Programmation parallèle sur mémoire distribuée

Juvigny Xavier

October 8, 2021

Contents

1 Architecture des calculateurs à mémoire distribuée

Si une machine multi–cœur demande moins de changement drastique quant à l'architecture logicielle et aux algorithmes utilisés, elle requière néanmoins pour être efficace une organisation complexe de sa mémoire organisée en différents niveaux de mémoire cache dont le coût est sur-linéaire au nombre de cœurs utilisés.

Afin d'obtenir un calculateur performant sans être pour autant prohibitif, on optera plutôt pour une machine dont chaque unité de calcul (en général multi–cœurs mais sans un nombre prohibitif de cœurs) possède sa propre mémoire vive. On appelera un tel ensemble formé de cœurs de calcul et d'une mémoire vive s'appelle un **nœud de calcul**.

Ces différents nœuds de calcul échangent des données au travers d'un bus spécialisé ou plus souvent au travers d'un réseau de type Gigabit. On parle alors d'un calculateur à

mémoire distribuée. Ce type de calculateur a un coût de fabrication *linéaire* en fonction du nombre d'unités de calcul sur le réseau du calculateur.

La programmation de ces calculateurs demande une mise en œuvre spécifique et fréquemment des algorithmes spécifiques permettant d'exploiter efficacement la capacité de calcul de ces machines.

Ce type de calculateur permet d'exploiter simultanément plusieurs centaines de milliers d'unités de calcul. Cependant, la lenteur relative des réseaux ou des bus ainsi que le "degré de parallélisme" de l'algorithme employé peuvent limiter la rapidité de traitement des données

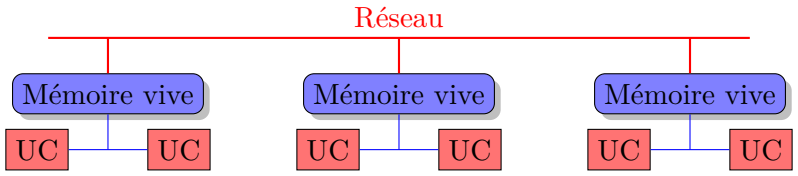


Figure 1: Architecture calculateur à mémoire distribuée

Les calculateurs que l'on peut trouver sur le marché offrent des topologies variées pour le réseau interconnectant les différents nœuds de calcul. Il est important lors d'une mise en œuvre sur un calculateur spécifique d'avoir en tête cette topologie. En effet, les données échangées au travers du réseau transiteront, selon la topologie du réseau, via d'autres nœuds de calcul, ce qui aura pour effet de ralentir les échanges de données en fonction du nombre d'unités de calcul intermédiaires par lesquelles les données transiteront.

Lorsqu'on exécute un algorithme parallèle qui n'exploite pas particulièrement la topologie du réseau du calculateur (par exemple si le logiciel exécuté a été conçu pour diverses plateformes), on peut s'employer à estimer la pénalité maximale qu'on risque de subir en terme de performance en calculant le nombre maximal de nœuds intermédiaires. Pour cela, nous allons définir pour un réseau la distance entre deux nœuds et son diamètre :

Définition 1 *On appelle* ***distance entre deux nœuds*** *sur un réseau donné le minimum de nœuds intermédiaires par lesquels doivent transiter des données echangées entre les deux nœuds.*

Définition 2 *On appelle* ***diamétre d'un réseau*** *la distance maximale entre deux nœuds du réseau.*

Pour la commodité de l'exposé, et en adéquation avec la vue programmeur fournie par les bibliothèques de calcul parallèle distribuée comme MPI, on numérote chacun des N nœuds de calcul de façon unique à l'aide d'un nombre entre 0 et $N-1$.

Passons en revue diverses topologies communément employées sur les calculateurs à mémoire distribuée.

1.1 Topologie linéaire

Les N nœuds de calcul sont reliés de tel manière qu'on peut trouver une numérotation de ces nœuds tel qu'un nœud i est lié directement aux seuls nœud $i-1$ (si $i > 0$) et $i+1$ (si $i < N-1$).

Un calcul rapide et trivial montre que le diamètre de ce réseau est de $N-1$.

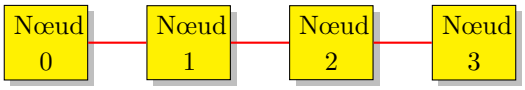


Figure 2: Réseau Linéaire

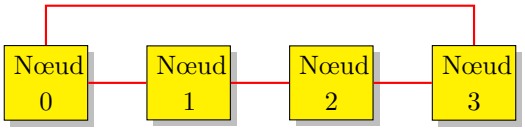


Figure 3: Réseau en anneau

1.2 Topologie anneau

La topologie de ce réseau est similaire à celui d'un réseau linéaire. Seule différence notable, le nœud 0 est relié au nœud numéroté $N-1$.

Le réseau étant bidirectionnel, le diamètre de ce réseau est $\frac{N}{2}$ si le nombre de nœud N est pair, $\frac{N-1}{2}$ sinon.

1.3 Topologie grille

Le réseau connectant les $N = W \times H$ nœuds est topologiquement équivalent à une grille de $W \times H$ nœuds.

Deux nœuds dont la distance est maximale sont le nœud zéro et le dernier nœud $N-1$. On en déduit le diamètre du réseau grille qui vaut donc $W + H - 2$.

