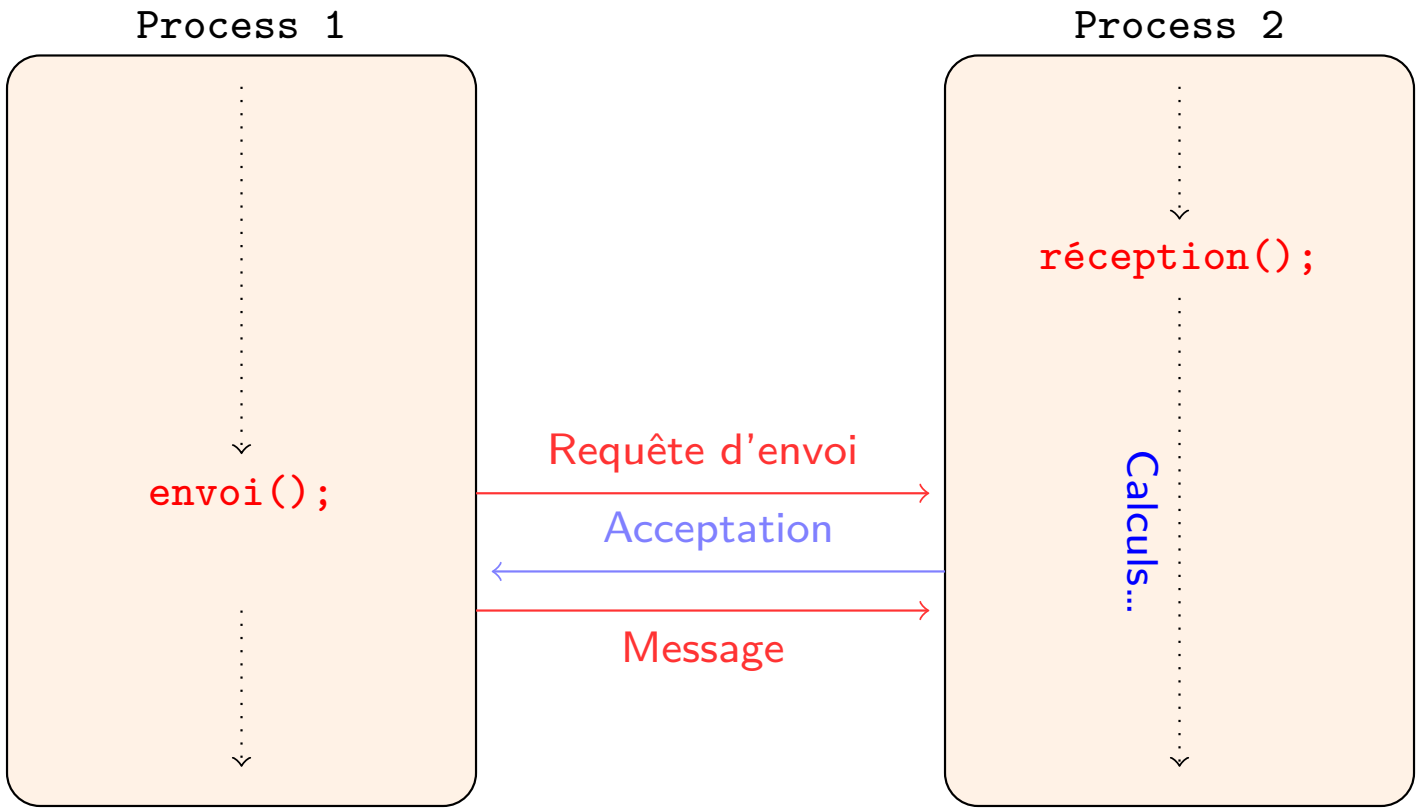
Première solution pour enlever interblocage

```
if (rank == 0)
{
    Recoit_synchrone( recvdata, count, tag, 1 );
    Envoi_synchrone ( senddata, count, tag, 1 );
}
else
{
    Envoi_synchrone ( senddata, count, tag, 0 );
    Recoit_synchrone( recvdata, count, tag, 0 );
}
```

Envoi/Réception asynchrone



Envoi/Réception asynchrone



Synchronisation reportée

Tester si l'envoi ou la réception a bien été effectuée;

