Programmation Parallèle avec MPI

Babacar Diop

Dpt. d'Informatique

UFR des Sciences Appliquées et de Technologies

Université Gaston Berger de Saint-Louis

2018/2019

Sommaire

- ✓Gestion de l'environnement MPI
- ✓Communications point à point
- ✓Communication collective
- ✓Types de données MPI

Message Passing Interface

- **Norme universelle** pour fournir une communication entre tous les processus dans un système de mémoire distribuée.
- La plupart des plates-formes informatiques parallèles couramment utilisées fournissent au moins une implémentation de MPI.
- Implémenté comme une collection de fonctions prédéfinies dans une bibliothèque et peuvent être appelées à partir de langages telles que C, C ++, Fortran, etc …

Message Passing Interface

- MPI est la première bibliothèque de transmission de messages normalisée, indépendante du vendeur.
- Avantages : portabilité, efficacité et flexibilité.
- MPI n'est pas une norme IEEE ou ISO, mais un "standard" pour l'écriture de programmes de transmission de messages sur des plates-formes HPC(High Performance Computing).

MPI interface

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM BG/Q MPI	BG/Q clusters	IBM, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

MPI interface

MPI Build Scripts - Linux Clusters			
Implementation	**Language**	**Script Name**	**Underlying Compiler**
MVAPCH2	**C**	**`mpicc`**	C compiler for loaded compiler package
	C++	**`mpicxx`** **`mpic++`**	C++ compiler for loaded compiler package
	Fortran	**`mpif77`**	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.
Open MPI	**C**	**`mpicc`**	C compiler for loaded compiler package
	C++	**`mpiCC`** **`mpic++`** **`mpicxx`**	C++ compiler for loaded compiler package
	Fortran	**`mpif77`**	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.

Niveau de support de threads

- Les bibliothèques MPI varient dans leur niveau de support de thread:
  - **MPI_THREAD_SINGLE - Niveau 0**: un seul thread.
  - **MPI_THREAD_FUNNELED - Niveau 1**: multi-threading avec un seul thread principal où sont dirigés tous les appels MPI.
  - **MPI_THREAD_SERIALIZED - Niveau 2**: multi-threading et sérialisation. Absence de parallélisme
  - **MPI_THREAD_MULTIPLE - Niveau 3**: Multithreading pouvant appeler MPI sans restrictions. Parallélisme absolue

Niveau de support de threads

Exemple simple permettant de déterminer la prise en charge au niveau du thread.

