# Systèmes Distribués

## *Travaux pratiques*

**Institut Galilée — Département Informatique — M1 informatique**

Camille Coti
camille.coti@lipn.univ-paris13.fr

# 1 Premiers pas avec MPI

### Exercice 1.1 : Configuration de votre environnement

Pour préparer votre environnement pour les TP de MPI, vous avez besoin de configurer votre compte de manière à pouvoir utiliser SSH sans mot de passe.

Générez une paire de clés publique/privée avec `ssh-keygen`. Tapez entrer à toutes les questions qu'on vous pose (clés à mettre dans les fichiers par défaut, passphrase vide). Un répertoire `~/.ssh` est alors créé, contenant notamment le fichier contenant votre clé privée et votre clé publique. Mettez votre clé publique dans un fichier `~/.ssh/authorized_keys`.

```
$ ssh-keygen -t dsa
[...]
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Il faut maintenant que vous puissiez vous connecter à d'autres machines sans que SSH vous demande si vous acceptez la clé des machines. Ajoutez la ligne suivante dans un fichier `~/.ssh/config`.

```
StrictHostKeyChecking no
```

### Exercice 1.2 : Exécution d'un programme sur plusieurs machines

En utilisant `mpiexec`, exécutez le programme `hostname` en parallèle sur 4 machines différentes.

### Exercice 1.3 : Exécution d'un code C

Copiez le code suivant dans votre éditeur de texte préféré, et sauvegardez-le dans un fichier `.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main( int argc, char** argv ) {
    int size, rank, len;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Get_processor_name( name, &len );
    fprintf( stdout, "Je suis le processus %d parmi %d sur %s\n",
                                rank, size, name);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

1. Compilez le code.
2. Exécutez le programme obtenu sur 8 processus sur 4 machines différentes.

3. Avec `mpiexec -help`, quelle est la différence entre les options `-bycore`, `-bynode` et `-byslot` ? Vérifiez leur fonctionnement sur votre programme.

## Exercice 1.4 : Exécution d'un code Python

Copiez le code suivant dans votre éditeur de texte préféré, et sauvegardez-le dans un fichier `.py`.

```
#!/usr/bin/env python

from mpi4py import MPI

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

print "Je suis le processus", rank, "parmi", size, "sur", name
```

Ou, si vous utilisez Python 3 :

```
#!/usr/bin/env python

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Je suis le processus %d parmi %d sur %s.\n"
    % (rank, size, name))
```

Exécutez-le sur 8 processus sur au moins 4 machines.

## Exercice 1.5 : Un bug !

Examinez le programme suivant.