Attendre plus tard que l'envoi (ou la réception) soit effectuée;

Deuxième solution pour enlever l'interblocage

```
if (rank == 0)
{
    Recoit_asynchrone( recvdata, count, tag, 1, status );
    Envoie_synchrone ( senddata, count, tag, 1 );
    Attend_fin_reception( status );
}
else
{
    Recoit_asynchrone( recvdata, count, tag, 0, status );
    Envoie_synchrone ( senddata, count, tag, 0 );
    Attend_fin_reception( status );
}
```

Exercice sur l'interblocage

Exercice

Expliquez pourquoi le code MPI ci dessous n'est pas sûr et peut par moment conduire à un interblocage.

```
MPI_Comm_rank(comm, &myRank ) ;
if (myRank == 0 ) {
    MPI_Ssend( sendbuf1, count, MPI_INT, 2, tag, comm);
    MPI_Recv( recvbuf1, count, MPI_INT, 2, tag, comm, &status);
} else if ( myRank == 1 ) {
    MPI_Ssend( sendbuf2, count, MPI_INT, 2, tag, comm);
  else if ( myRank == 2 ) {
    MPI_Recv( recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
    MPI_Ssend( sendbuf2, count, MPI_INT, 0, tag, comm);
    MPI_Recv( recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
}
```

Les différentes sortes de communications collectives

Effectuer une barrière de synchronisation;

Mouvement collectif de données

- Diffusion de données d'un processus sur tous les autres processus;
- Rassembler sur un ou tous les processus des données réparties sur chaque processus;
- Distribuer et répartir des données d'un processus sur tous les autres processus;
- Échanger des données de tous vers tous.

Des calculs globaux :

- Effectuer des réductions, c'est à dire faire une opérations sur les données réparties sur les processus (somme, max, multiplication, etc.);
- Effectuer un scan, opération cumulative, sur les données réparties sur les processus.

Optimalité

Permet d'être optimal sur toute topologie de réseau

Synchronisation

Créee un point de rendez-vous pour tous les processus.
Utile pour faire des mesures de temps entre autre.

```
std::chrono::time_point<std::chrono::system_clock> start, end;
// Tous les processus s'attendent à la ligne suivante
// afin de démarrer en même temps la partie du code
// dont on veut mesurer les performances en parallèle
MPI_Barrier(MPI_COMM_WORLD);
start = std::chrono::system_clock::now();
// La partie du code dont on veut mesurer les performances
...
end = std::chrono::system_clock::now();
...
```

Diffusion (Broadcast)