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] ) {
    int provided, claimed;
    /*** Select one of the following
    MPI_Init_thread( 0, 0, MPI_THREAD_SINGLE, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_FUNNELED, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_SERIALIZED, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_MULTIPLE, &provided );
    ***/
    MPI_Init_thread( 0, 0, MPI_THREAD_MULTIPLE, &provided );
    MPI_Query_thread( &claimed );
    printf( "Requete support de thread = %d Niveau de support= %d\n", claimed,
    provided );
    MPI_Finalize();
}
```

Communicateurs et groupes : MPI

- ✓MPI utilise des objets appelés communicateurs et groupes pour définir quel ensemble de processus peut communiquer entre eux
- ✓La plupart des routines MPI nécessite de spécifie un communicateur comme argument
  - ✓**MPI_COMM_WORLD** : communicateur prédéfini qui inclut tous vos processus MPI.

Rang d'un processus

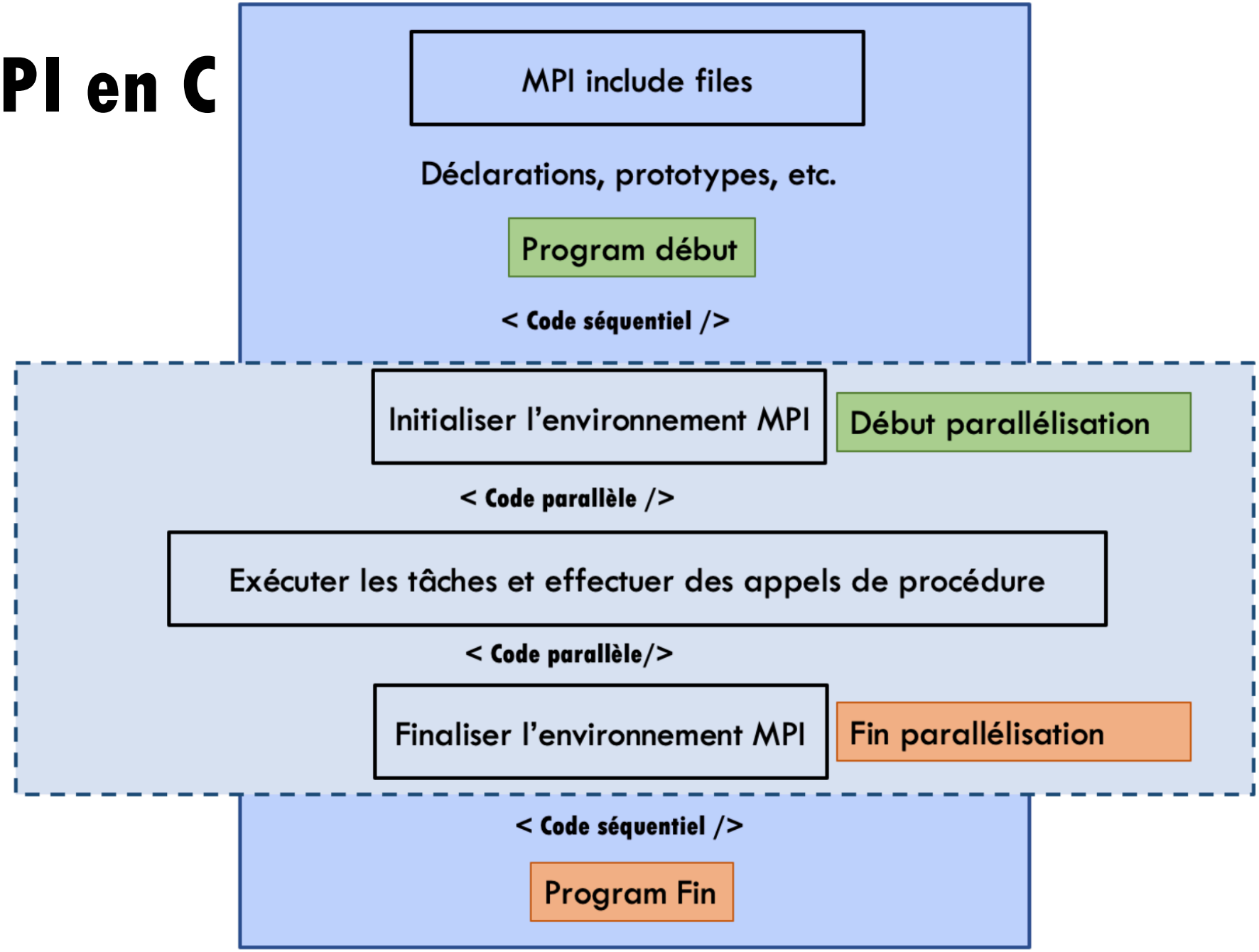
- Dans un communicateur, chaque processus a son propre identifiant, un entier unique attribué par le système lors de l'initialisation du processus.
- Un rang est parfois aussi appelé "ID du processsus". Les rangs sont contigus et commencent à zéro.
- Les rangs sont utilisés par le programmeur pour spécifier la source et la destination des messages. Souvent utilisé de manière conditionnelle par l'application pour contrôler l'exécution du programme

(if rank = 0, do this / if rank = 1, do that)

Gestion des erreurs

- La plupart des routines MPI incluent un paramètre de code retour/erreur.
- Toutefois, conformément à la norme MPI, le comportement par défaut d'un appel MPI consiste à sortir en cas d'erreur. Cela signifie que vous ne pourrez probablement pas capturer un code d'erreur autre que **MPI_SUCCESS (zero).**

Structure Code MPI en C



Initialisation: MPI_Init

- **MPI_Init**
  - Initialise l'environnement d'exécution MPI.
  - Cette fonction doit être appelée dans chaque programme MPI, avant toute autre fonction MPI et une seule fois dans un programme MPI.
  - Il peut être utilisé pour transmettre les arguments en ligne de commande à tous les processus
    - **MPI_Init (&argc, &argv)**
    - **MPI_INIT (ierr)**

Communicateur: MPI_Comm_size

- **MPI_Comm_size**
  - Retourne le nombre total de processus MPI dans le communicateur spécifié, tel que MPI_COMM_WORLD. Si le communicateur est MPI_COMM_WORLD, il représente le nombre de processus disponibles pour votre application.
    - **MPI_Comm_size (comm, & size)**
    - **MPI_COMM_SIZE (comm, size, ierr)**

Communication : MPI_Comm_rank

- **MPI_Comm_rank**
  - Renvoie le rang du processus MPI appelant dans le communicateur spécifié. Initialement, chaque processus se voit attribuer un rang entier unique compris entre 0 et le nombre de tâches - 1 au sein du communicateur MPI_COMM_WORLD. Si un processus devient associé à d'autres communicateurs, il aura également un rang unique dans chacun d'eux.
    - **MPI_Comm_rank (comm, & rank)**
    - **MPI_COMM_RANK (comm, rang, ierr)**

MPI_Abort

- **MPI_Abort**
  - Termine tous les processus MPI associés au communicateur. Dans la plupart des implémentations MPI, il met fin à TOUS les processus, quel que soit le communicateur spécifié.
    - **MPI_Abort (comm, errorcode)**
    - **MPI_ABORT (comm, errorcode, ierr)**

Nom de Processeur

- **MPI_Get_processor_name**
  - Renvoie le nom du processeur.
  - Retourne également la longueur du nom. La mémoire tampon pour "nom" doit comporter au moins MPI_MAX_PROCESSOR_NAME en caractères. Ce qui est retourné dans "name" dépend de la mise en œuvre.
    - **MPI_Get_processor_name (& name, & resultlength)**
    - **MPI_GET_PROCESSOR_NAME (name, resultlength, ierr)**

Version de MPI

- **MPI_Get_version**
  - Renvoie la version du standard MPI implémentée par la bibliothèque.
    - **MPI_Get_version (& version, & subversion)**
    - **MPI_GET_VERSION (version, sous-version, ierr)**

MPI initialisé ?

- **MPI_Initialized**
  - Indique si **MPI_Init** a été appelé - retourne un état logique vrai (1) ou faux (0).
  - MPI nécessite que **MPI_Init** soit appelé une seule fois par processus. Cela peut poser problème aux modules qui souhaitent utiliser MPI et sont prêts à appeler **MPI_Init** si nécessaire. **MPI_Initialized** résout ce problème.
    - **MPI_Initialized (&flag)**
    - **MPI_INITIALIZED (indicateur, ierr)**

Timing

- **MPI_Wtime**
  - Renvoie une durée d'horloge murale écoulée en secondes (double précision) sur le processeur appelant.
    - **MPI_Wtime ()**
    - **MPI_WTIME ()**
- **MPI_Wtick**
  - Renvoie la résolution en secondes (double précision) de MPI_Wtime.
    - **MPI_Wtick ()**
    - **MPI_WTICK ()**

Finalisation

- **MPI_Finalize**
- Termine l'environnement d'exécution MPI.
  - Cette fonction doit être la dernière routine MPI appelée dans chaque programme MPI - aucune autre routine MPI ne peut être appelée après.
    - **MPI_Finalize ()**
    - **MPI_FINALIZE (ierr)**

Exemple illustratif

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    // initialize MPI
    MPI_Init(&argc,&argv); // get number of tasks
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
     MPI_Get_processor_name(hostname, &len);
    printf ("No de taches = %d Mon Rang= %d exécutant sur %s\n",
    numtasks,rank,hostname);
    // do some work with message passing
    // done with MPI
    MPI_Finalize();
}
```