```
#!/usr/bin/env python

from mpi4py import MPI
import numpy

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
comm = MPI.COMM_WORLD

if rank%2:
    dest = rank - 1
else:
    dest = rank + 1
```

```
14
15 print "Je suis le processus", rank, "parmi", size, "et j'envoie a",
       dest
16
17 data = numpy.arange(1000, dtype='i') # Creation d'un tableau de
       taille 1000
18 comm.send( data, dest=dest, tag=42 )
```

1. Que fait-il ?
2. Quel est le problème ?
3. Quelle serait la modification que vous y feriez pour le faire fonctionner ?

# 2 Communications point-à-point en MPI

### Exercice 2.1 : Ping-pong

1. Créez un fichier contenant le code du ping-pong vu dans le cours
2. Compilez-le avec `mpicc`
3. Exécutez-le sur plusieurs machines de la salle de TP

### Exercice 2.2 : Communications non-bloquantes

1. Reprenez le ping-pong de l'exercice précédent et modifiez-le pour utiliser des communications non-bloquantes `MPI_Isend` et `MPI_Irecv`.
2. Où placez-vous les `MPI_Wait` ?
3. Modifiez le code pour effectuer non pas un ping-pong mais un envoi simultané. Auriez-vous pu le faire avec des communications bloquantes ?

### Exercice 2.3 : Anneau

Prenons un ensemble de processus MPI communiquant en suivant une topologie d'anneau.

1. Si un processus de cet ensemble a le rang $k$, quel est le rang du processus à qui $k$ va envoyer des messages ?
2. Quel est le rang du processus de qui $k$ va recevoir des messages ?
3. Comment la circulation du jeton doit-elle être initiée ?
4. Comment la circulation du jeton se termine-t-elle ?
5. Implémentez en MPI une circulation de jeton (un entier) selon un anneau entre des processus. Votre programme doit fonctionner quel que soit le nombre de processus.

# 3 Types de données

## Exercice 3.1 : Type dérivé I

1. Créez un type dérivé en MPI contenant un ensemble d'entiers contigus
2. Écrivez un programme où deux processus s'échangent un tableau d'entiers, en utilisant le type que vous venez de définir. Le nombre d'éléments envoyés doit être égal à 1, le type de données est votre type dérivé.

## Exercice 3.2 : Type dérivé II

1. Créez un type dérivé en MPI contenant des entiers non-contigus :
   — un entier par bloc
   — le stride (écart entre le début d'un bloc et le début du suivant) est égal à un paramètre N
   — le nombre de blocs est égal à un nombre M
2. Écrivez un programme où deux processus disposent chacun d'une matrice de M lignes et N colonnes stockée linéairement dans un tableau à une dimension. Ces processus s'échangent les premiers éléments de chaque ligne :
   (a) Implémentez-le avec des communications envoyant les éléments un par un. Combien de messages cette méthode implique-t-elle ? Quel est le volume de données (total et par message) envoyées ?
   (b) Implémentez-le en utilisant le type dérivé que vous venez de définir. Comparez le nombre de message et le volume de données transportées avec l'implémentation précédente.

## Exercice 3.3 : Transposition de matrice

Le but de cet exercice est de transposer efficacement une matrice distribuée sur des processus parallèles.

1. Écrivez un programme parallèle avec MPI dans lequel deux matrices sont réparties sur les processus en étant stockées linéairement dans des tableaux à une dimension. La première matrice est composée de M lignes et N colonnes, la deuxième est composée de N lignes et M colonnes. La matrice est distribuée par décomposition de domaine selon les lignes : ainsi, chaque processus a une matrice (M/P)xN et une matrice (N/P)xM. Pour des raisons de simplicité, prenez N=M=P. Ainsi, chaque processus dispose d'une ligne de chaque matrice.
2. En utilisant les types dérivés définis dans les exercices précédents, modifiez votre programme afin que chaque processus envoie le premier élément de chaque ligne de sa matrice au processus 0, qui stocke tous ces éléments de façon contiguë dans la ligne de la matrice dont il dispose. Pour cela, vous utiliserez une opération collective en étant particulièrement attentif aux types de données utilisés en envoi et en réception (indication : ils ne sont pas identiques).
3. En utilisant une autre opération collective, modifiez votre programme pour que les $k$-ièmes éléments de la matrice soient rassemblés sur le processus $k$. L'opération effectuée sur la matrice est une transposition.

4. Pour utiliser une matrice de taille quelconque, quelle doit être la taille des matrices contenues sur chaque processus ?
5. Quels doivent être les paramètres des types de données dérivés qui doivent alors être utilisés (nombre d'éléments, stride, nombre de blocs) ?
6. Modifiez votre programme pour transposer des matrices de taille quelconque.

## Exercice 3.4 : Maître-esclave statique

Implémentez un squelette de maître-esclave statique en utilisant des opérations collectives pour distribuer les données et récupérer les résultats.

## Exercice 3.5 : Maître-esclave dynamique

Implémentez un squelette de maître-esclave dynamique en mode pull : les esclaves demandent du travail au maître quand ils envoient un nouveau résultat ou à l'initialisation, et celui-ci leur envoie des données (un tableau d'entiers) tant qu'il en a.

# 4 Communications collectives en MPI

### Exercice 4.1 : Diffusion

Écrivez un programme en MPI dans lequel le processus de rang 0 récupère son *process id* et le diffuse à tous les autres processus. Les processus doivent alors tous afficher cette valeur.

### Exercice 4.2 : Réduction

Écrivez un programme en MPI qui se comporte comme suit :

1. Chaque processus alloue un tableau de 1014 éléments
2. Chaque processus remplit le tableau avec des valeurs aléatoires comprises entre 0 et 100 (attention à l'initialisation de votre générateur de nombres aléatoires, qui doit être seedée différemment sur chaque processus)
3. Chaque processus calcule la valeur maximum contenue dans ce tableau
4. En utilisant une opération collective, récupérez le maximum des maxima locaux dans le processus de rang 0.

### Exercice 4.3 : Scatter-Gather

**Question 1 :** Écrivez un pogramme en MPI qui se comporte comme suit :

1. Le processus de rang 0 alloue un tableau dont le nombre d'éléments est égal au nombre de processus multiplié par 1024