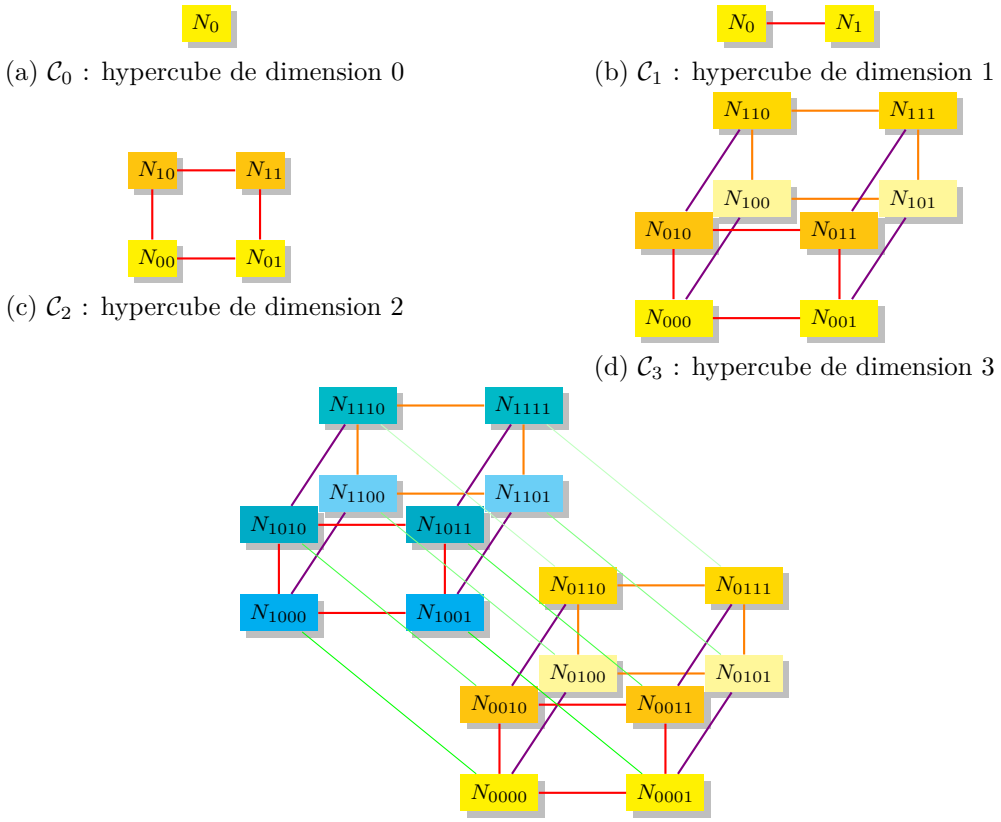
1.4 Topologie hyper cube

La topologie hyper cube est une topologie telle que la connection des nœuds de calcul deux à deux est équivalente aux arêtes reliant les sommets d'un hyper cube de dimension n.

Un hypercube de dimension n se construit de façon récursive en se basant sur l'hypercube de dimension $n-1$, en numérotant les nœuds en binaire et en remarquant que l'hypercube de dimension zéro est constitué d'un et un seul nœud numéroté avec la valeur zéro et que l'hypercube de dimension un est constitué de deux nœuds numérotés respectivement zéro et un, reliés par une connexion entre les deux nœuds (voir figure 4b)

Pour un hypercube $\mathcal{C}_n$ de dimension n supérieure à un, on considère l'hypercube $\mathcal{C}_{n-1}$ de dimension $n-1$ que nous allons dupliquer de la manière suivante :

- On effectue une première copie de $\mathcal{C}_{n-1}$ en rajoutant le chiffre zéro à gauche de la numérotation binaire. Par exemple, si dans $\mathcal{C}_{n-1}$, un nœud a pour numérotation 1001, il aura pour nouvelle numérotation 01001, et si un autre nœud a pour numérotation 0110, il aura pour nouvelle numérotation 00110.
- On effectue une deuxième copie de $\mathcal{C}_{n-1}$ en rajoutant le chiffre un à gauche de la numérotation binaire. Par exemple, si dans $\mathcal{C}_{n-1}$, un nœud a pour numérotation 1001, il aura pour nouvelle numérotation 11001, et si un autre nœud a pour numérotation 0110, il aura pour nouvelle numérotation 10110.
- On relie ensuite deux nœuds appartenant chacun à une copie différente de $\mathcal{C}_{n-1}$ par une connexion directe si et seulement si leur numérotation dans $\mathcal{C}_{n-1}$ est la même. Ainsi, si on considère le nœud 01001 de la première copie et le nœud 11001 de la deuxième copie, ils sont tous les deux issus du nœud 1001 de $\mathcal{C}_{n-1}$, et doivent donc



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

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

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

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

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

Figure 4: Réseau hypercube de dimensions 0 à 4. Les nœuds sont numérotés en binaire selon la numérotation de Gray.

être reliés par une connexion directe. En revanche, les nœuds 01001 et 10110 ne sont pas issus du même nœud de $\mathcal{C}_{n-1}$ et ne doivent donc pas être reliés.

Par exemple, pour construit $\mathcal{C}_2$, l'hypercube de dimension deux, on duplique l'hypercube de dimension un qui contient deux sommets connectés 0 et 1, en un premier hypercube de dimension un contenant les sommets connectés 00 et 01 et en deuxième hypercube de dimension un contenant les sommets connectés 10 et 11. On relie maintenant le sommet 00 avec le sommet 10 et le sommet 01 avec le sommet 11. On obtient bien in fine un carré (hypercube de dimension deux, voir figure 4c).