Communication point à point

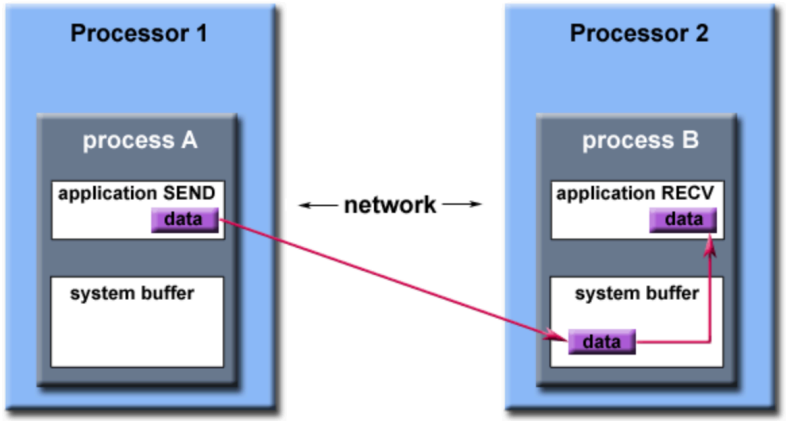
- **Types d'opérations**
  - Opérations impliquant le transfert de messages entre deux tâches MPI différentes
  - Une tâche effectue l'envoi et l'autre la réception correspondante
    - **Envoi synchrone**
    - **Blocage de l'envoi/blocage de la réception**
    - **Envoi non bloquant / réception non bloquante**
    - **Envoi tamponné**
    - **Envoi / réception combinés**
    - **"Prêt" à envoyer**
  - Toute fonction d'envoi peut être associée à n'importe quelle fonction de réception.

Buffering

- **Rôle**
  - Mise en mémoire tampon
- **Objectif**
  - Gérer le stockage des données lorsque deux tâches ne sont pas synchronisées

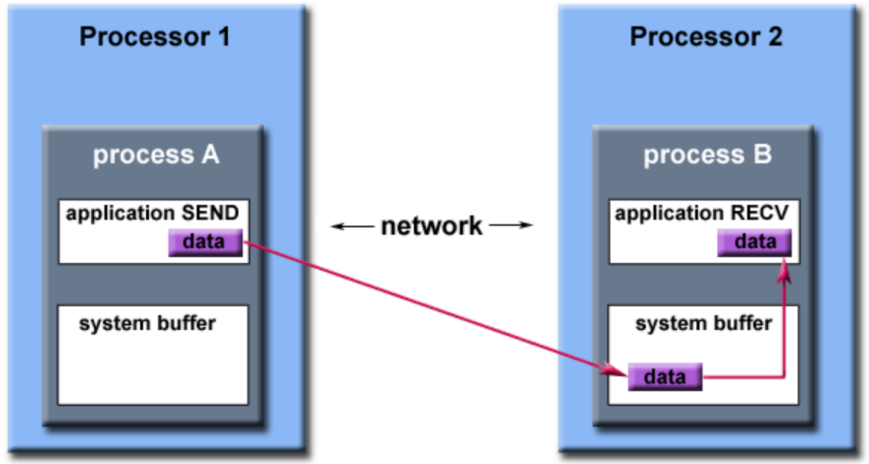
Buffering

- Considérons les deux cas suivants:
  - Une opération d'envoi se produit 5 secondes avant que la réception soit prête. Où se trouve le message pendant que la réception est en attente?
  - Plusieurs envois arrivent à la même tâche de réception qui ne peut accepter qu'un seul envoi à la fois. Qu'advient-il des messages en cours de "sauvegarde"?