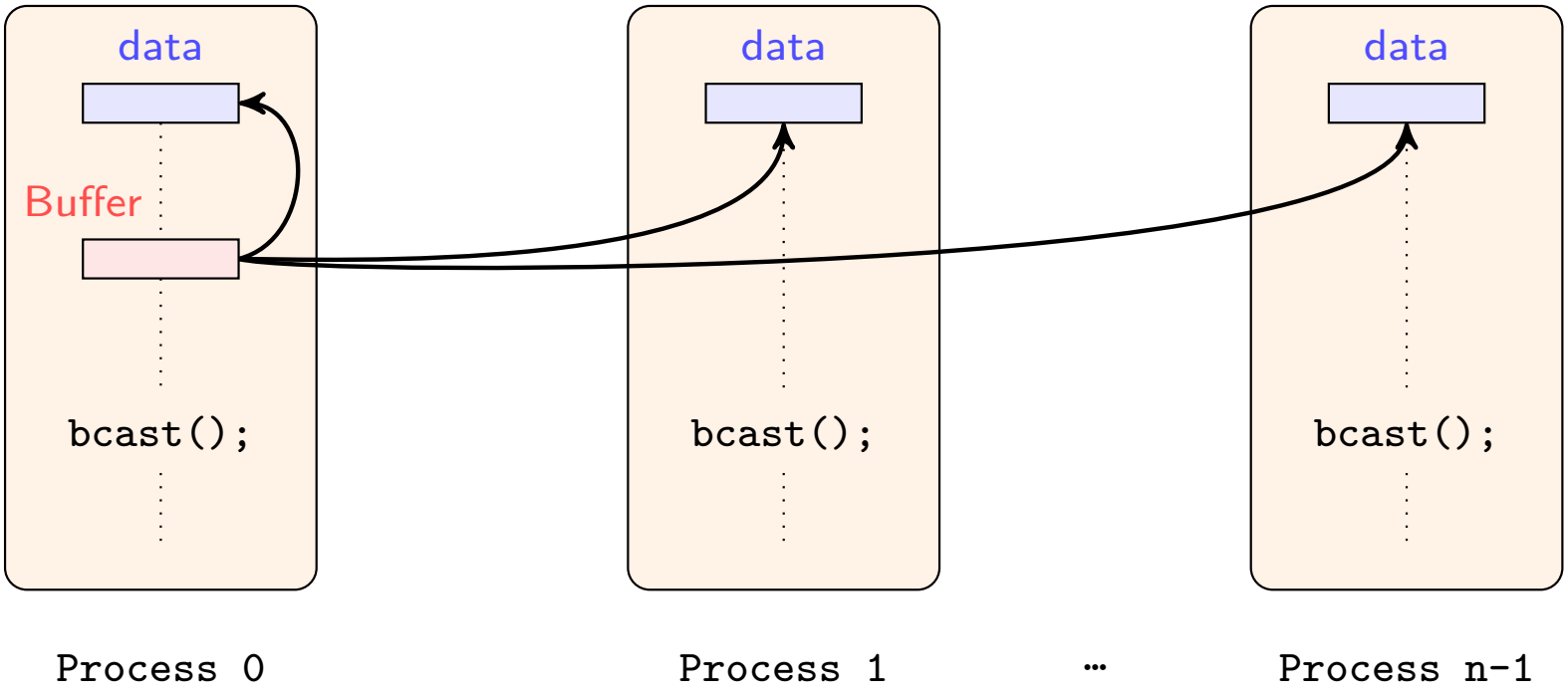


Figure: Principe de la diffusion

Répartition (Scatter)