Pour les dimensions supérieures, on reitère la procédure pour obtenir des hypercubes de dimension 3, 4 ou supérieure, comme sur les figures 4d et 4e.

La numérotation obtenue en appliquant l'algorithme de construction des hypercubes se nomme le *code Gray* (sur n bits, n étant la dimension de l'hypercube). Cette numérotation possède de nombreuses propriétés intéressantes, dont les propriétés suivantes, qu'on peut démontrer trivialement à partir de l'algorithme de construction :

Propriétés 1 *Deux nœuds d'un hypercube $\mathcal{C}_n$ de dimension n sont directement connectés par un arête si et seulement si leurs numérotations binaires de diffère que d'un seul bit.*

Par exemple, dans un hypercube $\mathcal{C}_4$ de dimension quatre, les nœuds numérotés *0101* et *0111* ont une arête les reliant tandis que les nœuds numérotés *0101* et *0110* n'ont pas de connection directe.

On en vient donc à la propriété suivante :

Propriétés 2 *La distance entre deux nœuds i et j dans un hypercube de dimension n est le nombre de bits différents dans leur numérotation binaire.*

Par exemple, dans un hypercube $\mathcal{C}_6$ de dimension six, la distance entre les nœuds (numérotés en binaire) *0*1*10*0*1 et *1*10*0*11 est de 3 (trois bits de différents entre les deux numéros).

On en déduit directement le diamètre du réseau hypercube :

Propriétés 3 *Un réseau hypercube de dimension n (contenant 2^n nœuds de calcul) a un diamètre de n.*

Les réseaux hypercubes sont des réseaux proposant des connections très riches, et permettant d'obtenir de bonnes performances en terme d'échanges de données et de résistance aux pannes réseaux sans pour autant avoir un coût trop élevé (contrairement à un graphe complet (un cluster) où chaque nœud est relié à tous les autres).

2 Les différents modèles de programmation parallèle

À l'instar du classement architectural des machines parallèles, il existe un classement software de la programmation parallèle.

- **SPSD** (**S**imple **P**rogram **S**imple **D**ata)
  C'est la programmation classique séquentielle où on ne traite qu'une seule donnée à la fois à l'aide d'un seul programme.
- **SPMD** (**S**imple **P**rogram **Multiple Data**)
  On exécute le même programme, ou on exécute la même fonction dans le cas des threads, sur plusieurs données simultanément. Attention, cela est différent d'une programmation sur machine SIMD puisque les différents processus peuvent ne pas exécuter les mêmes instructions. C'est de loin le modèle de programmation le plus utilisé pour les algorithmes parallèles sur machines à mémoire distribuée.
- **MPMD** (**M**ultiple **P**rogram **M**ultiple **D**ata)
  Plusieurs programmes (ou plusieurs fonctions) sont exécutés simultanément sur plusieurs données. Dans le cadre de la programmation partagée, cela peut être utilisé pour programmer une interface graphique exécutée sur un thread tandis que les autres threads effectuent des calculs par exemple, ou bien en mémoire distribuée pour faire un client/serveur.

Remarque 1 *Il ne faut surtout pas confondre le SIMD qui est une contrainte hardware avec le SPMD qui est un modèle de programmation !*

Remarquons par ailleurs que le modèle de programmation MPMD peut toujours être ramener à un modèle SPMD en écrivant un programme unique :

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

3 Programmation parallèle sur machines à mémoire distribuée

Dans ce modèle de programmation, plusieurs processus font tourner simultanément des programmes, identiques ou non, possédant chacun leur propre espace mémoire privé (en

général une mémoire physique différente mais on peut très bien avoir des processus qui se partagent une même mémoire physique).

Il est impossible à un processus d'aller lire ou écrire sur l'espace mémoire d'un autre processus. Les processus doivent donc pouvoir communiquer à l'aide de messages pour s'échanger des données.

3.1 Contexte et communication point à point

Pour cela, un processus devant envoyer un message à un autre processus doit pouvoir désigner le processus destinataire d'une manière ou d'une autre. On doit donc étiqueter chaque processus à l'aide d'un identifiant unique avant toute opération parallèle. Il faut donc initialiser un contexte global dupliqué sur chaque processus permettant d'attribuer à chacun un identifiant unique, en général un entier compris entre zéro et le nombre de processus utilisés moins un.

Il se peut également qu'un processus reçoit des données provenant de plusieurs autre processus. Il est alors nécessaire de distinguer la provenance de ces messages, c'est-à-dire de pouvoir identifier l'expéditeur du message.

De plus, il arrive souvent qu'un processus envoie plusieurs messages à un même destinataire. Or sur le réseau internet, le protocole ne garantit pas que l'ordre d'envoi des messages soit respecté à la réception. Pour identifier les messages provenant d'un même expéditeur, il est nécessaire de rajouter une étiquette à chaque message envoyé.

Enfin, en plus des données et de leur quantité, il est important dans le cas où on exécute un programme parallèle sur un ensemble de machines hétérogènes de préciser le type des données échangées. En effet, certains types de scalaires sont représentés de manière différents selon les processeurs employés. Ainsi, sur les processeurs Intel, les entiers sont représentés en little endian (les octets de poids faibles en premiers puis ceux de poids croissants) tandis que sur Arm les entiers sont représentés en big endian (les octets de poids forts en premiers puus ceux de poids décroissants). Il faut donc préciser le type de données qu'on envoi afin qu'une éventuelle conversion soit faite.