Buffering

- L'implémentation MPI décide de ce qu'il advient des données dans ces types de cas.
- En règle générale, une zone de mémoire tampon appelée **buffer** est réservée aux données en transit.
- Par exemple:



Buffering

- La mémoire tampon du système est:
  - Opaque pour le programmeur et entièrement gérée par la bibliothèque MPI
  - Une ressource finie qui peut être facile à épuiser
  - Capable d'exister du côté de l'envoi, du côté de la réception ou des deux
  - Un élément susceptible d’améliorer les performances du programme car il permet aux opérations d’envoi-réception d’être asynchrones.

Communication Point-à-Point sur MPI

Envoie/Réception bloquants

- Un envoi bloquant ne « retournera » qu'après la possibilité de modifier le tampon d'application (vos données d'envoi) en vue de sa réutilisation.
- Un envoi bloquant peut être synchrone, ce qui signifie qu'un échange de données est en cours avec la tâche de réception pour confirmer un envoi sécurisé.
- Un envoi bloquant peut être asynchrone si un tampon système est utilisé pour conserver les données en vue de leur remise éventuelle à la réception.
- Une réception bloquante ne « retournera » que lorsque les données sont arrivées et sont prêtes à être utilisées par le programme.

Envoie/Réception non bloquants

- Les routines d'envoi et de réception non bloquantes n'attendent pas que des événements de communication soient terminés, tels que la copie de messages de la mémoire de l'utilisateur dans l'espace tampon du système ou l'arrivée effective du message.

- Les opérations non bloquantes "demandent" simplement à la bibliothèque MPI d'effectuer l'opération quand elle le peut. L'utilisateur ne peut pas prédire quand cela se produira.

Envoie/Réception non bloquants

- Il est dangereux de modifier le tampon d'application (votre espace variable) jusqu'à ce que vous sachiez que l'opération de demandée a bien été effectuée. Il y a des routines "d'attente" utilisées pour cela.

- Les communications non bloquantes sont principalement utilisées pour superposer les calculs aux communications et exploiter les gains de performances possibles.

