Travaux dirigés n°2

Xavier JUVIGNY

19 novembre 2020

Table des matières

1 Une introduction à MPI (suite)

1.1 Les fonctions de communication collectives

Il existe trois sortes de fonctions de communication collectives :

1. **Synchronisation** : Les tâches attendent que les autres tâches du groupe aient atteint le point de synchronisation ;
2. **Échanges de données** : Diffusion, regroupement, partition, ...
3. **Réduction** : Calcul collectif : une tâche du groupe collecte toutes les données des autres membres et effectue une opération sur ces données (min, max, opération arithmétique, etc.).

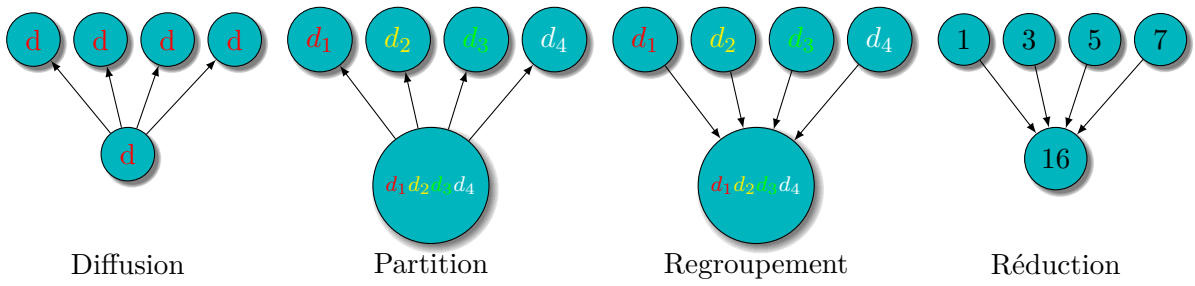


FIG. 1 : Les différents échanges de messages collectifs

Les communications collectives mettent en œuvre toutes les tâches appartenant à un même groupe de communication

- Toutes les tâches lancées par l'utilisateur en utilisant `MPI_COMM_WORLD` ;
- Un sous-ensemble de tâches si on a défini un nouveau communicateur ne prenant en compte que ce sous-ensemble

Des comportements aléatoires peuvent avoir lieu, dont un plantage du code, si même une seule tâche du communicateur n'appelle pas la fonction.

C'est de la responsabilité du programmeur de s'assurer que toutes les tâches du communicateur utilisé appellent la fonction de communication collective.

Les communications collectives ne prennent pas d'identificateur en arguments.

Si on veut exécuter une communication collective sur un sous-ensemble de tâches, il faut tout d'abord partitionner un groupe de tâches en sous-ensembles puis associer ces groupes à des communicateurs (voir

À partir de la norme MPI 3 on peut choisir entre des communications collectives bloquantes ou non bloquantes.

Fonctions de communications collectives

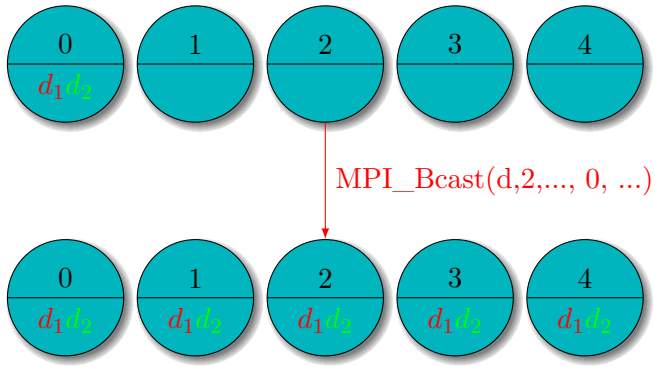
- MPI_Barrier

  Opération de synchronisation. Crée une barrière de synchronisation dans un groupe. Chaque tâche, lorsqu'elle atteint l'appel au `MPI_Barrier`, bloque jusqu'à ce que toutes les tâches du groupe atteignent le même appel à `MPI_Barrier`. Alors toutes les tâches peuvent continuer à travailler.

  ```
  MPI_Barrier (comm)
  ```