2. Il remplit alors le tableau de valeurs aléatoires comprises entre -100 et 100
3. En utilisant une opération collective, ce tableau est distribué à tous les processus du système
4. Chaque processus a alors une portion du tableau. Chacun élève au cube toutes les valeurs comprises dans ce tableau.
5. Chaque processus calcule la moyenne contenue dans son tableau.
6. En utilisant une opération collective, récupérez l'ensemble des moyennes dans un tableau contenu dans le processus de rang 0 (attention à l'allocation mémoire).
7. Le processus de rang 0 calcule la moyenne des moyennes récupérées.

**Question 2 :** Comparez le temps de calcul de l'implémentation parallèle que vous venez d'écrire avec une implémentation séquentielle.

### Exercice 4.4 : Implémentation de la diffusion

**Question 1 :** En utilisant uniquement des opérations d'envoi et de réception, implémentez la diffusion d'un entier en utilisant l'algorithme de l'étoile.

**Question 2 :** Même question en utilisant un arbre binomial.

## Exercice 4.5 : Implémentation de la distribution

**Question 1 :** En utilisant uniquement des opérations d'envoi et de réception, implémentez la distribution d'un entier en utilisant l'algorithme linéaire.

**Question 2 :** Même question en utilisant un arbre binomial.

**Question 3 :** Même question en utilisant un arbre binaire.

## Exercice 4.6 : Implémentation du all-to-all

En utilisant uniquement des opérations d'envoi et de réception (vous pourrez utiliser des communications non-bloquantes), implémentez un all-to-all où chaque processus envoie un entier différent à chaque autre processus.

## Exercice 4.7 : Normalisation d'une matrice

Le but de cet exercice est de normaliser les données d'une matrice, c'est-à-dire de diviser la valeur de tous ses éléments par la valeur du plus grand élément.

1. Créez un programme parallèle avec MPI dans lequel les processus disposent chacun d'une sous-matrice aléatoire de taille MxN.
2. Sur chaque processus, recherchez la valeur du plus grand élément de la sous-matrice.
3. En utilisant une opération collective, calculez la valeur du maximum global. Tous les processus doivent avoir cette valeur.
4. Sur chaque processus, divisez la valeur des éléments de la sous-matrice par le maximum global.

# 5 Communications unilatérales

### Exercice 5.1 : Intermédiaire

Écrivez un programme qui s'exécute avec au moins trois processus. Le processus 0 initialise un entier avec son `pid`. Le processus 1 va le chercher avec un `MPI_Get` et l'écrit dans la mémoire du processus 2 avec un `MPI_Put`.

### Exercice 5.2 : Anneau à jeton

Écrivez une variante de l'anneau à jeton utilisant des communications unilatérales. Attention : n'oubliez pas qu'un processus ne sait pas si un autre processus a lu ou écrit dans sa mémoire. Vous devrez donc mettre en place un mécanisme de synchronisation.

### Exercice 5.3 : Aléa reproductible

Les générateurs de nombres aléatoire sont le plus souvent en réalité pseudo-aléatoire : les nombres générés dépendent de l'état du générateur. C'est pour cela qu'on doit toujours initialiser le générateur avec une *graine*.

Le but de cet exercice est de générer le même tableau pseudo-aléatoire dans tous les processus. Pour cela, il faut qu'ils initialisent tous leur générateur de nombres aléatoires avec la même graine. Il y a trois façons d'implémenter ceci.

La première consiste à générer un tableau sur un processus et le diffuser aux autres processus. L'inconvénient est que si le tableau est très gros, les communications peuvent être très longues.

La deuxième solution consiste à générer la graine sur un processus et la diffuser à tous les autres processus, qui initialisent leur générateur de nombres aléatoires et génèrent chacun un tableau aléatoire. L'inconvénient de cette approche réside dans son caractère synchrone : il faut que tous les processus soient arrivés à ce point à ce moment de l'exécution du programme.

Vous allez implémenter la troisième solution. Un processus génère la graine et la stocke dans sa mémoire accessible par les autres processus. Les autres processus lisent la graine et génèrent leur tableau.

Quel est l'inconvénient de cette approche par rapport à la deuxième solution ?

### Exercice 5.4 : Maximum irrégulier

Le but de cet exercice est de calculer le maximum global d'un tableau distribué irrégulier. Chaque processus dispose d'un tableau aléatoire de taille différente (en pratique, vous tirerez la taille aléatoirement entre 100 et 100 000). Chaque processus calcule le maximum de son tableau et le processus 0 calcule le maximum des maxima.

Une façon simple d'implémenter un maximum global consiste à calculer le maximum local sur chaque processus et effectuer une réduction avec `MPI_MAX` vers un processus racine, qui disposera alors du maximum global. Cependant, dans le cas présent, le calcul est trop irrégulier, et les processus rapides devront attendre les processus les plus lents lors de la communication collective.

Vous pouvez alors utiliser des communications unilatérales : un processus qui a terminé écrit

son résultat dans le processus 0 avec `MPI_Accumulate`, et il passe au suivant. Attention, pour que le processus 0 sache que le calcul est terminé, il doit savoir combien de processus ont écrit : pour cela, vous pourrez utiliser un compteur qui est incrémenté également avec `MPI_Accumulate`.

# 6 OpenMP I

### Exercice 6.1 : Prise en main et compilation

Dans cet exercice, vous pourrez partir d'un code `Hello world` fourni dans le cours.

1. Compilez le code OpenMP (n'oubliez pas l'option qui active OpenMP)
2. En utilisant la variable d'environnement idoine, contrôlez le nombre de threads utilisés
3. Faites la même chose en utilisant la fonction idoine de la bibliothèque.

### Exercice 6.2 : Calcul du maximum global

Le but de cet exercice est d'implémenter le calcul du maximum global d'un tableau en parallèle sur plusieurs threads. Vous pourrez partir d'un tableau initialisé avec des nombres aléatoires compris entre 0 et 100.