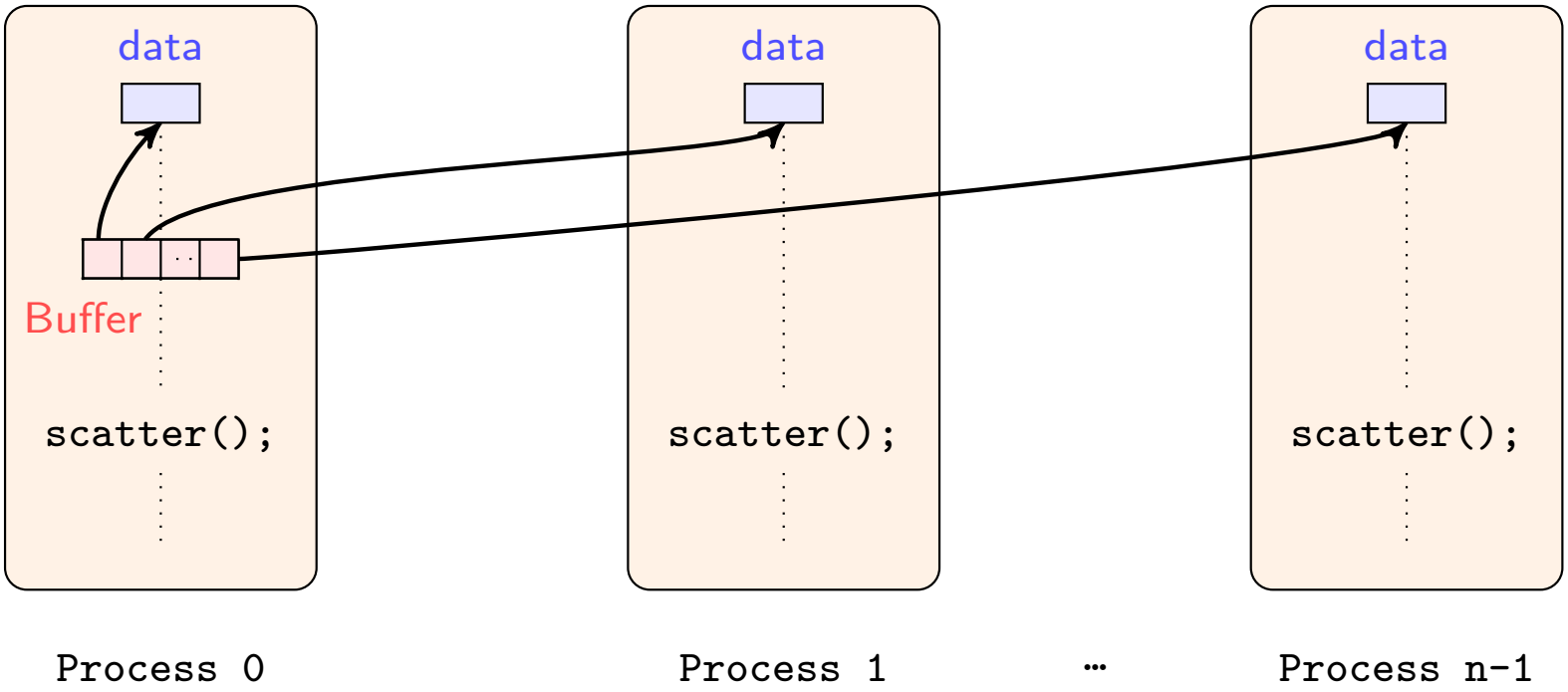


Figure: Schéma du fonctionnement d'une répartition collective

Rassemblement (Gather)