- MPI_Bcast

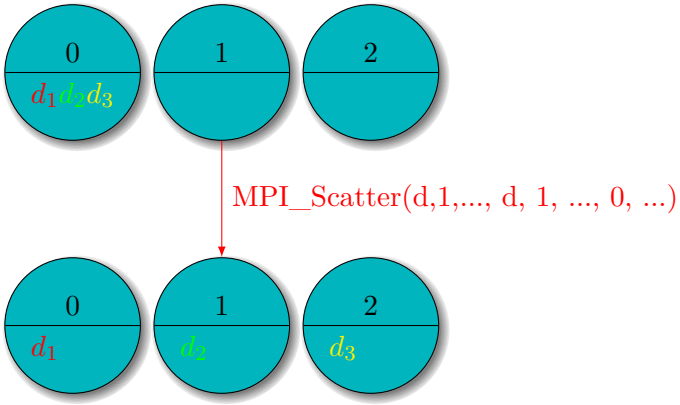
  Opération de mouvement de données. Diffuse (envoie) un message de la tâche ayant le rang "root" à toutes les autres tâches du groupe.




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

- MPI_Scatter

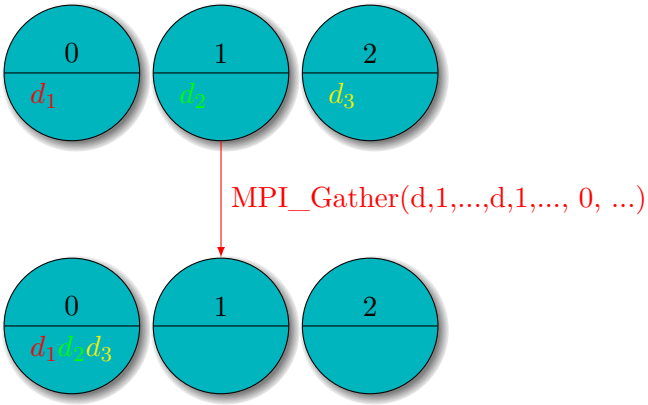
  Opération de mouvement de données. Distribue (en partitionnant) des données distinctes d'une seule tâche source "root" vers chaque tâche du groupe.




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

- MPI_Gather

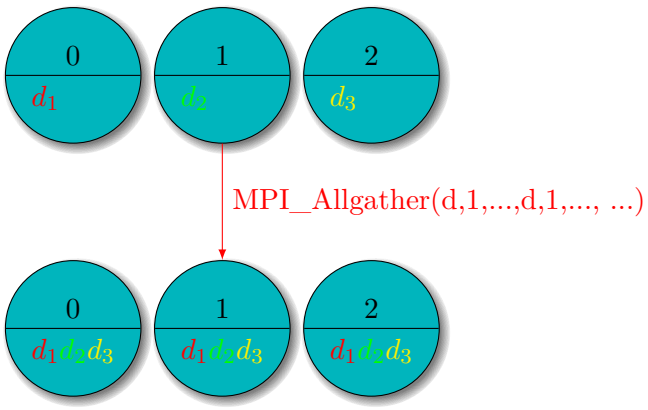
  Opération de mouvement de données. Rassemble différentes données provenant de chaque tâche du groupe dans une seule tâche destinataire "root". Cette fonction est l'opération inverse de `MPI_Scatter`.



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

- `MPI_Allgather`

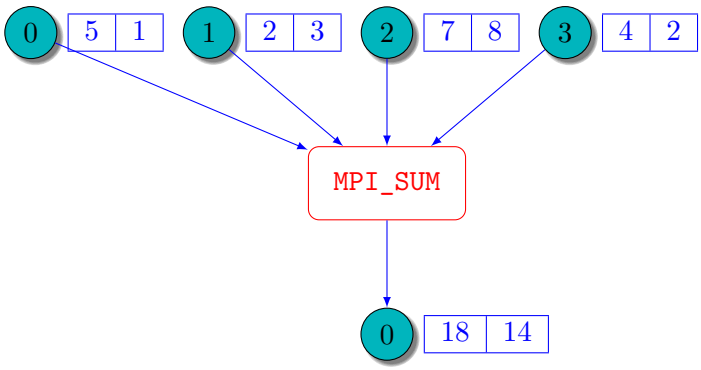
  Opération de mouvement de données. Concaténation des données sur toutes les tâches dans un groupe. Chaque tâche dans le groupe, en effet, effectue une diffusion une tâche vers toutes dans le groupe.




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

- `MPI_Reduce`

  Opération de calcul collectif. Applique une opération de réduction sur toutes les tâches du groupe et place le résultat dans une tâche.