```
1 void inittable( int* tab, int taille ) {
2     int i;
3     for( i = 0 ; i < taille ; i++ ) {
4         tab[i] = (int)((double)rand()*MAX / (double) RAND_MAX );
5     }
6 }
```

En n'oubliant pas d'initialiser votre générateur de nombres aléatoires :

```
1 srand( getpid() );
```

Nous avons vu dans le cours qu'il était plus efficace de calculer ce maximum global en deux étapes :

— Calcul des maxima locaux
— Calcul du maximum des maxima

Vous pourrez initialiser le maximum global et les maxima locaux avec la première case du tableau.

Implémentez cet algorithme avec une section parallèle et une boucle parallèle. Attention, n'oubliez pas que le calcul du maximum des maxima introduit une situation de concurrence.

### Exercice 6.3 : Ordre et ordonnancement

La construction `ordered` permet de s'assurer que les threads sont exécutés dans l'ordre de la boucle. Implémentez un programme où vous paralléliserez la boucle suivante :

```
1 for( i = 0 ; i < numthreads; i++ ) {
2     printf("Hello World from thread = %d \n", tid );
3 }
```

Si `numthreads` désigne le nombre de threads et `tid` l'identifiant du thread. Ainsi, votre programme affiche cette ligne autant de fois qu'il a de threads.

1. Avec la construction `ordered`, assurez-vous que l'affichage est effectué dans l'ordre des threads.

2. Que se passe-t-il si vous passez en ordonnancement statique et en jouant sur la taille des tranches ?

### Exercice 6.4 : Section critique

Examinez le code suivant.