En résumé :

1. Chaque processus lancé en début d'une session d'exécution parallèle se voit attribué à l'initialisation du contexte un identifiant unique (en général un entier);
2. Un message dans un contexte calcul distribué doit contenir :
   - L'identifiant de l'expéditeur ou du destinataire (selon qu'on reçoit ou on envoi);
   - Un identifiant pour le message;
   - Le type des données envoyées/reçues;
   - La quantité de données à envoyer;
   - Les données elles même.

3.2 Protocole des envois et des réceptions

Un échange de données entre deux processus se décompose en deux phases :

1. Une phase de "dialogue" entre les deux processus pour convenir d'un protocole commun entre les deux machines pour échanger des données. Cette première phase a un coût constant quelque soit la taille du message;
2. La phase d'échange proprement dite. Le coût de l'échange est linéaire par rapport à la taille (en octets) du message et dépend du débit soutenu par le réseau.

Au final, le coût d'un message en fonction de la taille en octet du message est (en secondes):

$$T(n) = T_{\text{Startup}} + \frac{n}{\text{Débit}} \tag{1}$$

Cette formule n'est valable que si les deux processus sont directement reliés par une connection éthernet. Dans le cas où la distance entre les deux processus est supérieure à un, il faut compter le temps de dialogue avec les nœuds intermédiaires et le temps de transfert des messages par ces nœuds.

Sur un réseau éthernet classique, le coût en temps d'un échange peut varier d'une exécution à l'autre. Des collisions de messages ou des corrections d'erreur font varier le coût d'un échange et la formule n'est utile qu'à titre indicatif dans ce contexte.

Enfin, remarquons que lorsqu'on fait tourner un programme parallèle sur une seule machine constituée de plusieurs cœurs en y lançant plusieurs processus simultanément, l'échange de message se fait via la mémoire et non internet. Les temps d'envoie et de réceptions y seront donc bien moins pénalisant que si on exécuter notre programme sur des nœuds de calcul différents.

La bibliothèque MPI utilisée en TD propose plusieurs protocoles d'envoi/réceptions.

Envoi/Réception bufférisée Lorsqu'on envoi un tableau, afin de s'assurer que le programmeur ne modifie pas le tableau durant le temps d'envoi, on recopie le tableau dans un *buffer* qui sera ensuite envoyé au processus destinataire. C'est ce qui est improprement appelé **message asynchrone** par la bibliothèque MPI mais qu'on appelera ici **message bufférisé**.

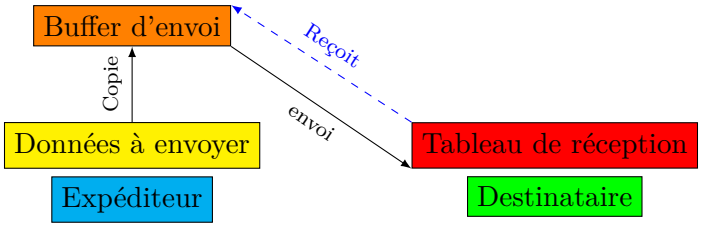


Figure 5: Schéma d'envoi bufférisé

Si cela garantit l'intégrité des données envoyées au processus destinataire, cela a néanmoins un coût (allocation mémoire plus copie de données) et il est possible dans ce cas d'effectuer un envoi non bufférisé. Il faut alors s'assurer que le programme n'écrit pas de nouvelles données dans le tableau envoyé tant que celui-ci n'a pas été envoyé.

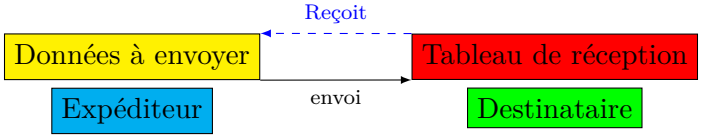


Figure 6: Schéma d'envoi non bufférisé

Envoi/Réception synchrone/asynchrone Lors de l'envoi de données ou de leur réception, le programme peut attendre que l'envoi soit terminer avant de rendre la main

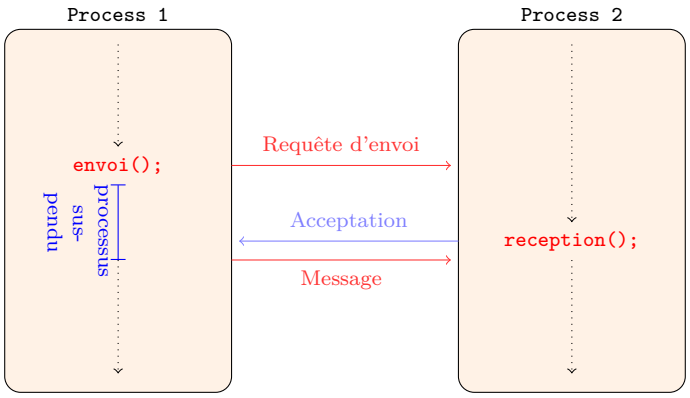


Figure 7: Exemple message synchrone quand la réception est exécutée après l'envoi