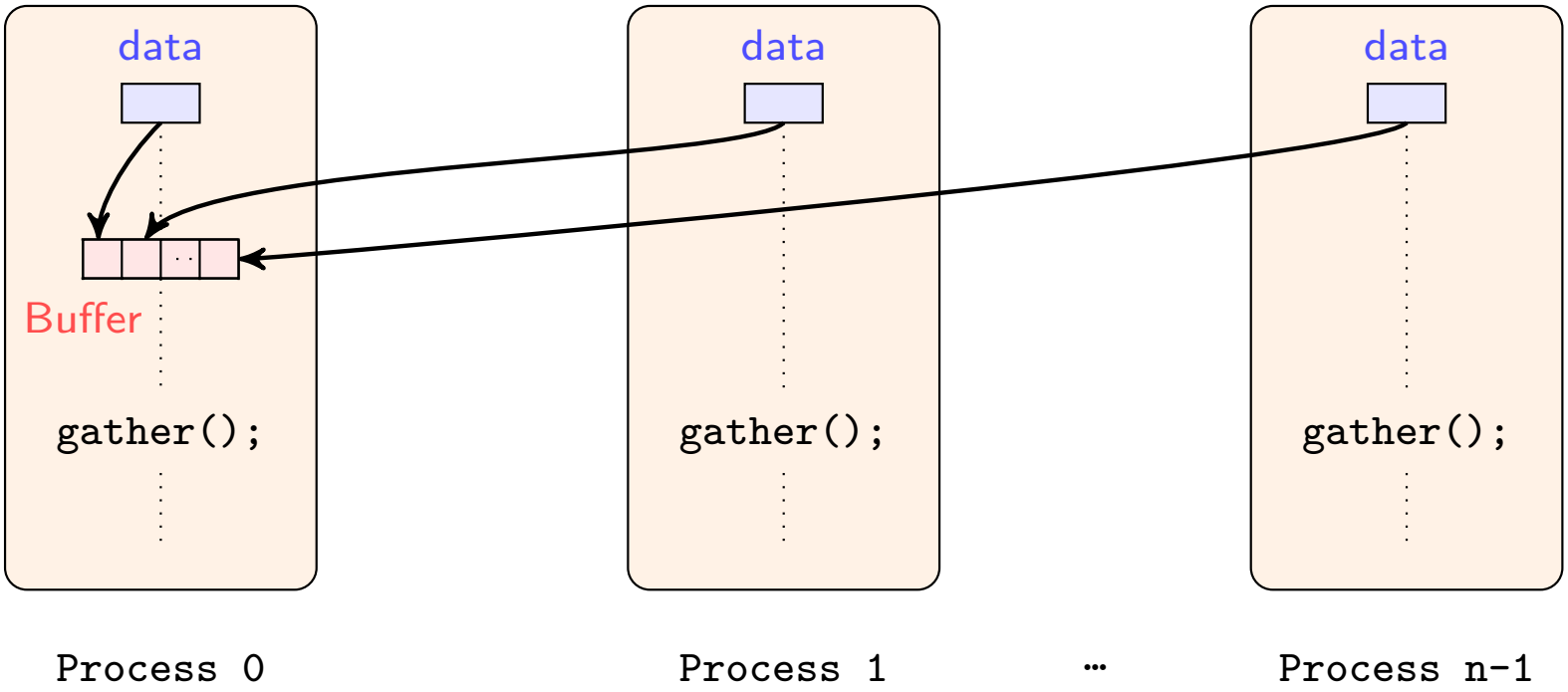
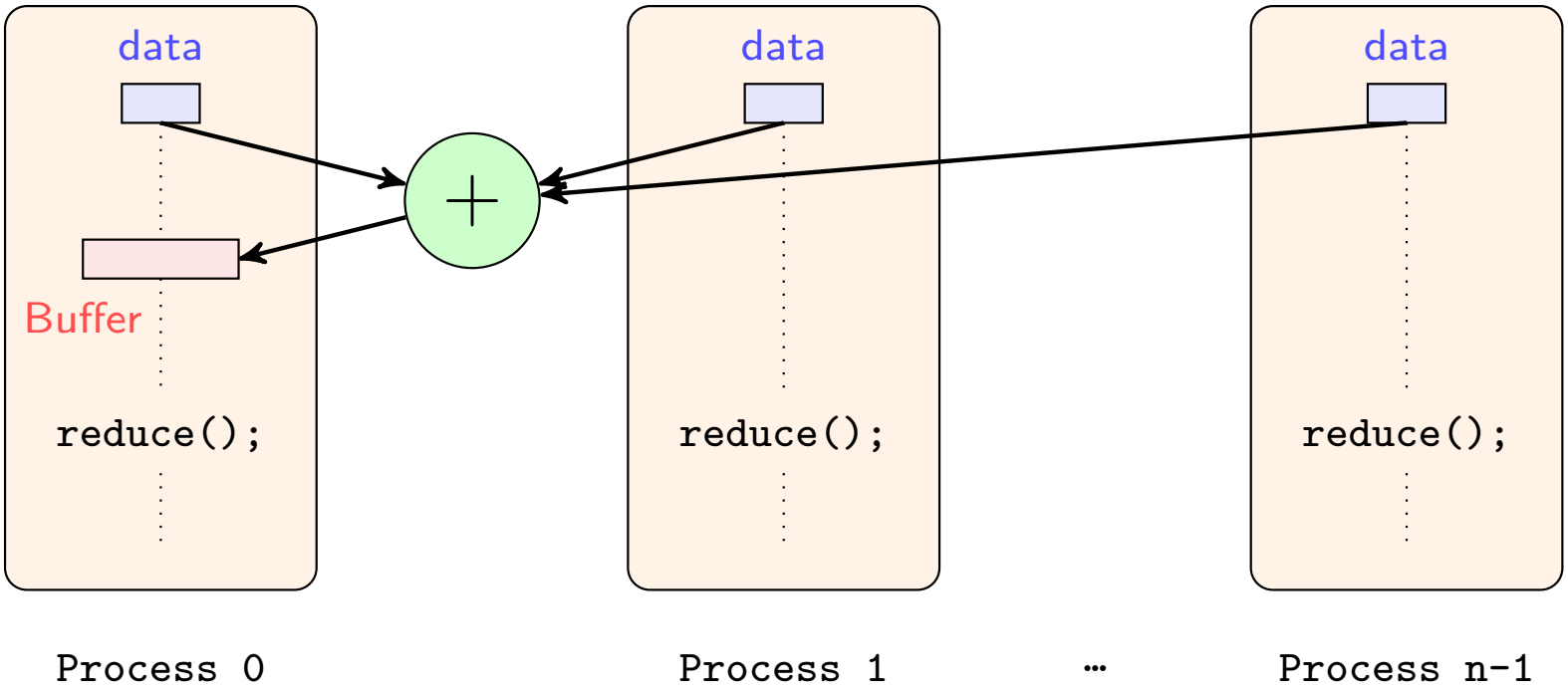
