Chap. 8 du livre de Foster: MPI

- Librairie pour échanges de messages = *Message Passing Interface*
- Disponibles dans différents langages: Fortran, C, Java
- Devenu, depuis quelques années, une standard *de facto*
- Relativement complexe: 129 fonctions

Slide 1

8.1 Le modèle de programmation MPI

- Allocation statique des processus: un nombre fixe de processus est créé au début de l'exécution du programme
- La façon dont les processus et les *threads* sont créés n'est pas décrite dans le standard
- Les processus communiquent entre eux en utilisant les fonctions de la librairie:
  - Communication point-à-point (entre deux processus)
  - Communication collective (entre plusieurs processus)

8.2 Éléments de base de MPI

Les six (6) fonctions de base de MPI:

Slide 2

- `MPI_Init`: Initialise l'utilisation de MPI
- `MPI_Finalize`: Termine l'utilisation de MPI
- `MPI_Comm_size`: Nombre de processus impliqués
- `MPI_Comm_rank`: Numéro d'identification du processus
- `MPI_Send`: Envoyer un message
- `MPI_Receive`: Recevoir un message

Description détaillée des 6 fonctions de base: Fig. 8.1 (p. 277)

Slide 3

Toutes les communications se font relative à un `communicateur`

Communicateur ≈ espace de communication dans lequel participe un certain nombre de processus

Communicateur de base = `MPI_Comm_world` = tous les processus

Exemple:

```
int main( int argc, char *argv[] );
{
  MPI_Init( &argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &count);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  printf( "Processeur %d parmi %d\n", myid, count );
  MPI_Finalize();
}
```

Slide 4

Stratégie typique pour création d'un programme avec maître/esclaves:

```
int main( int argc, char *argv[] )
{
  MPI_Init(&argc, &argv);
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
    maitre();
  else
    esclave();
  ...
  MPI_Finalize();
}
```

Autre exemple de programme: Programme 8.1 (p. 280)

Slide 5

8.2.1 Les différents langages supportant MPI

MPI défini de façon indépendante de tout langage de prog.

Différentes interfaces ont été développées pour différents langages:

- Fortran (encore trés utilisé pour la programmation scientifique)
- C
- Java (récent)

Convention en C pour les identificateurs et interfaces des opérations:

- "MPI_" ajouté devant les identificateurs
- Constantes en MAJUSCULES
- Nom d'opérations avec première lettre en Majuscule
- Résultats (OUT) retournés par l'intermédiaire de pointeurs
- Types de base pré-définis: MPI_CHAR, MPI_INT, MPI_FLOAT, etc.

Exemple: Programme 8.2 (p. 281): Programme SPMD avec organisation en anneau

Slide 6

8.2.2 Déterminisme

Ordre de réception des messages:

- Si deux processus A et B envoient un message à un autre processus C, l'ordre d'arrivée des messages n'est pas déterminé
- Si un processus A envoie un message m1, *puis* un message m2 à un autre processus C, le message m1 sera reçue et obtenue par C *avant* m2

Les communications *ne se font pas* par l'intermédiaire de canaux, mais directement d'un processus à un autre

Mais . . . un canal peut être simulé en spécifiant *l'enveloppe* d'un message:

- La source du message (numéro de processeur ou MPI_ANY_SOURCE)
- Une étiquette (*tag*) associée au message (entier ou MPI_ANY_TAG)
- Un contexte (un communicateur: voir Section 8.5)

8.3 Opérations globales de communication

En théorie: tout pourrait se faire avec `MPI_Send/Receive`

En pratique: les opérations de *communications collectives* (impliquant plusieurs processus) peuvent être mis en oeuvre de façon plus efficace par la librairie que par le programmeur

Principales opérations de communication collective = Figure 8.2 (p. 285)

Slide 7

- Barrière de synchronisation
- Diffusion d'une donnée spécifique aux autres processus
- Distribution d'un ensemble de données à l'ensemble des processus
- Collecte des données provenant des divers processus
- Application d'une opération de réduction (somme, max, etc.)

