Message Passing Interface (MPI)

Isabelle Dupays
Marie Flé
Jérémie Gaidamour
Dimitri Lecas

MPI – Plan I

1 Introduction . . . 6
- 1.1 Introduction . . . 7
- 1.2 Concepts de l'échange de messages . . . 8
- 1.3 Mémoire distribuée . . . 13
- 1.4 Historique . . . 18
- 1.5 Bibliographie . . . 20

2 Environnement . . . 24

3 Communications point à point . . . 30
- 3.1 Notions générales . . . 31
- 3.2 Opérations d'envoi et de réception bloquantes . . . 33
- 3.3 Types de données de base . . . 36
- 3.4 Autres possibilités . . . 38

4 Communications collectives . . . 46
- 4.1 Notions générales . . . 47
- 4.2 Synchronisation globale : `MPI_BARRIER()` . . . 49
- 4.3 Diffusion générale : `MPI_BCAST()` . . . 50
- 4.4 Diffusion sélective : `MPI_SCATTER()` . . . 53
- 4.5 Collecte : `MPI_GATHER()` . . . 56
- 4.6 Collecte générale : `MPI_ALLGATHER()` . . . 59
- 4.7 Collecte : `MPI_GATHERV()` . . . 62
- 4.8 Échanges croisés : `MPI_ALLTOALL()` . . . 66
- 4.9 Réductions réparties . . . 70

4.10 Compléments . . . 79

5 Modèles de communication . . . 80
5.1 Modes d'envoi point à point . . . 81
5.2 Appel bloquant . . . 82
5.3 Appel non bloquant . . . 95

6 Types de données dérivés . . . 105
6.1 Introduction . . . 106
6.2 Types contigus . . . 108
6.3 Types avec un pas constant . . . 109
6.4 Validation des types de données dérivés . . . 111
6.5 Exemples . . . 112
6.6 Types homogènes à pas variable . . . 118
6.7 Construction de sous-tableaux . . . 124
6.8 Types hétérogènes . . . 129
6.9 Taille des types de données . . . 133
6.10 Conclusion . . . 138

7 Communicateurs . . . 139
7.1 Introduction . . . 140
7.2 Exemple . . . 141
7.3 Communicateur par défaut . . . 142
7.4 Groupes et communicateurs . . . 143
7.5 Partitionnement d'un communicateur . . . 144

7.6 Communicateur construit à partir d'un groupe . . . 148
7.7 Topologies . . . 149

8 MPI-IO . . . 171
8.1 Introduction . . . 172
8.2 Ouverture et fermeture d'un fichier . . . 177
8.3 Lectures/écritures : généralités . . . 180
8.4 Lectures/écritures individuelles . . . 184
8.5 Lectures/écritures collectives . . . 197
8.6 Positionnement explicite des pointeurs dans un fichier . . . 208
8.7 Vues sur les fichiers . . . 211
8.8 Lectures/écritures non bloquantes . . . 225
8.9 Conseils . . . 234
8.10 Definitions . . . 235

9 Conclusion . . . 237

10 Annexes . . . 239
10.1 Communications collectives . . . 241
10.2 Types de données dérivés . . . 243
10.3 Optimisations . . . 260
10.4 Communicateurs . . . 264
10.5 Gestion de processus . . . 280
10.6 RMA . . . 305
10.7 MPI-IO . . . 332

MPI – Plan IV

11 Index 333
11.1 Index des constantes MPI 334
11.2 Index des sous-programmes MPI 337

1 Introduction
1.1 Introduction 7
1.2 Concepts de l'échange de messages 8
1.3 Mémoire distribuée 13
1.4 Historique 18
1.5 Bibliographie 20

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

1 – Introduction
1.1 – Introduction

Parallélisme

L'intérêt de faire de la programmation parallèle est :

- De réduire le temps de restitution ;
- D'effectuer de plus gros calculs ;
- D'exploiter le parallélisme des processeurs modernes (multi-coeurs, multithreading).

Mais pour travailler à plusieurs, la coordination est nécessaire. MPI est une bibliothèque permettant de coordonner des processus en utilisant le paradigme de l'échange de messages.

