Introduction à la programmation MPI

IF247 Algorithmique Parallèle

Philippe SWARTVAGHER
ph-sw.fr

À propos de ce (sous-)cours

Partie pratique de IF247 Algorithmique Parallèle

Organisation

- 5 séances de TP (5×2h)

Agenda

Séance 1	Introduction à MPI
Séance 2	Algorithme de tri en MPI
Séances 3 et 4	Multiplication de matrices
Séance 5	Suprise !

Objectif

- Découvrir les bases de MPI pour la programmation parallèle

Évaluation

- Petit rapport et code de la multiplication de matrices à rendre

Introduction rapide au HPC

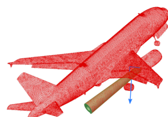
High Performance Computing

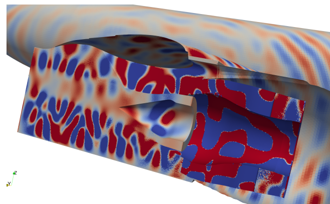
Utilisation d'ordinateurs pour résoudre des problèmes numériques complexes

Problèmes numériques

Généralement des simulations numériques :

- Prévisions météorologiques, climatiques
- Phénomènes physiques (évite de vraiment détruire des voitures !)
- Séismes
- Physique nucléaire
- Biologie
- Chimie
- ...





En haut : modélisation d'un avion ; en bas : simulation de la pression accoustique sur l'aile.

Emmanuel Agullo, Marek Felšöci, Guillaume Sylvand. Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context. IPDPS 2022 - 36th IEEE International Parallel and Distributed Processing Symposium, May 2022.

Problèmes numériques

- Demande *beaucooooooup* de calculs
- Gros problèmes : exécutions longues
  - Plusieurs heures, jours, semaines, ...
- Travail sur beaucoup de données : demande beaucoup de mémoire
  - Une matrice dense $300\,000 \times 300\,000$ en simple précision : 335 GB

Problèmes numériques

- Demande *beaucoooooooup* de calculs
- Gros problèmes : exécutions longues
  - Plusieurs heures, jours, semaines, ...
- Travail sur beaucoup de données : demande beaucoup de mémoire
  - Une matrice dense $300\,000 \times 300\,000$ en simple précision : 335 GB

Pas réalisable sur votre ordinateur portable !

Des super-ordinateurs

Super-calculateurs / clusters / machines / serveurs / nœuds / ...

- Serveurs « classiques » avec les dernières technologies
- Dédiés aux applications HPC
- En pratique : serveurs puissants connectés par un réseau

Frontier : super-ordinateur le plus puissant du monde en 2023, selon le classement Top500.
Wikipedia, OLCF at ORNL

Comparaison

Votre ordinateur portable	Un serveur HPC
1 processeur avec 4 cœurs	2 processeurs avec 64 cœurs chacun
4 GB mémoire RAM	256 GB mémoire RAM
Éventuellement 1 GPU	Jusqu'à 4 GPUs, FPGA, Xeon Phi, ...
Ethernet Latence : $\sim$ ms Débit : 1 Gbit/s	Réseau rapide Latence : $\sim$ µs Débit : 20 Gbit/s
200 Gflops	3000 Gflops

$\times$ 2 - 3000 serveurs dans un cluster !

Utilisation des clusters

Utlisation par de nombreuses personnes simultanèment

$\Rightarrow$ Ordonnanceur de travaux :

- *job scheduler*, *batch scheduler*
- Réserve un ensemble de serveurs pour une personne, pendant un temps donné, lance les programmes sur les serveurs
- Exemples : Slurm, OAR, Torque

Deux niveaux de parallèlisme

Inter-nœud

- Parallélisation sur **plusieurs nœuds**
- Mémoire distribuée
- Besoin de **communications réseau** entre les nœuds pour échanger des données
- Programmation : (par exemple) MPI (la suite de ce cours, option CISD)

Intra-nœud

- Parallélisation sur les différents cœurs d'**un nœud**
- Mémoire partagée (pas besoin de communications réseau)
- Programmation : (par exemple) OpenMP (cours *IT224 Programmation multicoeur et GPU* au S8, option CISD), mais aussi MPI

Deux expressions du parallélisme

Parallélisme explicit

- Parallélisme et communications exprimés explicitement dans le code
- Maîtrise des performances
- Difficulté d'utiliser de façon optimale toute la puissance de calcul disponible

Parallélisme implicit

- Parallélisme détecté par un support d'exécution (*runtime system*)
- Développement facilité
- Toutes les difficultés sont gérées par le support d'exécution

Difficultés de la programmation réseau HPC

API offerte par la bibliothèque C pas adaptée :

- Programmation par sockets pas vraiment pratique
- Appels système : interruption pour passer en mode privilégié dans le noyau $\rightarrow$ coûteux !