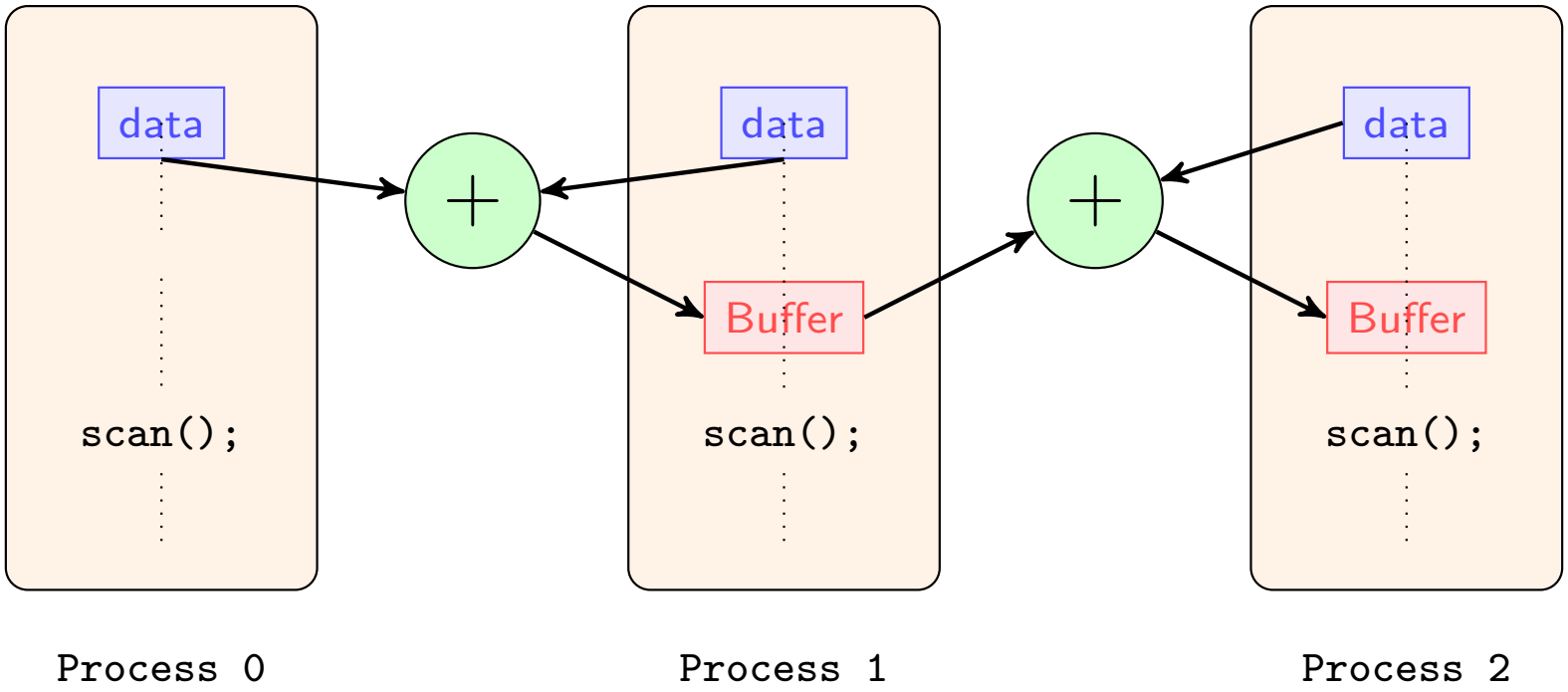