1 – Introduction
1.2 – Concepts de l'échange de messages

Modèle de programmation séquentiel

- le programme est exécuté par un et un seul processus ;
- toutes les variables et constantes du programme sont allouées dans la mémoire allouée au processus ;
- un processus s'exécute sur un processeur physique de la machine.

FIGURE 1 – Modèle de programmation séquentiel

Modèle de programmation par échange de messages

- le programme est écrit dans un langage classique (Fortran, C, C++, etc.) ;
- toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus ;
- chaque processus exécute éventuellement des parties différentes d'un programme ;
- une donnée est échangée entre deux ou plusieurs processus via un appel, dans le programme, à des sous-programmes particuliers.

FIGURE 2 – Modèle de programmation par échange de messages

Concepts de l'échange de messages

Si un message est envoyé à un processus, celui-ci doit ensuite le recevoir

FIGURE 3 – Échange d'un message

Constitution d'un message

- Un message est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s)
- En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :
  - l'identificateur du processus émetteur ;
  - le type de la donnée ;
  - sa longueur ;
  - l'identificateur du processus récepteur.

FIGURE 4 – Constitution d'un message

Environnement

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, à la télécopie, au courrier postal, à la messagerie électronique, etc.
- Le message est envoyé à une adresse déterminée
- Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés
- L'environnement en question est MPI (*Message Passing Interface*). Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI

1 – Introduction
1.3 – Mémoire distribuée

Architecture des supercalculateurs

La plupart des supercalculateurs sont des machines à mémoire distribuée. Ils sont composés d'un ensemble de nœud, à l'intérieur d'un nœud la mémoire est partagée.

Figure 5 – Architecture des supercalculateurs

Ada

- 332 nœuds de calcul
- 4 processeurs Intel SandyBridge (8-cœurs) à 2,7 GHz par nœud
- 10 624 cœurs
- 46 To (304 nœuds à 128Go et 28 nœuds à 256 Go)
- 230 Tflop/s crête
- 192 Tflop/s (linpack)
- 244 kWatt
- 786 MFLOPS/watt

Turing

- 4 096 nœuds
- 16 processeurs POWER A2 à 1,6 GHz par nœud
- 65 536 cœurs
- 262 144 *threads*
- 64 Tio (16 Go par nœud)
- 839 Tflop/s crête
- 716 Tflop/s (linpack)
- 329 kWatt
- 2 176 MFLOPS/watt

MPI vs OpenMP

OpenMP utilise un schéma à mémoire partagée, tandis que pour MPI la mémoire est distribuée.

FIGURE 6 – Schéma MPI

FIGURE 7 – Schéma OpenMP

Décomposition de domaine

Un schéma que l'on rencontre très souvent avec MPI est la décomposition de domaine. Chaque processus possède une partie du domaine global, et effectue principalement des échanges avec ses processus voisins.

FIGURE 8 – Découpage en sous-domaines

1 – Introduction
1.4 – Historique

Historique

- Version 1.0 : en juin 1994, le forum MPI (*Message Passing Interface Forum*), avec la participation d'une quarantaine d'organisations, abouti à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages `MPI`
- Version 1.1 : juin 1995, avec seulement des changements mineurs
- Version 1.2 : en 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes
- Version 1.3 : septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI-2.1
- Version 2.0 : apparue en juillet 1997, cette version apportait des compléments essentiels volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc.)
- Version 2.1 : juin 2008, avec seulement des clarifications dans MPI 2.0 mais aucun changement
- Version 2.2 : septembre 2009, avec seulement de « petites » additions

MPI 3.0

- Version 3.0 : septembre 2012
  - changements et ajouts importants par rapport à la version 2.2 ;
  - principaux changements :
    - communications collectives non bloquantes ;
    - révision de l'implémentation des copies mémoire à mémoire ;
    - Fortran (2003-2008) *bindings* ;
    - suppression de l'interface C++ ;
    - interfaçage d'outils externes (pour le débogage et les mesures de performance) ;
    - etc.
  - voir `http://meetings.mpi-forum.org/MPI_3.0_main_page.php`
    `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki`