Réseaux HPC spécifiques :

- Divers constructeurs, matériels, programmation différente
- Réseau haute performance
- Programmation en espace utilisateur

Remarques plus génériques :

- Comment assurer la portabilité du code ?
- Les développeurs d'applications HPC ne sont pas forcément des spécialistes de la programmation réseau...

$\Rightarrow$ Besoin d'abstraction !

Introduction à MPI

MPI

Message Passing Interface

- https://www.mpi-forum.org/
- Standard : définit une interface
- Première version du standard en 1994, version 4.0 en 2021
- Différentes implémentations : OpenMPI, MPICH, MVAPICH, Intel MPI, MadMPI, ...
- Permet d'abstraire les différentes façons de programmer les réseaux HPC
- Langages supportés dans le standard : C et Fortran ; portages en Python et autre

Documentation

- Le standard lui-même :

  https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

- Un site sympa (mais parfois incomplet) :

  https://rookiehpc.org/mpi/docs/index.html

- Directement chez les implémentations :

  https://www.open-mpi.org/doc/current/

MPI Hello world

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    printf("Hello world\n");

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

- Tous les symboles (fonctions, constantes, ...) sont préfixés par `MPI_`
- Pas d'appel à des fonctions MPI avant `MPI_Init()` ou après `MPI_Finalize()`

Compilation

```
mpicc hello.c -o hello
```

`mpicc` est juste un raccourci vers le compilateur (`gcc`, par exemple) avec toutes les bonnes options (`-I`, `-l`, `-L`, ...) pour lier avec l'implémentation MPI
⇒ vous pouvez lui passer les options que vous passez habituellement à `gcc` (`-g`, `-O3`, `-Wall`, ...)

Exécution

```
mpirun ./hello
```

Sépcification du nombre de processus :

```
mpirun -np 4 ./hello # lance 4 processus
```

Commencer avec un simple programme non MPI, par exemple :

```
mpirun hostname
```

Sur une machine de l'Enseirb-Matmeca

```
# Rend disponible les commandes MPI :
module load mpi/openmpi-x86_64

mpicc hello.c -o hello
mpirun ./hello
```

Sur plusieurs machines de l'Enseirb-Matmeca

Étape 1 : Connexion SSH

```
ssh <machine à côté de vous>
```

Si votre mot de passe vous est demandé, il faut créer une clé SSH :

```
ssh-keygen -t ed25519 -a 100 -C "Enseirb"
# Pas de passphrase
# Laissez l'emplacement par défaut

cat $HOME/.ssh/id_ed25519.pub > \
    $HOME/.ssh/authorized_keys
```

Connectez-vous aux machines que vous souhaitez utiliser (répondez `yes` si on vous demande une confirmation)

Sur plusieurs machines de l'Enseirb-Matmeca

Étape 2 : Module toujours disponible, partout

Ajouter à la fin de ~/.bashrc :

```
source /usr/share/Modules/init/bash
module load mpi/openmpi-x86_64
```

Étape 3 : Fichier machines

Spécification des machines où lancer les processus :

```
mpirun -machinefile machines -np 4 ./hello
```

Fichier machines :

```
piano
clavecin
```

Processus MPI

Chaque processus MPI est aussi un processus au sens système :

- Espace mémoire indépendant
- Programmes (presque) indépendants

Si (au plus) un processus MPI par cœur : exécution simultanée des processus ⇒ parallèlisme

Communicateur

Groupe de processus MPI

- Nombre de processus dans un communicateur :

```
int MPI_Comm_size(MPI_Comm comm, int* size)
```

- Identifiant du processus *dans un communicateur*, appelé **rang** (allant de 0 à $p-1$) :

```
int MPI_Comm_rank(MPI_Comm comm, int* rank)
```

- Communicateur existant par défaut, contenant tous les processus : `MPI_COMM_WORLD`

MPI Hello world

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

int main(int argc, char* argv[])
{
    int rank, worldsize, hostname_len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);

    MPI_Get_processor_name(hostname, &hostname_len);

    printf(
        "[%s] Hello from rank %d ( / %d processes)\n",
        hostname, rank, worldsize
    );