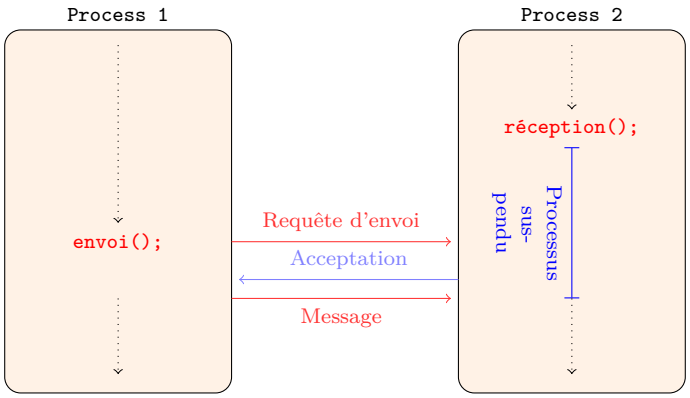


Figure 8: Exemple message synchrone quand la réception est exécutée avant l'envoi

pour le reste du programme, on parle alors d'envoi ou de réception **synchrone** (bloquant dans le jargon MPI) ou bien rendre tout de suite la main.

Dans le cas d'un message non bloquant, il est nécessaire en amont du code de s'assurer que l'envoi (ou la réception) a bien été effectuée. Pour cela on disposera de fonction permettant :

- Soit de tester si l'envoi ou la réception a bien été effectuée;
- Soit d'attendre que l'envoi (ou la réception) soit effectuée;

Dans le premier cas, cela permet en attendant que l'envoi (ou la réception) se termine de continuer à effectuer des calculs indépendants des données envoyées ou reçues. Cela permet en particuliers de recouvrir les temps de transferts des données entre processus par des calculs et obtenir un gain de temps non négligeable du temps d'exécution du code.

Cela permet en outre d'éviter des cas d'**interblocage** (*deadlock* en anglais) : c'est à dire des cas où :

- tous les processus envoient un message bloquant à un autre processus avant d'effectuer une réception;
- Ou le cas symétrique où tous les processus attendent un message en réception avant d'effectuer un envoi.

Un exemple d'interblocage a été donné par Dijkstra :

> Cinq philosophes aimeraient réfléchir ou manger des spaghetti autour d'une table circulaire ayant cinq assiettes et cinq fourchettes utilisées quand un philosophe a faim.
>
> Manger des spaghetti demande aux philosophes d'utiliser les fourchettes des deux côtés (?!).
>
> Un philosophe repose ses deux fourchettes quand il n'a plus faim.
>
> Une situation d'interblocage apparaît dans ce cas quand tous les philosophes présents tiennent une fourchette dans leur main droite.

Les problèmes d'**interblocage**, assez fréquents au demeurant dans des applications réelles, sont des bogues difficiles à trouver, qui gênent l'exécution des processus indéfiniment si on ne les tue pas manuellement.

Exemple d'interblocage sur deux processus :

```
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 l'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 );
}
```

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 );
}
```

Par défaut, dans la bibliothèque MPI, l'envoi est bufférisée et asynchrone si la quantité de donnée n'est pas trop importante tandis que la réception est synchrone et non bufférisée.

Exercice : : Expliquez pourquoi le code MPI ci dessous n'est pas sûr et peut par moment conduire à un interblocage et à un autre moment se terminer normalement :

```
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 );
}
```

4 Communications collectives

Une opération de communication collective est une opération d'échange de message mettant en jeu tous ou une partie des processus appartenant au contexte parallèle.

La majorité de ces opérations peuvent se programmer à l'aide des échanges de message point à point, mais l'optimisation des échanges de message pour ces types d'opérations vont dépendre fortement de la topologie du réseau, ce qui nuit à la portabilité du code sur d'autres machines aux topologies réseaux différentes. Aussi des bibliothèques comme MPI proposent ces opérations dans leurs APIs en garantissant que les échanges de message seront adaptés à la topologie réseau sur lequel le code sera exécuté en parallèle.

Les communications colllectives peuvent se ranger en trois catégories :

1. Effectuer une barrière de synchronisation;
2. 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.
3. 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, c'est à dire une opération cumulative sur les données réparties sur les processus.

4.1 Barrière de synchronisation globale

Cette opération crée un point de rendez-vous pour tous les processus dans le code. Les processus attendent que tous les autres processus soient arrivés sur la barrière de synchronisation pour continuer ensuite l'exécution du programme.

Cette opération de synchronisation est particulièrement utile pour pouvoir mesurer des temps de calcul d'une partie d'un programme parallèle :

Exemple en MPI :