1 – Introduction
1.5 – Bibliographie

Bibliographie

- Message Passing Interface Forum, *MPI : A Message-Passing Interface Standard, Version 3.0*, High Performance Computing Center Stuttgart (HLRS), 2012
  `https://fs.hlrs.de/projects/par/mpi/mpi30/`
- Message Passing Interface Forum, *MPI : A Message-Passing Interface Standard, Version 2.2*, High Performance Computing Center Stuttgart (HLRS), 2009
  `https://fs.hlrs.de/projects/par/mpi/mpi22/`
- William Gropp, Ewing Lusk et Anthony Skjellum : *Using MPI : Portable Parallel Programming with the Message Passing Interface*, second edition, MIT Press, 1999
- William Gropp, Ewing Lusk et Rajeev Thakur : *Using MPI-2*, MIT Press, 1999
- Peter S. Pacheco : *Parallel Programming with MPI*, Morgan Kaufman Ed., 1997
- Documentations complémentaires :
  `http://www.mpi-forum.org/docs/`
  `http://www.mpi-forum.org/mpi2_1/index.htm`
  `http://www.mcs.anl.gov/research/projects/mpi/learning.html`
  `http://www.mpi-forum.org/archives/notes/notes.html`

Implémentations MPI *open source*

Elles peuvent être installées sur un grand nombre d'architectures mais leurs performances sont en général en dessous de celles des implémentations constructeurs

- MPICH2 : http://www.mpich.org/
- Open MPI : http://www.open-mpi.org/

Outils

- Débogueurs
  - Totalview
    http://www.roguewave.com/products/totalview.aspx
  - DDT
    http://www.allinea.com/products/ddt/
- Outils de mesure de performances
  - MPE : *MPI Parallel Environment*
    http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm
  - Scalasca : *Scalable Performance Analysis of Large-Scale Applications*
    http://www.scalasca.org/
  - Vampir
    http://www.vampir.eu/

Bibliothèques scientifiques parallèles *open source*

- ScaLAPACK : résolution de problèmes d'algèbre linéaire par des méthodes directes. Les sources sont téléchargeables sur le site `http://www.netlib.org/scalapack/`
- PETSc : résolution de problèmes d'algèbre linéaire et non-linéaire par des méthodes itératives. Les sources sont téléchargeables sur le site `http://www.mcs.anl.gov/petsc/`
- MUMPS : résolution de grands systèmes linéaires creux par méthode directe multifrontale parallèle. Les sources sont téléchargeables sur le site `http://graal.ens-lyon.fr/MUMPS/`
- FFTW : transformées de Fourier rapides. Les sources sont téléchargeables sur le site `http://www.fftw.org`

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

Description

- Toute unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. En Fortran, il faut utiliser le *module* `mpi` introduit dans MPI-2 (dans MPI-1, il s'agissait du fichier `mpif.h`), et en C/C++ le fichier `mpi.h`.
- Le sous-programme `MPI_INIT()` permet d'initialiser l'environnement nécessaire :

```
integer, intent(out) :: code

call MPI_INIT(code)
```

- Réciproquement, le sous-programme `MPI_FINALIZE()` désactive cet environnement :

```
integer, intent(out) :: code

call MPI_FINALIZE(code)
```

Communicateurs

- Toutes les opérations effectuées par MPI portent sur des communicateurs. Le communicateur par défaut est `MPI_COMM_WORLD` qui comprend tous les processus actifs.

Figure 9 – Communicateur MPI_COMM_WORLD

Arrêt d'un programme

Parfois un programme se trouve dans une situation où il doit s'arrêter sans attendre la fin normale. C'est typiquement le cas si un des processus ne peut pas allouer la mémoire nécessaire à son calcul. Dans ce cas il faut utiliser le sous-programme `MPI_ABORT()` et non l'instruction Fortran *stop*.

```
integer, intent(in)  :: comm, erreur
integer, intent(out) :: code
call MPI_ABORT(comm, erreur, code)
```

- **comm** : tous les processus appartenant à ce communicateur seront stoppés, il est conseillé d'utiliser `MPI_COMM_WORLD` ;
- **erreur** : numéro d'erreur retourné à l'environnement UNIX.