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Modèle de programmation

- **Échange de messages** (message ⇔ donnée)

- Chaque processus MPI lance le même programme
- Différentes instructions suivant le rang du processus

```
// Le processus 0 envoie des données au processus 1:
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
{
    MPI_Send(...); // 0 envoie...
}
else if (rank == 1)
{
    MPI_Recv(...); // ... et 1 reçoit
}

// 0 envoie des données à tous les autres :
MPI_Bcast(buffer, ..., 0, ...);
```

Communications point-à-point

Communications point-à-point

Un processus envoie une donnée à **un** autre processus

Communications point-à-point

Un processus envoie une donnée à **un** autre processus

Envoi

```
MPI_Send(
    void* buffer,             // données à envoyer
    int count,                // nombre d'éléments
    MPI_Datatype datatype,    // type des données
    int dest_rank,            // rang du destinataire
    int tag,                  // tag du message
    MPI_Comm comm             // communicateur
);
```

Communications point-à-point

Un processus envoie une donnée à **un** autre processus

Réception

```
MPI_Recv(
    void* buffer,          // où recevoir les données
    int count,             // nombre d'éléments
    MPI_Datatype datatype, // type des données
    int src_rank,          // rang de l'émetteur
    int tag,               // tag du message
    MPI_Comm comm,         // communicateur
    MPI_Status* status     // informations
);
```

Communications point-à-point

- `MPI_Send` et `MPI_Recv` sont **bloquants** : tant que le message n'est pas envoyé (pour `MPI_Send`) ou reçu (pour `MPI_Recv`), les fonctions ne retournent pas
- Valeur de retour s'il n'y a pas eu d'erreur : `MPI_SUCCESS`
- L'émetteur et le destinataire doivent être dans le même communicateur `comm`
- `dest_rank` et `src_rank` sont respectivement les rangs du destinataire et de l'émetteur, dans le communicateur `comm`
- Le `tag` doit être le même pour l'envoi et la réception
- `comm`, `src_rank` et `tag` permettent de faire le **matching** des messages
- `count` : nombre de données contiguës à envoyer (pour pouvoir envoyer des tableaux)

Principaux types de données MPI

Valeurs possibles de datatype :

MPI_Datatype	**Type C**
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Jokers et MPI_Status

Processus source et/ou tag inconnus

- MPI_ANY_SOURCE pour accepter les messages de n'importe quel émetteur
- MPI_ANY_TAG pour accepter les messages avec n'importe quel tag

MPI_Status

```
struct MPI_Status {
    int MPI_SOURCE; // src_rank
    int MPI_TAG;    // suffisamment explicit
    int MPI_ERROR;  // éventuel code d'erreur
};
```

Si on n'a pas besoin du statut de MPI_Recv : MPI_STATUS_IGNORE

Exemple 1 : simple variable

```c
int rank, value, tag = 1;

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

if (rank == 0)
{
    value = 12;
    MPI_Send(&value, 1, MPI_INT, 1, tag, ↩
        MPI_COMM_WORLD);
}
else if (rank == 1)
{
    value = -1;
    MPI_Recv(&value, 1, MPI_INT, 0, tag, ↩
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("[%d] J'ai reçu la valeur %d\n", rank, ↩
        value);
}

MPI_Finalize();
```

Exemple 2 : tableau

```
int nb_values = 15;
int* values = malloc(nb_values * sizeof(int));

if (rank == 0)
{
    MPI_Send(values, nb_values, MPI_INT, 1, tag, ↩
        MPI_COMM_WORLD);
}
else if (rank == 1)
{
    MPI_Recv(values, nb_values, MPI_INT, 0, tag, ↩
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

free(values);
```

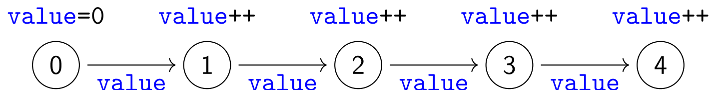
Exemple 2 : tableau

```c
int nb_values = 15;
int* values = malloc(nb_values * sizeof(int));

if (rank == 0)
{
    MPI_Send(values, nb_values, MPI_INT, 1, tag, ↩
        MPI_COMM_WORLD);
}
else if (rank == 1)
{
    MPI_Recv(values, nb_values, MPI_INT, 0, tag, ↩
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

free(values);
```

Question : pourquoi `else if (rank == 1)` et pas seulement `else` ?