8.3.1 Barrière

Opération relative à un communicateur, c'est-à-dire, un groupe de processus = assure qu'aucun processus ne pourra poursuivre tant que tous les processus du groupe ne sont pas rendus à la barrière

```
MPI_Barrier( communicator );
```

8.3.2 Déplacement de données

Slide 8

Trois principales opérations permettent à un groupe de processus d'interagir avec un processus *racine* pour déplacer, de façon collective, des données: Voir Figure 8.3 (p. 286)

1. `MPI_Bcast(tamp, nb_elems, t_elem, racine, comm)`: envoie `nb_elems` (de type `t_elem`) provenant de `tamp` (processus `racine`) vers chacun des autres processus.

   $\Rightarrow$ Communication "un vers plusieurs"

Slide 9

2. `MPI_Gather(tamp_in, nb_elems_in, t_elem_in, tamp_out, nb_elems_out, t_elem_out, racine, comm)`: `racine` reçoit dans `tamp_out` les copies des données contenues dans `tamp_in` de chaque processus. Les items du processus `i` sont mis devant ceux de `i+1`.

   Note: `tamp_out` n'est modifié que sur `racine` et sa taille doit être P fois la taille de `tamp_in`.

   $\Rightarrow$ Communication "plusieurs vers un"

3. `MPI_Scatter(tamp_in, nb_elems_in, t_elem_in, tamp_out, nb_elems_out, t_elem_out, racine, comm)`: inverse de `MPI_Gather` = `racine` envoie la `i`ème portion des données provenant de `tamp_in` au processus `i`

   $\Rightarrow$ Communication "un vers plusieurs"

Différence entre `MPI_Bcast` et `MPI_Scatter`:

- `MPI_Bcast`: *même valeur* envoyée à chaque processus
- `MPI_Scatter`: morceaux *différents* envoyés à chaque processus

Slide 10

8.3.3 Opérations de réduction

Une opération de *réduction* permet de *combiner* (avec une opération binaire appropriée) les valeurs contenus dans les tampons des différents processus.

Deux formes de base:

- `MPI_Reduce`: laisse le résultat uniquement dans le tampon de sortie du processeur `racine`
- `MPI_Allreduce`: laisse le résultat dans le tampon de sortie *de chacun* des processeurs

Exemple: 4 processeurs avec des tampons de deux éléments

Processeur	inbuf	outbuf
0	2 4	? ?
1	5 7	? ?
2	0 3	? ?
3	6 2	? ?

```
MPI_Reduce(inbuf, outbuf, 2, MPI_INT, MPI_SUM, 3, MPI_COMM_WORLD)

 Processeur        inbuf      outbuf
 0                 2   4      ?    ?
 1                 5   7      ?    ?
 2                 0   3      ?    ?
 3                 6   2      13  16


MPI_Allreduce(inbuf, outbuf, 2, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD)
 Processeur        inbuf      outbuf
 0                 2   4      0    2
 1                 5   7      0    2
 2                 0   3      0    2
 3                 6   2      0    2
```

Autres exemples: Figure 8.4 (p. 288)

Slide 11

8.4 Communications asynchrones

Diverses opérations de communication *asynchrone* sont disponibles:

- `MPI_Iprobe`: vérifie si des messages ont été reçus et pas encore traités, mais *sans les recevoir*
- `MPI_Probe`: bloque jusqu'à ce qu'un message ayant l'enveloppe spécifiée ait été reçu. Ne lit pas/reçoit pas le message mais fait simplement retourner un statut
- `MPI_Count`: permet de déterminer la longueur d'un message identifié avec `MPI_Probe`

Slide 12

Exemple = réception de messages de longueur variable:

```
MPI_Probe( MPI_ANY_SOURCE, 0, comm, &status );  /* Bloque ... */
source = status.MPI_SOURCE;
MPI_Get_count(status, MPI_INT, &count);         /* Taille = ? */
buf = malloc(count * sizeof(int));              /* Allocation. */
MPI_Recv( buf, count, MPI_INT, source, 0, comm, &status );
```