Code

Il n'est pas nécessaire de tester la valeur de **code** après des appels aux routines MPI. Par défaut, lorsque MPI rencontre un problème, le programme s'arrête comme lors d'un appel à `MPI_ABORT()`.

Rang et nombre de processus

- À tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme MPI_COMM_SIZE() :

```
integer, intent(out) :: nb_procs,code

call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
```

- De même, le sous-programme MPI_COMM_RANK() permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par MPI_COMM_SIZE() – 1) :

```
integer, intent(out) :: rang,code

call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

```
1  program qui_je_suis
2    use mpi
3    implicit none
4    integer :: nb_procs,rang,code
5
6    call MPI_INIT(code)
7
8    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
9    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
10
11   print *,'Je suis le processus ',rang,' parmi ',nb_procs
12
13   call MPI_FINALIZE(code)
14 end program qui_je_suis
```

```
> mpiexec -n 7 qui_je_suis

Je suis le processus 3 parmi 7
Je suis le processus 0 parmi 7
Je suis le processus 4 parmi 7
Je suis le processus 1 parmi 7
Je suis le processus 5 parmi 7
Je suis le processus 2 parmi 7
Je suis le processus 6 parmi 7
```

1 Introduction

2 Environnement

3 Communications point à point
3.1 Notions générales 31
3.2 Opérations d'envoi et de réception bloquantes 33
3.3 Types de données de base 36
3.4 Autres possibilités 38

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

3 – Communications point à point
3.1 – Notions générales

Notions générales

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).

Figure 10 – Communication point à point

Notions générales

- L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur.
- Ce que l'on appelle l'enveloppe d'un message est constituée :
  - du rang du processus émetteur ;
  - du rang du processus récepteur ;
  - de l'étiquette (*tag*) du message ;
  - du communicateur qui définit le groupe de processus et le contexte de communication.
- Les données échangées sont typées (entiers, réels, etc ou types dérivés personnels).
- Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.

3 – Communications point à point
3.2 – Opérations d'envoi et de réception bloquantes

Opération d'envoi MPI_SEND

```
<type et attribut>:: message
integer :: longueur, type
integer :: rang_dest, etiquette, comm, code
call MPI_SEND(message,longueur,type,rang_dest,etiquette,comm,code)
```

Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type, étiqueté etiquette, au processus rang_dest dans le communicateur comm.

Remarque :
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée.

Opération de réception MPI_RECV

```
<type et attribut>:: message
integer :: longueur, type
integer :: rang_source, etiquette, comm, code
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_RECV(message,longueur,type,rang_source,etiquette,comm,statut,code)
```

Réception, à partir de l'adresse message, d'un message de taille longueur, de type type, étiqueté etiquette, du processus rang_source.

Remarques :

- statut reçoit des informations sur la communication : rang_source, etiquette, code, ...
- L'appel MPI_RECV ne pourra fonctionner avec une opération MPI_SEND que si ces deux appels ont la même enveloppe (rang_source, rang_dest, etiquette, comm).
- Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message corresponde au message reçu.

```
1  program point_a_point
2    use mpi
3    implicit none
4
5    integer, dimension(MPI_STATUS_SIZE) :: statut
6    integer, parameter                  :: etiquette=100
7    integer                             :: rang,valeur,code
8
9    call MPI_INIT(code)
10
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   if (rang == 2) then
14      valeur=1000
15      call MPI_SEND(valeur,1,MPI_INTEGER,5,etiquette,MPI_COMM_WORLD,code)
16   elseif (rang == 5) then
17      call MPI_RECV(valeur,1,MPI_INTEGER,2,etiquette,MPI_COMM_WORLD,statut,code)
18      print *,'Moi, processus 5, j''ai reçu ',valeur,' du processus 2.'
19   end if
20
21   call MPI_FINALIZE(code)
22
23 end program point_a_point
```