Exemple

Blocking Send	Non-blocking Send

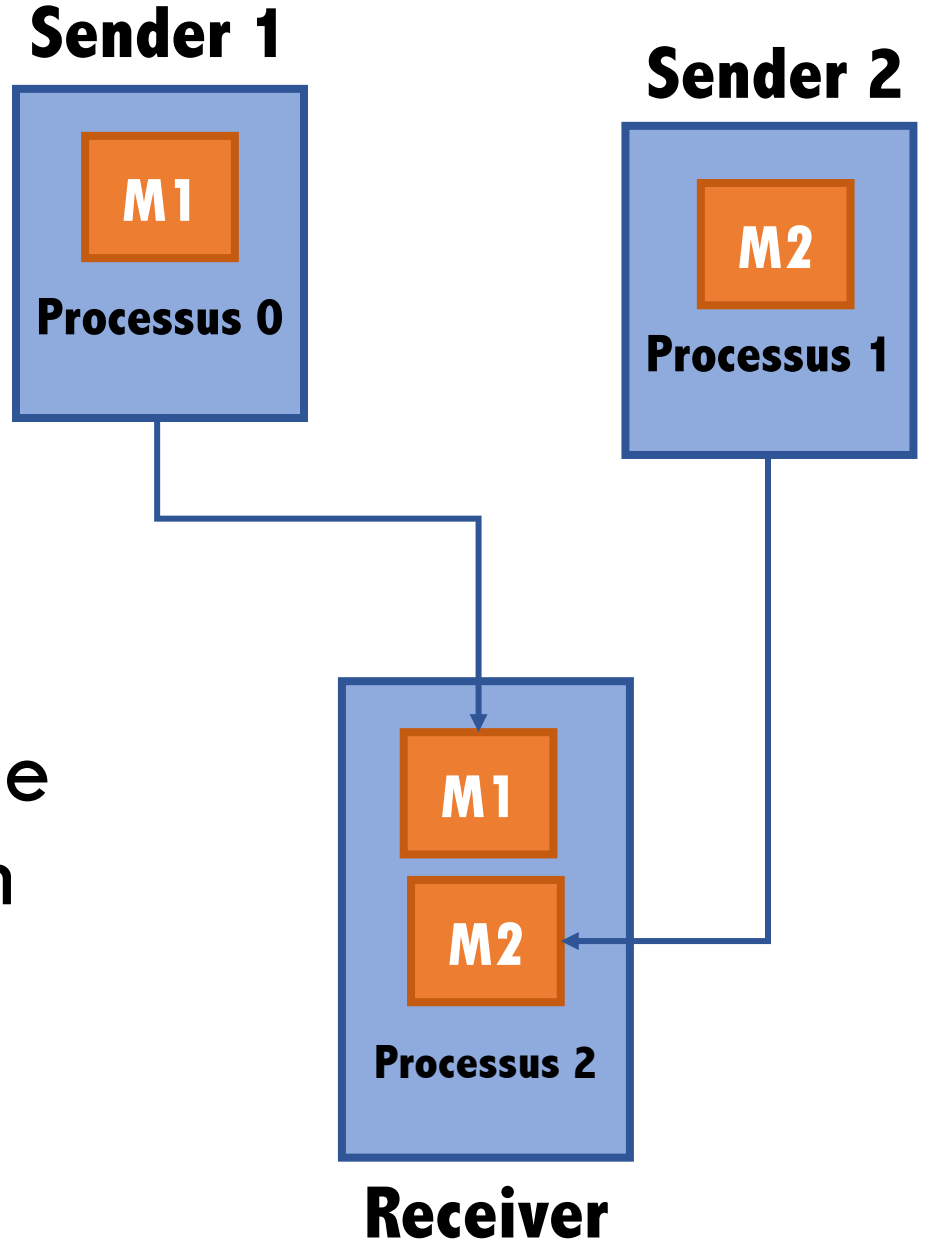
`myvar = 0;`
 	
`for (i=1; i<ntasks; i++) {`
`  task = i;`
`  MPI_Send (&myvar ... ... task ...);`
`  myvar = myvar + 2`
 	
`  /* do some work */`
 	
`  }`
	`myvar = 0;`
	
	`for (i=1; i<ntasks; i++) {`
	`  task = i;`
	`  MPI_Isend (&myvar ... ... task ...);`
	`  myvar = myvar + 2;`
	
	`  /* do some work */`
	
	`  MPI_Wait (...);`
	`  }`

Safe. Why?	Unsafe. Why?

Ordre d'envoi

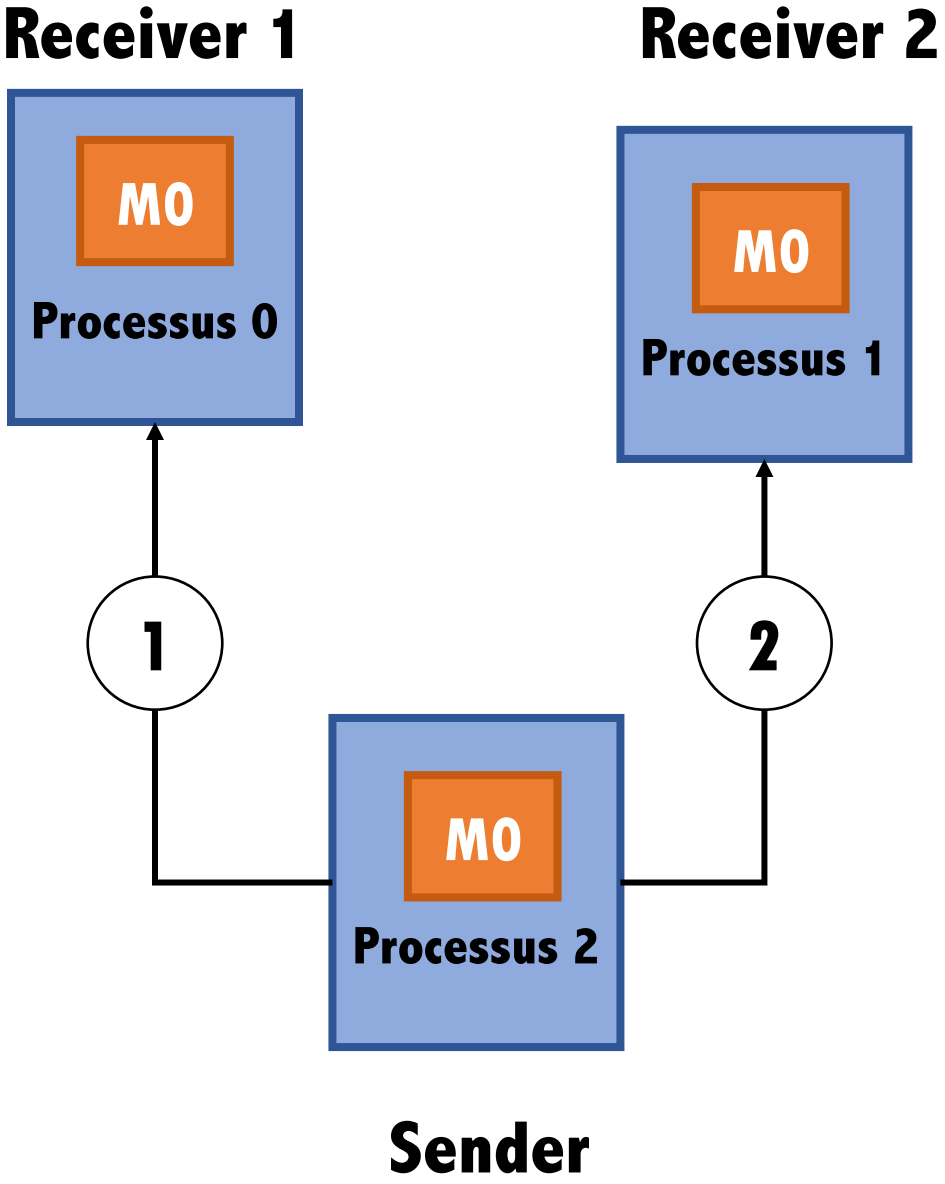
- MPI garantit l'ordre des messages.