```
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();
...
```

4.2 Mouvement collectif de données

Diffusion (BCast) La diffusion consiste à ce qu'un processus envoie les mêmes données à tous les autres processus.

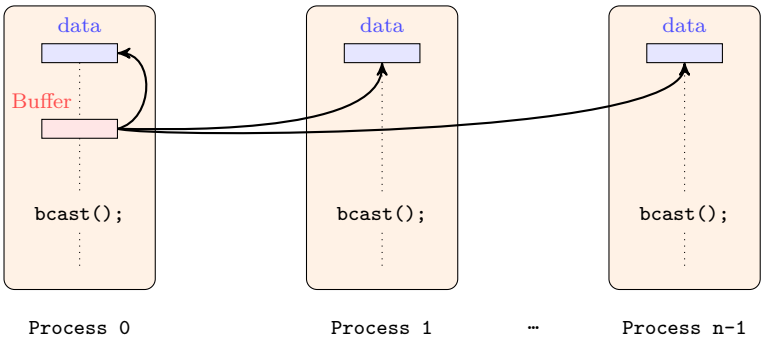


Figure 9: Principe de la diffusion

Répartition (Scatter) L'opération collectif de répartition consiste à répartir les données d'un tableau appartenant à un processus sur tous les processus (processus contenant le tableau compris) de telle sorte que la $i^{\text{ème}}$ partie du tableau ira sur le processus numéro i.

Rassemblement (Gather) C'est l'opération inverse de la répartition collective : le processus destinataire rassemble dans un tableau les données envoyées par tous les processus (la donnée du $i^{\text{ème}}$ processus va à la $i^{\text{ème}}$ position dans le tableau).

Réduction L'opération de réduction combine l'opération de rassemblement avec une opération logique ou arithmétique : le processus destinataire collecte les données puis applique l'opération et stocke le résultat dans son propre espace mémoire.

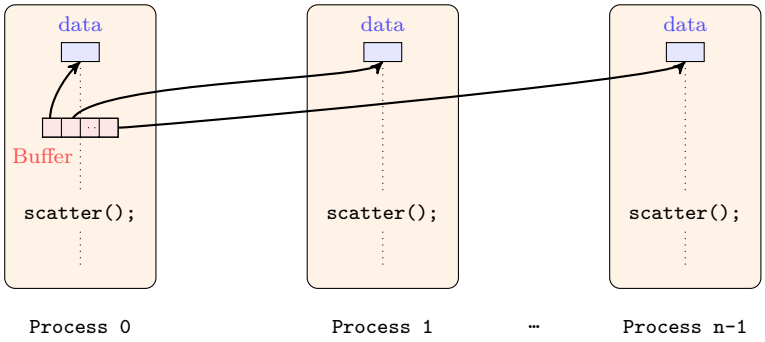


Figure 10: Schéma du fonctionnement d'une répartition collective

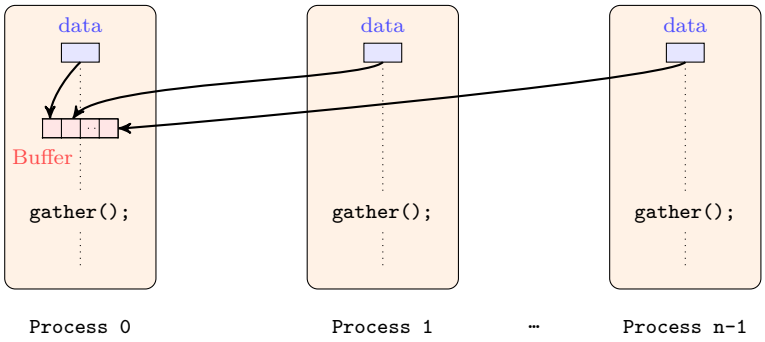


Figure 11: Schéma opératoire du rassemblement collectif

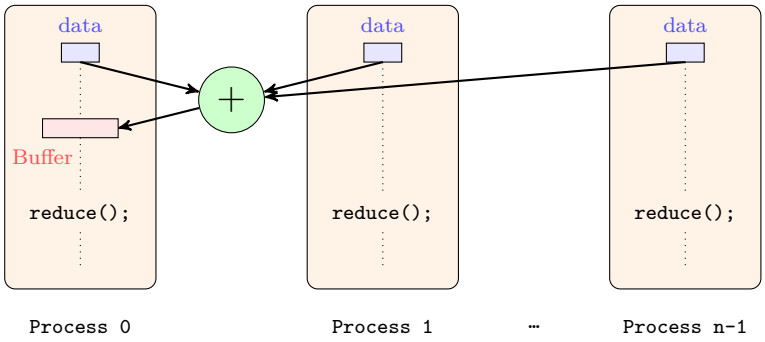


Figure 12: Exemple illustratif d'une réduction avec somme

Il existe dans certaines bibliothèques, comme MPI, une instruction combinant une opération de réduction suivi d'une opération de diffusion sur tous les processus.

Scan L'opération de scan est une opération proche de l'opération de réduction, mis à part que l'opération se fait par accumulation de processus en processus (ordonné selon leur rang).

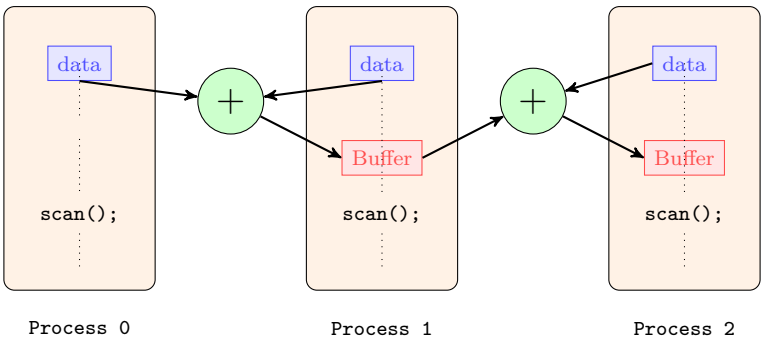


Figure 13: Illustration d'un scan avec somme

5 Développement et Mesures de performances

5.1 Une stratégie de débogage

Déboguer un programme en parallèle est une tâche bien plus ardue qu'en séquentiel ! Une approche *efficace* en trois étapes pour déboguer des programmes faisant de la communication point à point est :

1. Si possible, d'abord exécuter le programme sur un seul processus, et le déboguer comme un programme séquentiel classique;
2. 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;
3. 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.

5.2 Algorithme parallèle à coût optimal

Un algorithme à coût optimal est un algorithme tel que :

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

Exemple : Supposons qu'un problème P a une complexité en $O(n.\log(n))$. Un algorithme parallèle résolvant le même problème, utilisant n processus pour un coût de $O(\log(n))$ en complexité sur chaque processus est optimal en coût, tandis qu'un algorithme parallèle utilisant n^2 processus avec un coût en $O(1)$ sur chaque processus n'est pas optimal en coût.

5.3 Estimation du coût d'envoi/réception par une méthode de ping–pong

Pour estimer empiriquement le temps de communication entre un processus P_1 et un processus P_2, une solution consiste à suivre la méthode suivante : **Immédiatement** après que P_2 ait reçu un message envoyé par P_1, il renvoie le message à P_1 et on mesure le temps d'aller retour de ce message :

Exemple :