```
> mpiexec -n 7 point_a_point

Moi, processus 5, j'ai reçu 1000 du processus 2
```

3 – Communications point à point
3.3 – Types de données de base

Types de données de base Fortran

Table 1 – Principaux types de données de base (Fortran)

Type MPI	Type Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

Types de données de base C

Table 2 – Principaux types de données de base (C)

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

3 – Communications point à point
3.4 – Autres possibilités

Autres possibilités

- À la réception d'un message, le rang du processus et l'étiquette peuvent être des « *jokers* », respectivement `MPI_ANY_SOURCE` et `MPI_ANY_TAG`.
- Une communication avec le processus « fictif » de rang `MPI_PROC_NULL` n'a aucun effet.
- `MPI_STATUS_IGNORE` est une constante prédéfinie qui peut être utilisée à la place de la variable prévue pour récupérer en réception le *statut*.
- Il existe des variantes syntaxiques, `MPI_SENDRECV()` et `MPI_SENDRECV_REPLACE()`, qui effectuent un envoi et une réception (dans le premier cas, la zone de réception doit être forcément différente de la zone d'émission).
- On peut créer des structures de données plus complexes grâce à ses propres types dérivés (voir le chapitre 5).

Opération d'envoi et de réception simultanés MPI_SENDRECV

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_source, rang_dest, etiq_source, etiq_dest, comm, code
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_SENDRECV(message_emis,longueur_message_emis,type_message_emis,
                  rang_dest,etiq_source,
                  message_recu, longueur_message_recu, type_message_recu,
                  rang_source, etiq_dest, comm, statut, code)
```

- Envoi, à partir de l'adresse message_emis, d'un message de taille longueur_message_emis, de type type_message_emis, étiquetté etiq_source, au processus rang_dest dans le communicateur comm ;
- réception, à partir de l'adresse message_recu, d'un message de taille longueur_message_recu, de type type_message_recu, étiquetté etiq_dest, du processus rang_source dans le communicateur comm.

Opération d'envoi et de réception simultanés MPI_SENDRECV

FIGURE 11 – Communication **sendrecv** entre les processus 0 et 1

```
1  program sendrecv
2    use mpi
3    implicit none
4    integer                                :: rang,valeur,num_proc,code
5    integer,parameter                      :: etiquette=110
6
7    call MPI_INIT(code)
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10   ! On suppose avoir exactement 2 processus
11   num_proc=mod(rang+1,2)
12
13   call MPI_SENDRECV(rang+1000,1,MPI_INTEGER,num_proc,etiquette,valeur,1,MPI_INTEGER, &
14                     num_proc,etiquette,MPI_COMM_WORLD,MPI_STATUS_IGNORE,code)
15
16   print *,'Moi, processus',rang,', j''ai reçu',valeur,'du processus',num_proc
17
18   call MPI_FINALIZE(code)
19 end program sendrecv
```

```
> mpiexec -n 2 sendrecv

Moi, processus 1, j'ai reçu 1000 du processus 0
Moi, processus 0, j'ai reçu 1001 du processus 1
```

Attention !

Il convient de noter que dans le cas d'une implémentation synchrone du MPI_SEND() (voir le chapitre 6), le code précédent serait en situation de verrouillage si à la place de l'appel MPI_SENDRECV() on utilisait un MPI_SEND() suivi d'un MPI_RECV().
En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens.

```
call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,code)
call MPI_RECV(valeur,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,statut,code)
```

Opération d'envoi et de réception simultanés MPI_SENDRECV_REPLACE

```
<type et attribut>:: message
integer :: longueur
integer :: type
integer :: rang_source, rang_dest, etiq_source, etiq_dest, comm, code
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_SENDRECV_REPLACE(message,longueur,type,
                  rang_dest,etiq_source,
                  rang_source, etiq_dest, comm, statut, code)
```

- Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type, d'étiquette etiq_source, au processus rang_dest dans le communicateur comm ;
- réception d'un message à la même adresse, d'une taille et d'un type identique, d'étiquette etiq_dest, du processus rang_source dans le communicateur comm.

```
1  program joker
2    use mpi
3    implicit none
4    integer, parameter :: m=4,etiquette=11
5    integer, dimension(m,m) :: A
6    integer :: nb_procs,rang,code,i
7    integer, dimension( MPI_STATUS_SIZE ):: statut
8
9    call MPI_INIT (code)
10   call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)
11   call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
12   A(:,:) = 0
13
14   if (rang == 0) then
15     ! Initialisation de la matrice A sur le processus 0
16     A(:,:) = reshape((/ (i,i=1,m*m) /), (/ m,m /))
17     ! Envoi de 3 éléments de la matrice A au processus 1
18     call MPI_SEND (A(1,1),3, MPI_INTEGER ,1,etiquette, MPI_COMM_WORLD ,code)
19   else
20     ! On reçoit le message
21     call MPI_RECV (A(1,2),3, MPI_INTEGER ,MPI_ANY_SOURCE,MPI_ANY_TAG, &
22                    MPI_COMM_WORLD ,statut,code)
23     print *,'Moi processus ',rang ,'je recois 3 elements du processus ', &
24             statut( MPI_SOURCE ), "avec l'etiquette", statut( MPI_TAG ), &
25             " les elements sont ", A(1:3,2)
26   end if
27   call MPI_FINALIZE (code)
28 end program joker
```

```
> mpiexec -n 2 joker
 Moi processus            1 je recois 3 elements du processus            0
 avec l'etiquette          11  les elements sont            1           2
           3
```

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives
4.1 Notions générales 47
4.2 Synchronisation globale : MPI_BARRIER() 49
4.3 Diffusion générale : MPI_BCAST() 50
4.4 Diffusion sélective : MPI_SCATTER() 53
4.5 Collecte : MPI_GATHER() 56
4.6 Collecte générale : MPI_ALLGATHER() 59
4.7 Collecte : MPI_GATHERV() 62
4.8 Échanges croisés : MPI_ALLTOALL() 66
4.9 Réductions réparties 70
4.10 Compléments 79

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

4 – Communications collectives
4.1 – Notions générales

Notions générales

- Les communications collectives permettent de faire en une seule opération une série de communications point à point.
- Une communication collective concerne toujours tous les processus du communicateur indiqué.
- Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).
- Il est inutile d'ajouter une synchronisation globale (barrière) après une opération collective.
- La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Types de communications collectives

Il y a trois types de sous-programmes :

1. celui qui assure les synchronisations globales : `MPI_BARRIER()`.
2. ceux qui ne font que transférer des données :
   - diffusion globale de données : `MPI_BCAST()` ;
   - diffusion sélective de données : `MPI_SCATTER()` ;
   - collecte de données réparties : `MPI_GATHER()` ;
   - collecte par tous les processus de données réparties : `MPI_ALLGATHER()` ;
   - diffusion sélective, par tous les processus, de données réparties : `MPI_ALLTOALL()`.
3. ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
   - opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : `MPI_REDUCE()` ;
   - opérations de réduction avec diffusion du résultat (équivalent à un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`) : `MPI_ALLREDUCE()`.

4 – Communications collectives
4.2 – Synchronisation globale : MPI_BARRIER()

Synchronisation globale : MPI_BARRIER()

FIGURE 12 – Synchronisation globale : MPI_BARRIER()

```
integer, intent(out) :: code

call MPI_BARRIER(MPI_COMM_WORLD,code)
```

4 – Communications collectives
4.3 – Diffusion générale : MPI_BCAST()

Figure 13 – Diffusion générale : MPI_BCAST()

Diffusion générale : MPI_BCAST()

```
<type et attribut> :: message
integer :: longueur, type, rang_source, comm, code

call MPI_BCAST(message, longueur, type, rang_source, comm, code)
```

1. Envoi, à partir de l'adresse message, d'un message constitué de longueur élément de type type, par le processus rang_source, à tous les autres processus du communicateur comm.
2. Réception de ce message à l'adresse message pour les processus autre que rang_source.

```
1  program bcast
2    use mpi
3    implicit none
4
5    integer :: rang,valeur,code
6
7    call MPI_INIT(code)
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10   if (rang == 2) valeur=rang+1000
11
12   call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
13
14   print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus 2'
15
16   call MPI_FINALIZE(code)
17
18 end program bcast
```

```
> mpiexec -n 4 bcast

Moi, processus 2, j'ai reçu 1002 du processus 2
Moi, processus 0, j'ai reçu 1002 du processus 2
Moi, processus 1, j'ai reçu 1002 du processus 2
Moi, processus 3, j'ai reçu 1002 du processus 2
```