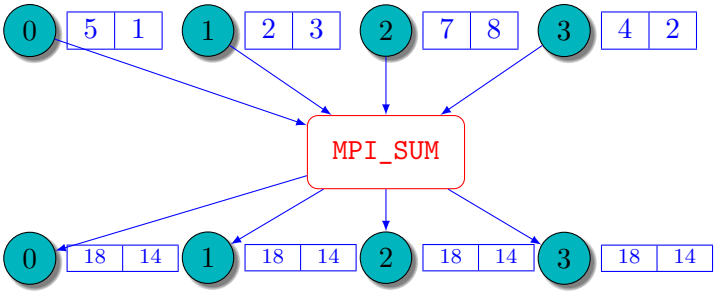

```
MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm )
```

Les opérations de réduction MPI prédéfinies sont données dans la table ci-dessous. Le programmeur peut également définir ses propres fonctions de réduction à l'aide de la fonction `MPI_Op_create`.

Opération de réduction MPI		Type de données C
`MPI_MAX`	Maximum	Entiers, Réels
`MPI_MIN`	Minimum	Entiers, Réels
`MPI_SUM`	Somme	Entiers, Réels
`MPI_PROD`	Produit	Entiers, Réels
`MPI_LAND`	ET logique	Entiers
`MPI_BAND`	ET par bit	Entiers, `MPI_BYTE`
`MPI_LOR`	OU logique	Entiers
`MPI_BOR`	OU par bit	Entiers, `MPI_BYTE`
`MPI_LXOR`	OU exclusif logique	Entiers
`MPI_BXOR`	OU exclusif bit à bit	Entiers, `MPI_BYTE`
`MPI_MAXLOC`	Valeur maximale et location	float, double et long double
`MPI_MINLOC`	Valeur minimale et location	float, double et long double

- `MPI_Allreduce`

  Opération de calcul collectif et de déplacement de données. Applique une opération de réduction et diffuse le résultat dans toutes les tâches du groupe. Cette fonction est équivalente à appeler une fonction de réduction suivi d'un appel à `MPI_Bcast`.




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

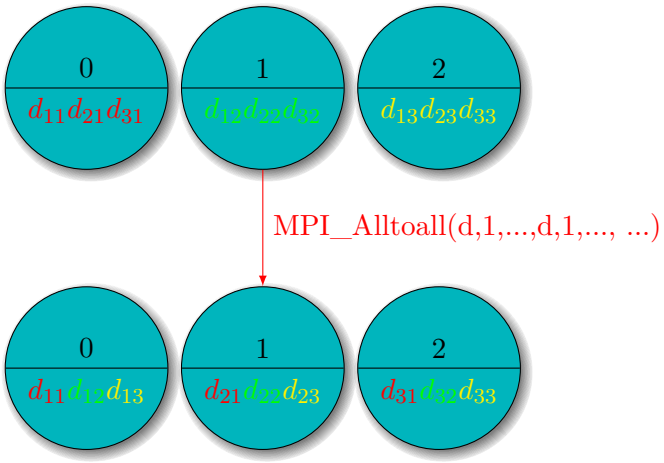
- MPI_Reduce_scatter

  Opération de calcul collectif et de déplacement de données. Effectue en premier une réduction élément par élément sur un vecteur sur toutes les tâches du groupe puis partitionne le vecteur résultat pour le distribuer parmi les tâches du groupe.

```
MPI_Reduce_scatter (&sendbuf, &recvbuf, recvcnt, datatype, op, comm )
```

- MPI_Alltoall

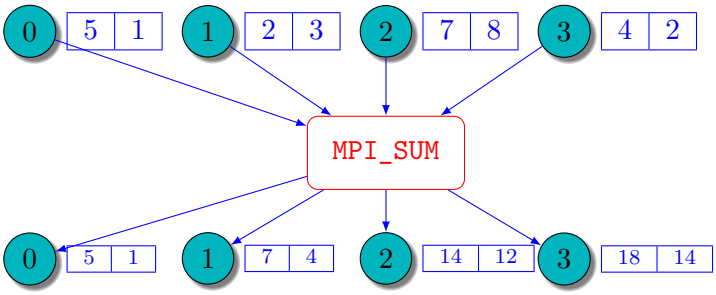
  Opération de mouvement de données collective. Chaque tâche dans le groupe effectue une opération de diffusion, envoyant un message distinct à toutes les tâches du groupe dans l'ordre par index.