Figure: Schéma opératoire du rassemblement collectif

Réduction (reduce)



Scan


data
scan();
data
Buffer
scan();
data
Buffer
scan();
Process 0
Process 1
Process 2

Déboguer un programme parallèle

Si possible, d'abord exécuter le programme sur un seul processus, et le déboguer comme un programme séquentiel classique;

Exécuter le programme sur de deux à quatre processus sur un même nœud ou sur son ordinateur personnel. Vérifier que les messages sont bien envoyés au bon processus et qu'un processus reçoit bien le bon message. Il arrive souvent qu'il y ait une erreur sur l'identité du message ou bien des messages envoyés aux mauvais processus;

Exécuter maintenant le programme à l'aide de deux à quatre processus sur différents nœuds de calcul. Cela vous aidera à mesurer l'impact des délais dûs au réseau et à la synchronisation et ainsi mesurer le temps pris pour cela dans votre programme.

Algorithme parallèle à coût optimal

Définition

Algorithme parallèle vérifiant que :

$$\text{(Complexité en temps parallèle)} \times \text{(nombre de processus)} = \text{(complexité en temps séquentiel)}.$$

Example

Soit problème P avec complexité séquentielle en $O(n.\log(n))$.
Algorithme optimal Algorithme parallèle résolvant P avec n processus et coût de $O(\log(n))$ par processus.
Algorithme non optimal Algorithme parallèle résolvant P avec n^2 processus et coût en $O(1)$ sur chaque processus.

Accélération (speedup)

Définition

Soit

t_s : temps d'exécution séquentiel

$t_p(n)$: Temps d'exécution sur n nœuds de calcul;

L'accélération est alors défini par :

$$S(n) = \frac{t_s}{t_p(n)} \tag{1}$$

Remarque

Algorithme utilisé en séquentiel est souvent différent de celui utilisé en parallèle. Notion d'accélération est un notion très délicate :

L'algorithme utilisé en séquentiel est-il optimal ?

Le code séquentiel est-il bien optimisé, et exploite-t'il bien les mémoires caches ?

Loi d'Amdahl

$$S(n) = \frac{t_s}{f.t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1+(n-1)f} \quad (2)$$

t_s Temps passé en séquentiel

f Fraction (en temps) du code non parallélisable.

Permet de dimensionner le nombre de processus pour une application donnée.

Limitation

Cette loi est appliquée pour une application donnée, c'est à dire pour une taille de problème fixée.

Loi de Gustafson

Définition

Estimation de l'accélération en fonction du volume de donnée :

$$S_s(n) = \frac{s + n.p}{s + p} = s + n.p = n + (1 - n)s \tag{3}$$

s : proportion en temps du code exécuté en séquentiel;

p : proportion en temps du code exécuté en parallèle;

$p + s = 1, 0 \leq p, s \leq 1$

Diagramme espace–temps

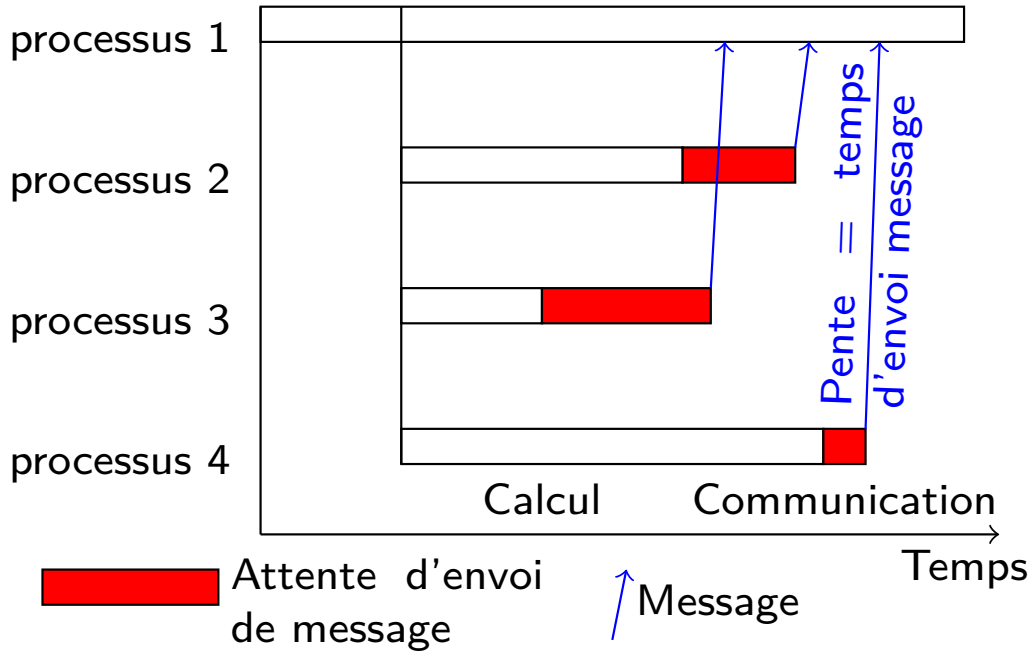


Figure: Exemple de diagramme espace–temps

Visualise les temps d'inaction des processus (synchronisation, attente de messages, etc.)