4 – Communications collectives
4.4 – Diffusion sélective : MPI_SCATTER()

Figure 14 – Diffusion sélective : MPI_SCATTER()

Diffusion sélective : MPI_SCATTER()

```
<type et attribut>:: message_a_repartir, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_source, comm, code

call MPI_SCATTER(message_a_repartir,longueur_message_emis,type_message_emis,
           message_recu,longueur_message_recu,type_message_recu,rang_source,comm,code)
```

1. Distribution, par le processus rang_source, à partir de l'adresse message_a_repartir, d'un message de taille longueur_message_emis, de type type_message_emis, à tous les processus du communicateur comm ;
2. réception du message à l'adresse message_recu, de longueur longueur_message_recu et de type type_message_recu par tous les processus du communicateur comm.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont distribuées en tranches égales, une tranche étant constituée de longueur_message_emis éléments du type type_message_emis.
- La ième tranche est envoyée au ième processus.

```
1  program scatter
2    use mpi
3    implicit none
4
5    integer, parameter                :: nb_valeurs=8
6    integer                           :: nb_procs,rang,longueur_tranche,i,code
7    real, allocatable, dimension(:) :: valeurs,donnees
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12   longueur_tranche=nb_valeurs/nb_procs
13   allocate(donnees(longueur_tranche))
14   allocate(valeurs(nb_valeurs))
15
16   if (rang == 2) then
17      valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
18      print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19               valeurs(1:nb_valeurs)
20   end if
21
22   call MPI_SCATTER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
23                    MPI_REAL,2,MPI_COMM_WORLD,code)
24   print *,'Moi, processus ',rang,', j''ai reçu ', donnees(1:longueur_tranche), &
25           ' du processus 2'
26   call MPI_FINALIZE(code)
27
28 end program scatter
```

```
> mpiexec -n 4 scatter
Moi, processus 2 envoie mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

Moi, processus 0, j'ai reçu 1001. 1002. du processus 2
Moi, processus 1, j'ai reçu 1003. 1004. du processus 2
Moi, processus 3, j'ai reçu 1007. 1008. du processus 2
Moi, processus 2, j'ai reçu 1005. 1006. du processus 2
```

4 – Communications collectives
4.5 – Collecte : MPI_GATHER()

Figure 15 – Collecte : MPI_GATHER()

Collecte : MPI_GATHER()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_dest, comm, code

call MPI_GATHER(message_emis,longueur_message_emis,type_message_emis,
                message_recu,longueur_message_recu,type_message_recu,rang_dest,comm,code)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par le processus rang_dest, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

```
1  program gather
2    use mpi
3    implicit none
4    integer, parameter                :: nb_valeurs=8
5    integer                           :: nb_procs,rang,longueur_tranche,i,code
6    real, dimension(nb_valeurs)       :: donnees
7    real, allocatable, dimension(:)   :: valeurs
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   longueur_tranche=nb_valeurs/nb_procs
14
15   allocate(valeurs(longueur_tranche))
16
17   valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
18   print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19               valeurs(1:longueur_tranche)
20
21   call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
22                   MPI_REAL,2,MPI_COMM_WORLD,code)
23
24   if (rang == 2) print *,'Moi, processus 2', 'j''ai reçu ',donnees(1:nb_valeurs)
25
26   call MPI_FINALIZE(code)
27
28 end program gather
```

```
> mpiexec -n 4 gather
Moi, processus 1 envoie mon tableau valeurs :1003. 1004.
Moi, processus 0 envoie mon tableau valeurs :1001. 1002.
Moi, processus 2 envoie mon tableau valeurs :1005. 1006.
Moi, processus 3 envoie mon tableau valeurs :1007. 1008.

Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

4 – Communications collectives

4.6 – Collecte générale : MPI_ALLGATHER()

FIGURE 16 – Collecte générale : MPI_ALLGATHER()

Collecte générale : MPI_ALLGATHER()

Correspond à un MPI_GATHER() suivi d'un MPI_BCAST() :

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: comm, code

call MPI_ALLGATHER(message_emis,longueur_message_emis,type_message_emis,
                   message_recu,longueur_message_recu,type_message_recu,comm,code)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par tous les processus, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