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

- MPI_Scan

  Effectue une opération de balayage par rapport à une opération de réduction au travers d'un groupe de tâches.




```
MPI_Scan ( &sendbuf, &recvbuf, count, datatype, op, comm )
```

Exemple : Distribue les lignes d'un tableau

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[]) {
```

```
  int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0},
    {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0},
    {13.0, 14.0, 15.0, 16.0}  };
  float recvbuf[SIZE];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  if (numtasks == SIZE) {
    // define source task and elements to send/receive, then perform collective scatter
    source = 1;
    sendcount = SIZE;
    recvcount = SIZE;
    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
                MPI_FLOAT,source,MPI_COMM_WORLD);

    printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
           recvbuf[1],recvbuf[2],recvbuf[3]);
    }
  else
    printf("Must specify %d processors. Terminating.\n",SIZE);

  MPI_Finalize();
}
```

1.2 Fonctions de gestion de groupes et de communicateurs

Groupes et communicateurs

- Un groupe est un ensemble ordonné de tâches. On associe un rang unique à chaque tâche dans le groupe. La valeur du rang est comprise entre 0 et N-1 où N est le nombre de tâches dans le groupe. Dans MPI, un groupe est représenté par un objet en mémoire uniquement accessible par le programmeur à l'aide d'une "poignée". Un groupe est toujours associé avec un objet communicateur.
- Un communicateur englobe un groupe de tâches qui vont devoir communiquer entre elles. Tout message MPI doit spécifier un communicateur. Dans le sens le plus simple, le communicateur est un identifiant supplémentaire obligatoire pour les messages MPI. Comme les groupes, les communicateurs sont representés comme des objets uniquement accessibles par le programmeur à l'aide d'une "poignée". Par exemple, la poignée pour le communicateur contenant toutes les tâches est `MPI_COMM_WORLD`.
- Du point de vue du programmeur, un groupe et un communicateur sont identiques. Les fonctions de manipulation de groupes sont uniquement utilisées pour désigner les tâches servant à construire le communicateur.

Utilités primaires des groupes et des communicateurs

1. Vous permet d'organiser les tâches selon des fonctions en groupes de tâches ;
2. Permet d'utiliser les communications collectives sur un sous-ensemble de tâches ;
3. Permet des communications sécurisées.

Considération de programmation et restrictions

- Les groupes/communicateurs sont dynamiques : ils ne peuvent être créés ou détruits que pendant l'exécution du programme.
- Les tâches peuvent appartenir à plusieurs groupes/communicateurs. Elles auront un identifiant de rang unique pour chaque groupe/communicateur.
- Plus de quarante fonctions sont disponibles pour manipuler les groupes et les communicateurs.

- Programmation typique :
  1. Récupérer la poignée du groupe global associé à `MPI_COMM_WORLD` en faisant appel à la fonction `MPI_Comm_group` ;
  2. Créer un nouveau groupe en tant que sous-ensemble du groupe global en appelant `MPI_Group_incl` ;
  3. Créer un nouveau communicateur pour le nouveau groupe en utilisant `MPI_Comm_create` ;
  4. Déterminer le rang de la tâche dans le nouveau communicateur en utilisant `MPI_Comm_rank` ;
  5. Utiliser le nouveau communicateur dans votre programme ;
  6. Quand le nouveau communicateur n'est plus utilisé, détruire le nouveau communicateur et éventuellement son groupe associé en appelant `MPI_Comm_free` et `MPI_Group_free`.
- Pour effectuer une partition des tâches au travers de plusieurs communicateurs, on utilisera plutôt `MPI_Comm_split`.

Exemple : Créer deux communicateurs : un pour les tâches de rang pair dans le communicateur global et l'autre pour les tâches de rang impair.