- Si 2 messages sont envoyés (M1 et M2) successivement à la même destination, l'ordre de réception sera M1 ensuite M2



Ordre de réception

- Si 2 destinataires postent une opération de réception successivement (R1, R2) depuis le même message, alors R1 sera reçu R2

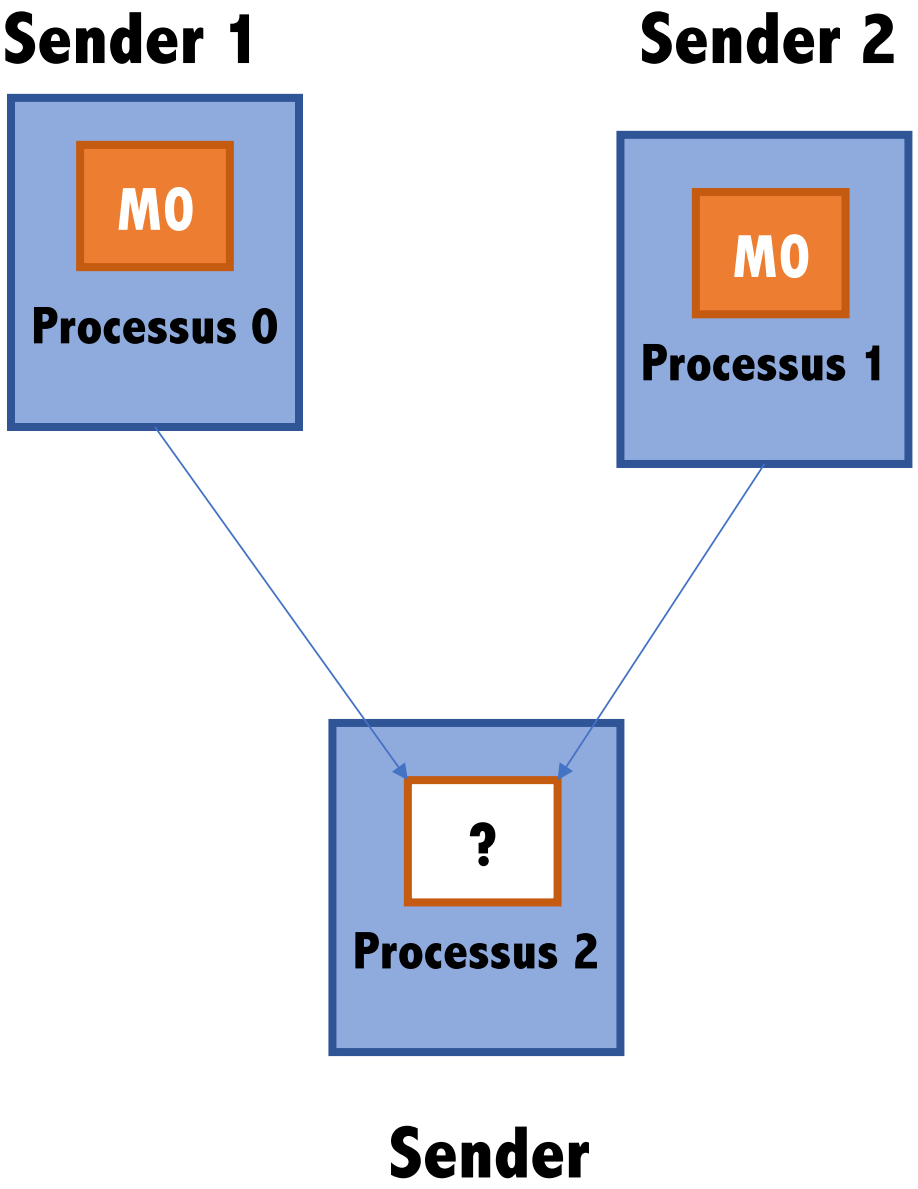


Ordre de réception

MPI ne garantit pas tous les cas

Exemple: la tâche 0 envoie un message à la tâche 2. La tâche 1 envoie un message concurrent à la tâche 2.