Exercice : communication en anneau

Chaque processus p reçoit une valeur du processus $p-1$, l'incrémente et l'envoie au processus $p+1$.



Remarques

- Doit fonctionner quelque soit le nombre de processus > 1
- Le processus avec le rang le plus élevé affichera le résultat

À vous de jouer !

Communications collectives

Communications collectives

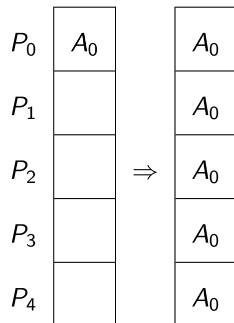
Tous les processus MPI appartenant à un communicateur sont impliqués

- Ils doivent **tous** appeler la fonction collective

Différents types de collectives

- **One-to-all** : un processus envoie à tous les autres
- **All-to-one** : tous les processus envoient à un processus
- **All-to-all** : tous les processus envoient à tous les processus

Broadcast

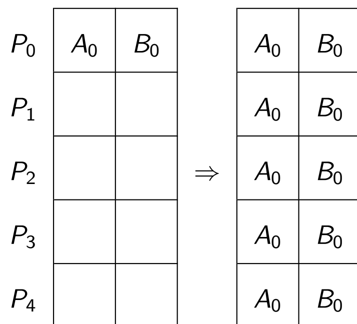


Exemple avec `count` $= 1$

```c
int MPI_Bcast(
    void* buffer,
    int count,
    MPI_Datatype datatype,
    int root_rank,
    MPI_Comm comm
);
```

- `root_rank` est le rang du processus dans `comm` qui envoie les données
- `buffer` est l'adresse qui contient les données si `rank==root_rank`, ou l'adresse où recevoir les données sinon

Broadcast



Exemple avec `count` $= 2$

```
int MPI_Bcast(
    void* buffer,
    int count,
    MPI_Datatype datatype,
    int root_rank,
    MPI_Comm comm
);
```

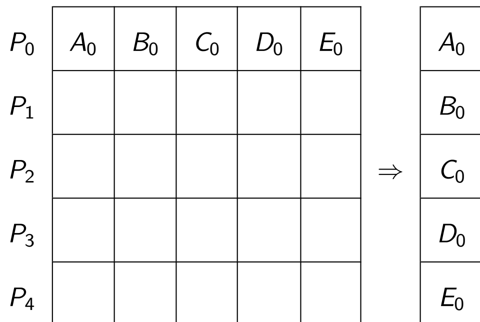
- `root_rank` est le rang du processus dans `comm` qui envoie les données
- `buffer` est l'adresse qui contient les données si `rank==root_rank`, ou l'adresse où recevoir les données sinon

Broadcast : exemple

```c
int rank, value = -1;

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

if (rank == 0)
{
    value = 12;
}

MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

printf("[%d] J'ai la valeur %d\n", rank, value);

MPI_Finalize();
```

Scatter

Répartit les données d'un processus sur plusieurs processus :

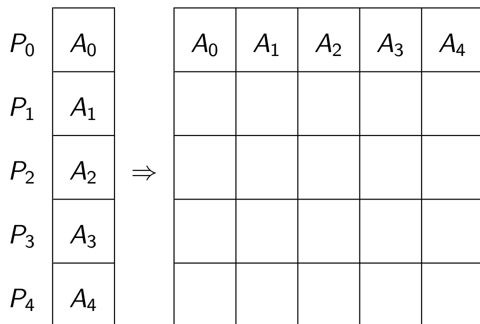


Scatter

```
int MPI_Scatter(
    const void* buffer_send,
    int count_send,
    MPI_Datatype datatype_send,
    void* buffer_recv,
    int count_recv,
    MPI_Datatype datatype_recv,
    int root,
    MPI_Comm communicator
);
```

Gather

Inverse de `MPI_Scatter` : rassemble les données de plusieurs processus sur un processus :

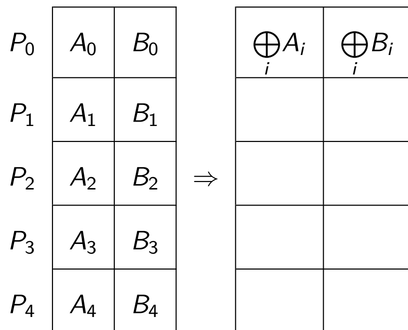


Gather

```
int MPI_Gather(
    const void* buffer_send,
    int count_send,
    MPI_Datatype datatype_send,
    void* buffer_recv,
    int count_recv,
    MPI_Datatype datatype_recv,
    int root,
    MPI_Comm communicator
);
```