```c
#include "mpi.h"
#include <stdio.h>

int  main(int argc, char *argv[])  {
   int        rank, new_rank, sendbuf, recvbuf, numtasks;
   MPI_Comm   new_comm;    // required variable

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   if (numtasks%2 == 2 )
     MPI_Comm_split( MPI_COMM_WORLD, 0, rank, &new_comm );
   else
     MPI_Comm_split( MPI_COMM_WORLD, 1, rank, &new_comm );

   sendbuf = rank;

   MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

   // get rank in new communicator
   MPI_Comm_rank (new_comm, &new_rank);
   printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

   MPI_Finalize();
   return 0;
}
```

2 Exercices à rendre

2.1 Question du cours

Reprendre l'exercice sur l'interblocage donné dans le cours et décrivez deux scénarios :

1. Un premier scénario où il n'y a pas d'interblocage ;
2. Un deuxième scénario où il y a interblocage.

Quelle est à votre avis la probabilité d'avoir un interblocage ?

2.2 Question du cours n°2

Alice a parallélisé en partie un code sur machine à mémoire distribuée. Pour un jeu de données spécifiques, elle remarque que la partie qu'elle exécute en parallèle représente en temps de traitement 90% du temps d'exécution du programme en séquentiel.

En utilisant la loi d'Amdhal, pouvez-vous prédire l'accélération maximale que pourra obtenir Alice avec son code (en considérant $n \gg 1$) ?

À votre avis, pour ce jeu de donné spécifique, quel nombre de nœuds de calcul semble-t-il raisonnable de prendre pour ne pas trop gaspiller de ressources CPU ?

En effectuant son calcul sur son calculateur, Alice s'aperçoit qu'elle obtient une accélération maximale de quatre en augmentant le nombre de nœuds de calcul pour son jeu spécifique de n données.

En doublant la quantité de données à traiter, et en supposant la complexité de l'algorithme parallèle linéaire, quelle accélération maximale peut espérer Alice en utilisant la loi de Gustafson ?

2.3 Ensemble de mandelbrot

L'ensemble de Mandelbrot est un ensemble fractal inventé par Benoit Mandelbrot permettant d'étudier la convergence ou la rapidité de divergence dans le plan complexe de la suite récursive

$$\begin{cases} c \text{ valeurs complexe donnée} \\ z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

en fonction de c.

On montre facilement que si il existe un N tel que $|z_N| > 2$, alors la suite diverge.

L'ensemble de Mandelbrot consiste à calculer une image de $W \times H$ pixels telle qu'à chaque pixel (p_i, p_j) de l'espace image, on associe une valeur complexe $c = x_{\min} + p_i \frac{x_{\max} - x_{\min}}{W} + i \left(y_{\min} + p_j \frac{y_{\max} - y_{\min}}{H} \right)$.

1. À partir du code séquentiel `mandelbrot.cpp`, faire une partition équitable par ligne de l'image à calculer pour distribuer le calcul sur les `nbp` tâches exécutées par l'utilisateur puis rassembler l'image sur le processus zéro pour la sauvegarder. Calculer le temps d'exécution pour différents nombre de tâches et calculer le speedup. Comment interpréter les résultats obtenus ?

2. Mettre en œuvre une stratégie maître–esclave pour distribuer les différentes lignes de l'image à calculer. Calculer le speedup avec cette nouvelle approche. Qu'en conclure ?

2.4 Produit matrice–vecteur

On considère le produit d'une matrice carrée A de dimension N par un vecteur u de même dimension dans $\mathbb{R}$. La matrice est constituée des cœfficients définis par $A_{ij} = (i + j) \mod N$

Par soucis de simplification, on supposera que N est divisible par le nombre de tâches `nbp` exécutées.

1. **Produit parallèle matrice – vecteur par colonne**

   Afin de paralléliser le produit matrice–vecteur, on décide dans un premier temps de partitionner la matrice par un découpage par bloc de colonnes. Chaque tâche contiendra N_{loc} colonnes de la matrice. Calculer en fonction du nombre de tâches la valeur de N_{loc} puis paralléliser le code séquentiel `matvec.cpp` en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à sa somme partielle du produit matrice-vecteur. On s'assurera que toutes les tâches à la fin du programme contiennent le vecteur résultat complet.

2. **Produit parallèle matrice – vecteur par ligne**

   Afin de paralléliser le produit matrice–vecteur, on décide dans un deuxième temps de partitionner la matrice par un découpage par bloc de lignes. Chaque tâche contiendra N_{loc} lignes de la matrice. Calculer en fonction du nombre de tâches la valeur de N_{loc} puis paralléliser le code séquentiel `matvec.cpp` en veillant à ce que chaque tâche n'assemble que la partie de la matrice utile à son produit matrice-vecteur partiel. On s'assurera que toutes les tâches à la fin du programme contiennent le vecteur résultat complet. .