Un seul des envois sera effectué. Lequel ? Juste un des 2



Fonctions de passage de message

- Les fonctions de communication point à point de MPI ont généralement une liste d'arguments prenant l'un des formats suivants:

Envoi bloquant	`MPI_Send(buffer, count, type, dest, tag, comm)`
Envoi non bloquant	`MPI_Isend(buffer, count, type, dest, tag, comm, request)`
Réception bloquant	`MPI_Recv(buffer, count, type, source, tag, comm, status)`
Réception non bloquante	`MPI_Irecv(buffer, count, type, source, tag, comm, request)`

Arguments de fonctions

buffer

- Espace mémoire référençant les données à envoyer ou à recevoir.
- Dans la plupart des cas, il s'agit simplement du nom de la variable à envoyer/recevoir.
- Pour les programmes C, cet argument est passé par référence

count

- Indique le nombre d 'éléments du type à envoyer

Arguments de fonctions

source, dest

- Rang du processus émetteur/récepteur

tag

- Entier positif aléatoire attribué par le programmeur pour identifier un message de manière unique. Les opérations d'envoi et de réception doivent correspondre au tag de message.
  - MPI_ANY_TAG permet de recevoir n'importe quel message, quelle que soit le tag.

comm

- Contexte de communication
- Exemple: MPI_COMM_WORLD est généralement utilisé

Arguments de fonctions

status

- Pointeur sur une structure prédéfinie **MPI_Status**

request

- Utilisé par les opérations d'envoi et de réception non bloquantes
- Pointeur sur une structure prédéfinie **MPI_Request**

Arguments de fonctions
Types de données MPI

C Data Types		
MPI_CHAR	char	**MPI_CHARACTER**
MPI_WCHAR	wchar_t - wide character	
MPI_SHORT	signed short int	
MPI_INT	signed int	**MPI_INTEGER** **MPI_INTEGER1** **MPI_INTEGER2** **MPI_INTEGER4**
MPI_LONG	signed long int	
MPI_LONG_LONG_INT **MPI_LONG_LONG**	signed long long int	
MPI_SIGNED_CHAR	signed char	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_UNSIGNED_LONG_LONG	unsigned long long int	

Arguments de fonctions
Types de données MPI

MPI_FLOAT	float	**MPI_REAL** **MPI_REAL2** **MPI_REAL4** **MPI_REAL8**
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**
MPI_LONG_DOUBLE	long double	
MPI_C_COMPLEX **MPI_C_FLOAT_COMPLEX**	float _Complex	**MPI_COMPLEX**
MPI_C_DOUBLE_COMPLEX	double _Complex	**MPI_DOUBLE_COMPLEX**
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	
MPI_C_BOOL	_Bool	**MPI_LOGICAL**
MPI_INT8_T **MPI_INT16_T** **MPI_INT32_T** **MPI_INT64_T**	int8_t int16_t int32_t int64_t	
MPI_UINT8_T **MPI_UINT16_T** **MPI_UINT32_T** **MPI_UINT64_T**	uint8_t uint16_t uint32_t uint64_t	
MPI_BYTE	8 binary digits	**MPI_BYTE**
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	**MPI_PACKED**

Comm bloquante

- MPI_Send (&buf,count,datatype,dest,tag,comm)
- MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
- MPI_Ssend (&buf,count,datatype,dest,tag,comm)
- MPI_Sendrecv(&sendbuf,sendcount,sendtype,dest,sendtag,&recvbuf, recvcount,recvtype,source,recvtag,comm,&status)
- MPI_Probe (source,tag,comm,&status)
- MPI_Get_count (&status,datatype,&count)
- MPI_Get_count (&status,datatype,&count)

https://www.mpich.org/static/docs/v3.2/www3/index.htm

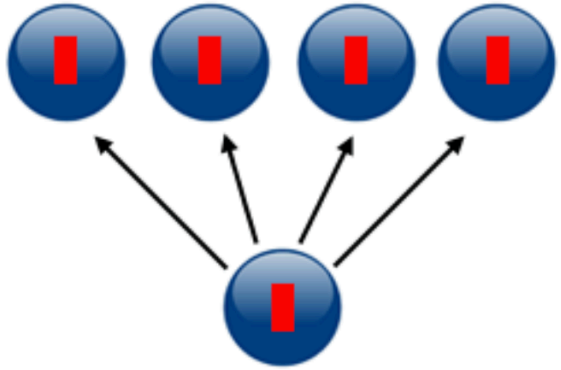
Comm non bloquante

- MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
- MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
- MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
- MPI_Iprobe (source,tag,comm,&flag,&status)
- MPI_Test (&request,&flag,&status)
  MPI_Testany (count,&array_of_requests,&index,&flag,&status)
  MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
  MPI_Testsome (incount,&array_of_requests,&outcount, &array_of_offsets, &array_of_statuses)