```
double t1 = get_time();
if (myrank == P1) {
    send(&x, P2); recv(&x, P2);
} else if (myrank == P2) {
    recv(&x, P1); send(&x, P1);
}
double t2 = get_time();
elapsedTime = 0.5*(t2-t1);
```

5.4 La loi d'Amdahl

La loi d'Amdahl est une loi émise par l'informaticien Gene Amdahl travaillant chez Intel en 1967 pour promouvoir les architectures séquentielles à l'époque. Elle permet pour un

problème fixe donné, de calculer le rapport de temps que l'on peut espérer gagner quelque soit le nombre de processus employés.

La loi d'Amdahl peut être formulée de la façon suivante :

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

où t_s est le temps pris pour exécuter le programme en séquentiel, f est la proportion du programme (en temps CPU séquentiel) qui ne peut être parallélisé et fonctionne donc en séquentiel, n le nombre de processus utilisés en parallèle pour exécuter le programme.

D'après cette formule, on observe qu'en prenant une infinité de nœuds de calcul, le rapport de temps maximal espéré sera limité à $\frac{1}{f}$.

Exemple : En supposant qu'un programme ait 5% de son code (en temps CPU séquentiel) non parallélisable, le rapport de temps maximal gagné sera inférieur à vingt.

Cette loi est surtout utile pour dimensionner le nombre de processus qu'on utilisera pour exécuter un code en parallèle.

5.5 Loi de Gustafson

Si la loi de d'Amdahl permet de prédire un gain maximal sur le rapport de temps pris entre la version séquentielle du code et la version parallèle, elle n'est cependant valable que pour une taille fixe de problème.

Or, en réalité, le nombre de processus qu'on prendra pour exécuter un code parallèle dépendra fortement de la taille du problème à traiter.

L'informaticien John L. Gustafson et son collègue Edwin H. Barsis, en 1988, dans un article intitulé *Reevaluating Amdahl's Law* ont émis une nouvelle loi dans le cadre où la quantité de donnée traitée en parallèle augmente avec le nombre de processus utilisés (ce qui est vérifié dans la plupart des cas).

La loi de Gustafson peut être formulée comme suit :

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

où s est la proportion du code en temps CPU exécuté en séquentiel et p la proportion de code exécuté en parallèle (en temps CPU) : $0 < s, p < 1$ et $s + p = 1$.

La loi de Gustafson donne des résultats plus réalistes que la loi d'Amdahl et souvent meilleurs !

5.6 Mesures de performances

L'accélération (speedup) : L'accélération est une mesure permettant d'estimer le degré de parallélisme de votre code.

Soit t_s le temps pris pour exécuter la version séquentielle du programme sur un simple processus et $t_p(n)$ le temps pris pour exécuter la version parallèle du programme sur n nœuds de calcul.

L'accélération est alors défini par :

$$S(n) = \frac{t_s}{t_p(n)} \tag{4}$$

Remarque : En général, l'algorithme utilisé en séquentiel est différent de celui utilisé en parallèle. La notion d'accélération est une 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 ?

Par définition, l'accélération est toujours un nombre réel inférieur au nombre de processus, mais il arrive, du fait de la partition du problème à traiter et des mémoires caches qu'on obtienne une surlinéarité en traçant la courbe du speedup. Cela indique dans ce cas que l'optimisation du code séquentiel a été mal faite et qu'on peut encore gagner du temps CPU en gérant mieux les mémoires caches.

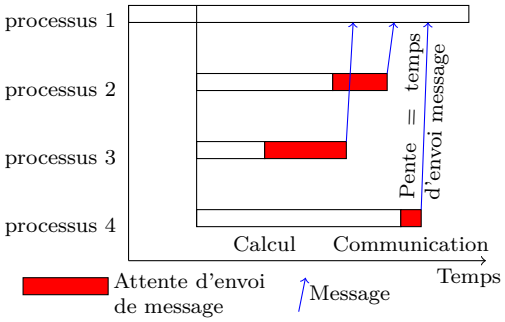


Figure 14: Exemple de diagramme espace–temps

Diagramme espace–temps Le diagramme d'espace–temps permet de visualiser facilement les temps d'attente (en rouge sur la figure (14)) des différents processus pour envoyer ou attendre un message sans effectuer de calcul. Le but sera lors de la phase d'optimisation de minimiser ces zones rouges afin d'obtenir de meilleurs performances et un temps de réponse réduit pour le code optimisé.

Efficacité L'efficacité donne la fraction de temps durant laquelle les processeurs sont utilisés pour traiter des données (et non à transférer des données ou à attendre une synchronisation).

$$E(n) = 100. \times \frac{t_s}{t_p(n) \times n} = 100 \times \frac{S(n)}{n} \tag{5}$$

Coût d'un calcul parallèle Cette mesure permet à des administrateurs de calculateurs parallèle d'évaluer le coût et la facture d'un cacul parallèle. Elle est donnée par la formule :

$$C(n) = n \times t_p(n) = \frac{n.t_s}{S(n)} = \frac{t_s}{E(n)} \tag{6}$$

Dans le cas d'un calcul séquentiel, le coût du calcul revient à

$$C(1) = t_s \times 1 = t_s$$

Un algorithme parallèle à coût optimal est un algorithme traitant un problème en parallèle sur plusieurs processus tel que le coût de traitement est proportionnel au traitement en séquentiel (autrement dit, $E(n)$ est constant).

Scalabilité Cette mesure observe la situation où on augmente les données à traiter tout en augmentant le nombre de processus permettant de les traiter. On mesura l'accélération du programme en augmentant la quantité de données en fonction du nombre de processus exécutés.

5.7 Critères pour avoir de bonnes performances

Équilibre des charges : Il est important lors de la mise au point de programmes parallèle de s'assurer que tous les processus traitent des données tout le long de l'exécution du programme. En effet, il peut arriver dans certaines situations, que certains processus finissent longtemps avant les autres. Dans ce cas, il est certain qu'on obtiendra une mauvaise accélération (ou une mauvaise efficacité).

Exemple : Pour un problème donné, un programme est executé sur n processus. On observe que la moitié des processus mettent deux fois moins longtemps pour traiter leurs données que l'autre moitié. Soit t_p^i le temps mis par le processus de range i pour traiter ses données propres en parallèle et t_s le temps mis par le programme séquentiel (supposé optimal) pour traiter les données en séquentiel. On sait que (sinon le programme séquentiel ne serait pas optimal) :

$$\sum_{i=1}^{n} t_p^i \geq t_s$$

Si on note t_p le temps maximal mis par un processus pour traiter ses données, on aura donc d'après l'observation faite sur le temps pris par chaque processus que :

$$\frac{n}{2}\frac{t_p}{2} + \frac{n}{2}t_p \geq t_s$$

Si bien que l'efficacité maximal d'un tel programme, sans compter les temps de communication et les synchronisations, vaudra :

$$E = \frac{t_s}{n.t_p} \leq \frac{3}{4} = 75\%$$

Il est donc primordial de s'assurer que tous les processus auront à peu près la même charge de travail afin d'obtenir une bonne efficacité.

Il arrive toutefois qu'on soit dans l'incapacité de connaître en avance la charge de travail de chacun des processus (en particuliers pour des algorithmes itératifs dont on ne connait pas d'avance le nombre d'itération nécessaire à faire sur chaque ensemble de données).

On utilise dans ce cas un algorithme permettant d'assurer dynamiquement l'équilibre des charges entre les processus. L'algorithme le plus utilisé pour cela est l'algorithme de *task farming*. Le principe est de diviser le problème en un certain nombre de tâches indépendantes qui doivent être exécutées. Un des processus (qu'on nommera **le maître**) est responsable de générer ces tâches et de les distribuer parmi tous les processus travailleurs libres (dont il peut éventuellement lui même faire parti). Chaque processus exécute une certaine tâche et renvoie le résultat au processus maître. En retourn le processus maître génère une nouvelle tâche qu'il envoie au travailleur qui vient de terminer sa tâche. Ce procédé continue jusqu'à ce qu'il n'y ai plus de tâches à faire, et que le problème est donc résolu.