Réduction

Applique l'opération $\bigoplus$ aux données sur plusieurs processus et stocke le résultat sur **un** processus :



Principales opérations possibles :

MPI_Op
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD

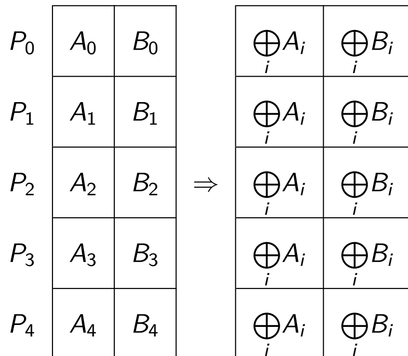
- D'autres opérations existent
- Possibilité de créer ses propres opérations

Réduction

```c
int MPI_Reduce(
    const void* send_buffer,
    void* receive_buffer,
    int count,
    MPI_Datatype datatype,
    MPI_Op operation,
    int root,
    MPI_Comm communicator
);
```

Réduction sur tous les processus

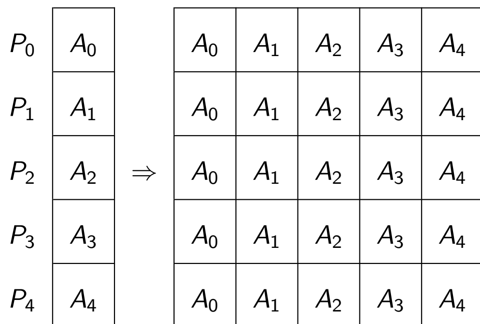
Applique l'opération $\bigoplus$ aux données sur plusieurs processus et stocke le résultat sur **tous les** processus :




```
int MPI_Allreduce(
    void* send_buffer,
    void* receive_buffer,
    int count,
    MPI_Datatype datatype,
    MPI_Op operation,
    MPI_Comm communicator
);
```

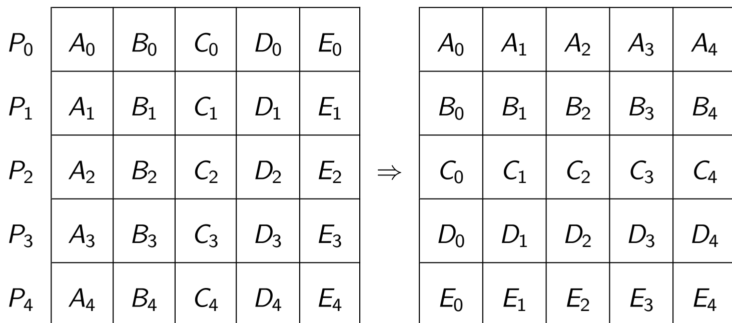
Autres collectives

`MPI_Allgather`



Autres collectives

`MPI_Alltoall`



Autres collectives

Autres collectives

Et encore d'autres collectives :

- `MPI_Exscan`
- `MPI_Reduce_scatter`
- `MPI_Reduce_scatter_block`

Certaines variantes :

- suffixées par `v` pour pouvoir préciser un nombre d'éléments distinct à envoyer à chaque processus
- suffixées par `w` pour pouvoir préciser des datatypes différents

Barrière

```
int MPI_Barrier(MPI_Comm communicator);
```

Retourne lorsque **tous** les processus de `communicator` sont entrés dans la fonction

- Synchronise les processus
- Aide pour faire des mesures de temps
- Mais **attention** : aucune garantie que tous les processus sortent *simultanèment* de la barrière !

Question : comment faire une barrière en utilisant les autres opérations collectives ?

Exercice : approximer π

Par la méthode de l'aire d'un disque (l'aire d'un disque de rayon r vaut πr^2) :

1. Choisir aléatoirement deux nombres x et y entre 0 et 1
2. Si $x^2 + y^2 \leq 1$, incrémenter un compteur c
3. Répéter n fois les étapes 1 et 2
4. $\pi \simeq 4\frac{c}{n}$

Plus n est grand et plus l'approximation sera précise

Parallélisation

?