```
1
2  void lafonction( int *data ) {
3    /* Calcul */
4    #pragma omp critical( sectioncritique )
5    {
6      data = rand();
7    }
8  }
9
10 int main(){
11   int data1, data2;
12   #pragma omp parallel
13   {
14     lafonction( &data1 );
15     /* Un peu de calcul */
16     lafonction( &data2 );
17   }
18
19   return EXIT_SUCCESS;
20  }
```

1. Quel est le problème avec ce programme ? Fonctionnera-t-il correctement ? Aura-t-il de bonnes performances ?
2. Proposez une autre implémentation qui résout ce problème.

### Exercice 6.5 : Élimination gaussienne

L'élimination gaussienne est une technique de diagonalisation de matrice permettant la résolution d'un système d'équations linéaires. Elle consiste à éliminer la partie qui se situe sous la diagonale d'une matrice colonne par colonne.

Prenons la matrice suivante : $\begin{pmatrix} 3 & 5 & 7 \\ 2 & 4 & 8 \\ 1 & -2 & 3 \end{pmatrix}$

Nous allons chercher à éliminer la première colonne, en utilisant la première ligne comme pivot. En multipliant la troisième ligne par 3 et la deuxième par $3/2$, on obtient :

$$\begin{pmatrix} 3 & 5 & 7 \\ 3 & 6 & 12 \\ 3 & -6 & 9 \end{pmatrix}$$

Si l'on soustrait la première ligne des suivantes, on obtient :

$$\begin{pmatrix} 3 & 5 & 7 \\ 0 & 1 & 5 \\ 0 & -11 & 2 \end{pmatrix}$$

Vous constatez que les éléments de la première colonne situés sous la diagonales ont été mis à 0. Occupons-nous maintenant de la deuxième colonne. On utilise maintenant la deuxième ligne comme pivot : il faut multiplier les éléments de la troisième ligne par -1/11.

$$\begin{pmatrix} 3 & 5 & 7 \\ 0 & 1 & 5 \\ 0 & 1 & -2/11 \end{pmatrix}$$

On soustrait maintenant la deuxième ligne de la troisième :

$$\begin{pmatrix} 3 & 5 & 7 \\ 0 & 1 & 5 \\ 0 & 0 & -57/11 \end{pmatrix}$$

Nous avons bien éliminé les éléments sous-diagonaux de la matrice. Il est possible d'effectuer la même chose pour mettre à 0 la partie supérieur de la matrice et obtenir une matrice diagonale : c'est l'élimination de Gauss-Jordan.

L'algorithme se fait donc de la façon suivante :
— Pour chaque colonne, effectuer ce qui suit pour chaque ligne située sous la diagonale.
— Calcul du ratio entre les deux lignes, en utilisant le terme diagonal concerné.
— Multiplication de la ligne à éliminer par ce ratio.
— Soustraction la ligne pivot de la ligne.

1. Implémentez l'algorithme d'élimination Gaussienne sur une matrice carrée.
2. En utilisant OpenMP, parallélisez ce calcul. Quelle boucle allez-vous paralléliser ?

# 7 OpenMP II

## Exercice 7.1 : Somme

Implémentez un calcul qui calcule le maximum des valeurs dans un tableau :

1. Sans utiliser de réduction, en utilisant une section critique ou une opération atomique
2. En utilisant une réduction.

Comparez les performances sur des gros tableaux. Vous pourrez pour cela utiliser la fonction `clock_gettime()`.

## Exercice 7.2 : Multiplication de matrices et parallélisme imbriqué

La multiplication de matrices peut être implémentée de différentes façons en OpenMP.

1. Implémentez une multiplication matrice-matrice en parallélisant la boucle principale
2. Parallélisez la boucle intérieure. Vous pourrez utiliser une réduction. Attention au nombre de threads utilisés pour chaque niveau de parallélisation.
3. Implémentez une multiplication matrice-matrice en utilisant des tâches pour paralléliser la boucle principale.

## Exercice 7.3 : Multiplication de matrices par bloc

Une multiplication matrice-matrice peut être réalisée par blocs. Si on découpe la matrice A en 4 blocs comme suit :

$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$

Le produit $AB$ devient : $AB = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{21} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$

Plus généralement, si on note $C = AB$ et qu'on a $b$ blocs, on a $C_{ij} = \sum_{k=0}^{b-1} A_{ik}B_{kj}$

Implémentez un produit matrice-matrice par blocs en utilisant des tâches. Chaque tâche effectuera un produit de deux blocs ou une somme.

Vous pourrez utiliser les dépendances de données pour donner des indices à l'ordonnanceur de tâches.