```
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, status.MPI_SOURCE, ... );
    count_task += 1;
  }
  // On envoie un signal de terminaison à tous
  // les processus
  count_task = -1;
  for ( int i = 1; i < nbp; ++i ) {
     send(&count_task, 1, MPI_INT, i, ... );
  }
} else
{// 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();
```

Figure 15: Algorithme de task farming

Granularité parallèle : On a vu qu'un envoi de message engendré un coût fixe d'initialisation et un coût linéaire par rapport à la quantité de donnée envoyée (dépendant du débit supporté par le réseau). Si on divise trop le travail entre les processus, le temps de communication deviendra trop importante par rapport à la quantité de travail que devra faire un processus entre deux échanges de message. Au final, on risque de passer plus de temps à échanger des messages qu'à traiter les données par notre programme.

On appelle granularité parallèle le rapport entre la quantité de donnée traitée et la quantité de données échangées.

Il faut donc veiller à ne pas trop découper notre problème. Le problème étant, qu'avoir de trop gros blocs de données à traiter peut être antagoniste avec le problème d'équilibre de charge. Il faut donc trouver un juste milieu dans la répartition des données permettant à la fois un équilibrage des charges raisonnable et une granularité parallèle.

Recouvrement des échanges de message par des calculs Enfin, signalons que pour obtenir une bonne performance de notre code parallèle, on peut toujours utiliser les envois/réceptions asynchrones afin de couvrir nos échanges de message par des calculs sur d'autres données, non échangées à ce moment là.