Efficacité

Définition

Fraction du temps durant lequel les nœuds sont utilisés pour traiter les données

$$E(n) = 100. \times \frac{t_s}{t_p(n) \times n} = 100 \times \frac{S(n)}{n} \quad (4)$$

Coût d'un calcul parallèle

Définition

Estimation du coût en énergie mais également pour la facturation des heures de calcul d'un programme parallèle en train de s'exécuter

$$C(n) = n \times t_p(n) = \frac{n.t_s}{S(n)} = \frac{t_s}{E(n)} \tag{5}$$

programme séquentiel

$$C(1) = 1 \times t_s = t_s \tag{6}$$

Coùt optimal si $E(n)$ est constant.

Scalabilité

Définition

Capacité du programme à avoir un bon speedup avec la montée en nombre de processus et une taille croissante de donnée à traiter

Évaluation de la scalabilité

En fait, on fera deux courbes de speed-up :

- L'une avec une taille fixe de donnée;
- L'autre avec une taille croissante de donnée en fonction du nombre de processus.

Équilibrage des charges

Définition

Tous les processus mettent le même temps de traitement de leurs données.

Example

Un programme s'exécute en t secondes, mais la moitié des processus mettent $\frac{t}{2}$ secondes pour finir leur traitement.
L'efficacité maximal d'un tel programme sera de 75%

Équilibrage dynamique des charges

Algorithme itératif : impossible de prédire le temps d'exécution de chaque processus (pour Mandelbrot par exemple);

Répartition des charges impossible à l'avance. Doit être décidé durant l'exécution du programme.

Stratégie maître–esclave :

- Un processus maître créee un ensemble de tâches pour résoudre le problème
- Il distribue une tâche sur chaque processus esclave.
- Dès qu'un processus esclave a fini sa tâche, il renvoie la solution;
- Le processus maître lui renvoie immédiatement la prochaine tâche non executée;
- Le processus maître envoie un signal de terminaison dès que toutes les tâches ont été exécutées.

Squelette algorithme maître–esclave

Processus maître : processus 0

```
MPI_Init();
...
if ( rank == 0 )// rank == 0 => master
{
  int count_task = 0;
  for ( int i = 1; i < nbp; ++i ) {
     send(count_task, 1, MPI_INT, i, ... );
     count_task += 1;
  }
  while (count_task < nb_tasks) {
    // status contiendra le numéro du proc ayant envoyé
    // le résultat... : status.MPI_SOURCE en MPI
    recv(&result, ..., MPI_ANY_SOURCE, ..., &status );
    send(count_task, 1, MPI_INT, result[0], ... );
    count_task += 1;
  }
  // On envoie un signal de terminaison à tous
  // les processus
  for ( int i = 1; i < nbp; ++i ) send(-1, 1, MPI_INT, i, ... );
}
```

Squelette algorithme maître–esclave

```
if (rank > 0)
{// Cas où je suis un travailleur
  int num_task = 0;
  // Tant que je ne reçois pas le n° de terminaison
  while (num_task != -1)
  {
    recv(&num_task, 1, MPI_INT, 0, ... );
    if (num_task >= 0) {
      // Exécute la tâche correspondant au numéro
      execute_task(num_task, ...);
      // Renvoie le résultat avec son numéro
      send(result, ..., 0, ... );
    }
  }
}
MPI_Finalize();
```

Granularité parallèle

Définition

Rapport du nombre d'opérations effectuées sur le nombre de données échangés

Quand ce nombre devient trop petit, le programme passe plus de temps à communiquer qu'à calculer $\Rightarrow$ mauvaise efficacité obtenue.
Antagoniste avec la notion d'équilibre des charges

Recouvrement des échanges de message par les calculs

En utilisant les messages asynchrones, on peut recouvrir une partie des échanges de message par des calculs.
Il faudra cependant à faire une synchronisation a postiori plus loin dans le code.