Exercice : approximer π

Par la méthode de l'aire d'un disque (l'aire d'un disque de rayon r vaut πr^2) :

1. Choisir aléatoirement deux nombres x et y entre 0 et 1
2. Si $x^2 + y^2 \leq 1$, incrémenter un compteur c
3. Répéter n fois les étapes 1 et 2
4. $\pi \simeq 4\frac{c}{n}$

Plus n est grand et plus l'approximation sera précise

Parallélisation

1. Avec p processus, chaque processus exécute $\frac{n}{p}$ fois les étapes 1 et 2
2. Un unique processus récupère les c_i des autres processus et calcule l'estimation : $\pi \simeq 4\frac{\sum_{i=0}^{p} c_i}{n}$

Récupérer les c_i

?

Exercice : approximer π

Par la méthode de l'aire d'un disque (l'aire d'un disque de rayon r vaut πr^2) :

1. Choisir aléatoirement deux nombres x et y entre 0 et 1
2. Si $x^2 + y^2 \leq 1$, incrémenter un compteur c
3. Répéter n fois les étapes 1 et 2
4. $\pi \simeq 4\frac{c}{n}$

Plus n est grand et plus l'approximation sera précise

Parallélisation

1. Avec p processus, chaque processus exécute $\frac{n}{p}$ fois les étapes 1 et 2
2. Un unique processus récupère les c_i des autres processus et calcule l'estimation : $\pi \simeq 4\frac{\sum_{i=0}^{p} c_i}{n}$

Récupérer les c_i

- `MPI_Gather` : le processus racine doit ensuite faire la somme
- `MPI_Reduce` : le processus racine récupère directement la somme

Communicateurs

Création de communicateurs

Rappel : un communicateur (`MPI_Comm`) contient un ensemble de processus MPI pour communiquer

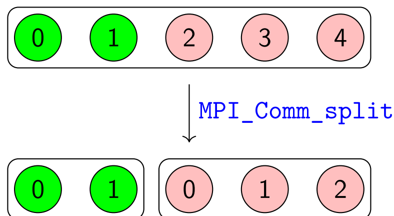
Plusieurs façons de créer des communicateurs :

- à partir d'un groupe de processus (`MPI_Group`)
- en scindant un communicateur en plusieurs sous-communicateurs

Création de sous-communicateur

On scinde un communicateur parent en plusieurs communicateurs enfants :

- Tous les processus appellent `MPI_Comm_split`
- Les processus qui utilisent la même `color` se retrouvent ensemble dans le même nouveau communicateur enfant
- `key` permet de préciser le rang du processus dans le communicateur enfant



Création de sous-communicateur

On scinde un communicateur parent en plusieurs communicateurs enfants :

- Tous les processus appellent `MPI_Comm_split`
- Les processus qui utilisent la même `color` se retrouvent ensemble dans le même nouveau communicateur enfant
- `key` permet de préciser le rang du processus dans le communicateur enfant

```
int MPI_Comm_split(
    MPI_Comm parent_comm,
    int color,
    int key,
    MPI_Comm* new_comm
);
```

Tout communicateur créé doit être libéré à la fin :

```
int MPI_Comm_free(MPI_Comm* comm);
```

Exemple

```c
int world_rank, world_size, sub_comm_rank;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int comm_color = world_rank < (world_size / 2);
MPI_Comm sub_comm;
MPI_Comm_split(
    MPI_COMM_WORLD,
    comm_color,
    world_rank,
    &sub_comm
);
MPI_Comm_rank(sub_comm, &sub_comm_rank);

printf(
    "[%d] Rank in comm %d: %d\n",
    world_rank, comm_color, sub_comm_rank
);

MPI_Comm_free(&sub_comm);
```

Pour finir

Encore beaucoup de choses à découvrir

Fonctionnalités de MPI

- Communications non-bloquantes
- Datatypes et opérations de réduction personnalisés
- Topologies des processus MPI
- Communications one-sided
- Support du multithreading
- I/O parallèle
- Communications partitionnées
- ...

Fonctionnement de MPI

- Différents protocoles réseau
- Mécanismes de progression
- Algorithmes de collectives
- ...

Encore beaucoup de choses à découvrir

Fonctionnalités de MPI

- Communications non-bloquantes
- Datatypes et opérations de réduction personnalisés
- Topologies des processus MPI
- Communications one-sided
- Support du multithreading
- I/O parallèle
- Communications partitionnées
- ...

Venez en spécialité CISD !

Fonctionnement de MPI

- Différents protocoles réseau
- Mécanismes de progression
- Algorithmes de collectives
- ...