Slide 13

8.5 Modularité

Modularité supportée à l'aide des *communicateurs*

Un communicateur permet de définir un espace limité de communication local, *privé* à un groupe de processus

Quatre (4) principales fonctions pour définir des communicateurs:

1. `MPI_Comm_dup`: Duplique un communicateur existant $\Rightarrow$ même groupe de processus, mais dans un contexte distinct
2. `MPI_Comm_split`: Crée un nouveau communicateur en sélectionnant un certain nombre de processus à l'intérieur d'un groupe existant
3. `MPI_Intercomm_create`: Crée un communicateur qui relie des processus provenant de deux groupes indépendants
4. `MPI_Comm_free`: Détruit un communicateur

Slide 14

8.5.1 Création de communicateurs

Les étiquettes de messages (*tags*) permettent de distinguer entre différentes communications à l'intérieur d'un certain contexte, mais ne permettent pas créer un contexte complètement indépendant, d'où le rôle des communicateurs

Exemple = appel d'une procédure de librairie (parallèle):

```
MPI_Comm_dup(comm, &newcomm, &ierr);

transpose(A, newcomm);   /* Procedure appelee avec
                            un contexte independant. */
MPI_Comm_free(&newcomm, &ierr);
```

Les communications faites durant l'exécution de `transpose` se font alors dans un contexte indépendant du contexte initial d'appel, assurant qu'il n'y aura pas d'interactions indésirées entre le contexte initial et le contexte de la librairie

8.5.2 Partitionnement de processus

Dans plusieurs langages, il est possible de créer de façon dynamique de nouveaux processus, de nouvelles tâches, *mais pas* en MPI!

Solution style MPI:

- Les processus existants "*se transforment*" en nouvelles tâches
- Ces processus transformés utilisent des communicateurs indépendants, pour éviter les interactions indésirées avec les autres tâches

Slide 15

`MPI_Comm_split(comm, couleur, cle, &newcomm)`:

- Doit être exécutée par tous les processus
- Crée un ou plusieurs nouveaux communicateurs (en fonction des `couleur`s ayant été spécifiées)
- Les processus qui ont indiqués la même `couleur` se retrouvent dans le même groupe
- Le numéro du processeur dans le nouveau groupe est déterminé par l'ordre des valeurs `cle`

L'opération suivante crée trois communicateurs, donc trois sous-groupes
(Voir Figure 8.10, p. 298):

```
MPI_Comm_rank( comm, &myid )
MPI_Comm_split( comm, myid % 3, myid, &newcomm );
```

Slide 16

Exemple: les deux opérations suivantes ont le même effet, parce que tous les processus spécifient la même `couleur` et `cle`:

```
MPI_Comm_split( comm, 0, 0, &newcomm );

MPI_Comm_dup( comm, &newcomm );
```

Slide 17

8.6 Autres caractéristiques de MPI

- Types dérivés: nouveaux types définis par le programmeur qui permettent de transférer, dans un seul message, des éléments arbitraires, possiblement non-contigus

  Exemple: l'opération `MPI_Send` suivante transmet les 10 nombres points-flottants `data[0]`, `data[32]`, `data[64]`, etc., et ce en un seul message:

```
float data[1024];
MPI_Datatype floattype;

MPI_Type_vector(10, 1, 32, MPI_FLOAT, &floattype);
  /* 10: nombre total d'items a transmettre
      1: nombre d'items a partir de chaque position
     32: intervalle entre les items */
MPI_Type_commit(&floattype);
MPI_Send(data, 1, floattype, dest, tag, MPI_COMM_WORLD);
```

Slide 18

- Opérations pour examiner les paramètres de l'environnement d'exécution
- Autres opérations globales de communication, y compris des opérations de réduction utilisant une opération spécifiée par l'usager
- Modes spéciaux de communication

MPI-2 = Nouvelle version en développement

- Création dynamique de processus: `MPI_Spawn`
- Communication unidirectionnelle: `MPI_Put/Get`
- Interface C++
- Communication collective non-bloquante
- *Message handlers*

En tout: 120 nouvelles fonctions ($\Rightarrow$ total de 249 fonctions!)