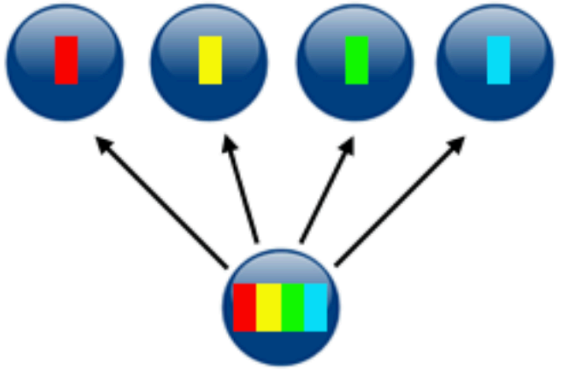
https://www.mpich.org/static/docs/v3.2/www3/index.htm

Communication collective sur MPI

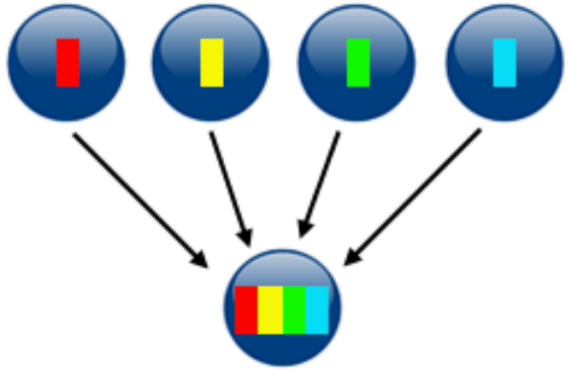
Types de communication collective



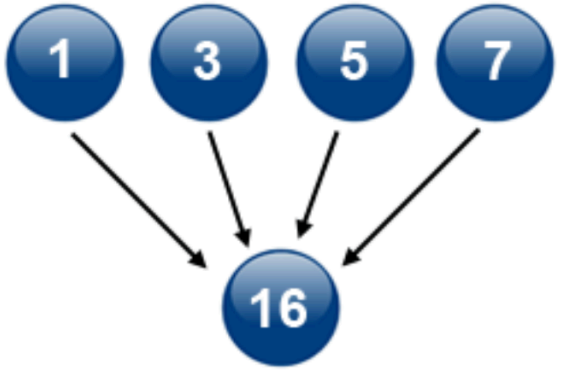
Diffusion



Diffusion ciblée



Regroupement



Aggrégation

Caractéristiques

- Les routines de communication collectives ne prennent pas de tag

- Lors d'une opération collective, les processus sont partitionnés en de nouveaux groupes, puis reliés à de nouveaux communicateurs

- Utilisable uniquement avec les types de données prédéfinis MPI - pas avec les types de données dérivées MPI.

Fonctions de comm collective

- MPI_Barrier (comm)
- MPI_Bcast (&buffer,count,datatype,root,comm)
- MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)
- MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)
- MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf, recvcount,recvtype,comm)
- MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
- MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
- MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype, op,comm)
- MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)
- MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)

https://www.mpich.org/static/docs/v3.2/www3/index.htm

Aggrégations avec MPI-Reduce

MPI Reduction Operation	
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Types natifs et définition de types dérivés sur MPI

Types natifs MPI

C Data Types	
MPI_CHAR	MPI_C_COMPLEX
MPI_WCHAR	MPI_C_FLOAT_COMPLEX
MPI_SHORT	MPI_C_DOUBLE_COMPLEX
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX
MPI_LONG	MPI_C_BOOL
MPI_LONG_LONG_INT	MPI_LOGICAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX
MPI_SIGNED_CHAR	MPI_INT8_T
MPI_UNSIGNED_CHAR	MPI_INT16_T
MPI_UNSIGNED_SHORT	MPI_INT32_T
MPI_UNSIGNED_LONG	MPI_INT64_T
MPI_UNSIGNED	MPI_UINT8_T
MPI_FLOAT	MPI_UINT16_T
MPI_DOUBLE	MPI_UINT32_T
MPI_LONG_DOUBLE	MPI_UINT64_T
	MPI_BYTE
	MPI_PACKED

Types dérivés

- MPI fournit plusieurs méthodes pour construire des types de données dérivés:
  - Contigu
  - Vecteur
  - Indexé
  - Struct

Fonctions de créations de types

- **MPI_Type_contiguous (count,oldtype,&newtype)**
  - Constructeur produisant un nouveau type de données en faisant **count** copies de **oldtype**
- **MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)**
  - Semblable à contiguë, mais avec des écarts réguliers (**stride**) dans les déplacements.
- **MPI_Type_struct(count,blocklens[],offsets[],old_types,&newtype)**
  - Le nouveau type de données est formé à partir de types de données composant

Fonctions de créations de types

- **MPI_Type_extent (datatype,&extent)**
  - Renvoie la taille en octets du type de données spécifié.
- **MPI_Type_commit (&datatype)**
  - Valide un nouveau type de données au système.
- **MPI_Type_free (&datatype)**
  - Libère l'objet spécifié, important pour éviter l'épuisement de la mémoire si de nombreux objets sont créés