```
1  program allgather
2    use mpi
3    implicit none
4
5    integer, parameter                :: nb_valeurs=8
6    integer                           :: nb_procs,rang,longueur_tranche,i,code
7    real, dimension(nb_valeurs)       :: donnees
8    real, allocatable, dimension(:)   :: valeurs
9
10   call MPI_INIT(code)
11
12   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   longueur_tranche=nb_valeurs/nb_procs
16   allocate(valeurs(longueur_tranche))
17
18   valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
19
20   call MPI_ALLGATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
21                      MPI_REAL,MPI_COMM_WORLD,code)
22
23   print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)'
24
25   call MPI_FINALIZE(code)
26
27 end program allgather
```

```
> mpiexec -n 4 allgather

Moi, processus 1, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 3, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 0, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

4 – Communications collectives
4.7 – Collecte : MPI_GATHERV()

Figure 17 – Collecte : MPI_GATHERV()

Collecte "variable" : MPI_GATHERV()

Correspond à un MPI_GATHER() pour lequel la taille des messages varie :

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis
integer :: type_message_emis, type_message_recu
integer, dimension(:) :: nb_elts_recus, deplts
integer :: rang_dest, comm, code

call MPI_GATHERV(message_emis,longueur_message_emis,type_message_emis,
               message_recu,nb_elts_recus,deplts,type_message_recu,
               rang_dest,comm,code)
```

Le ième processus du communicateur comm envoie au processus rang_dest, un message depuis l'adresse message_emis, de taille longueur_message_emis, de type type_message_emis, avec réception du message à l'adresse message_recu, de type type_message_recu, de taille nb_elts_recus(i) avec un déplacement de deplts(i).

Remarques :

- Les couples (longueur_message_emis, type_message_emis) du ième processus et (nb_elts_recus(i), type_message_recu) du processus rang_dest doivent être tels que les quantités de données envoyées et reçues soient égales.

```
1  program gatherv
2    use mpi
3    implicit none
4    integer, parameter                  :: nb_valeurs=10
5    integer                             :: reste, nb_procs, rang, longueur_tranche, i, code
6    real, dimension(nb_valeurs)         :: donnees
7    real, allocatable, dimension(:)     :: valeurs
8    integer, allocatable, dimension(:) :: nb_elements_recus,deplacements
9
10   call MPI_INIT(code)
11   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   longueur_tranche=nb_valeurs/nb_procs
15   reste = mod(nb_valeurs,nb_procs)
16   if (reste > rang) longueur_tranche = longueur_tranche+1
17
18   ALLOCATE(valeurs(longueur_tranche))
19   valeurs(:) = (/(1000.+(rang*(nb_valeurs/nb_procs))+min(rang,reste)+i, &
20                  i=1,longueur_tranche)/)
21
22   PRINT *, 'Moi, processus ', rang,'envoie mon tableau valeurs : ',&
23            valeurs(1:longueur_tranche)
24
25   ALLOCATE(nb_elements_recus(nb_procs),deplacements(nb_procs))
26   IF (rang == 2) THEN
27     nb_elements_recus(1) = nb_valeurs/nb_procs
28     if (reste > 0) nb_elements_recus(1) = nb_elements_recus(1)+1
29     deplacements(1) = 0
30     DO i=2,nb_procs
31       deplacements(i) = deplacements(i-1)+nb_elements_recus(i-1)
32       nb_elements_recus(i) = nb_valeurs/nb_procs
33       if (reste > i-1) nb_elements_recus(i) = nb_elements_recus(i)+1
34     END DO
35   END IF
```

```
  CALL MPI_GATHERV (valeurs,longueur_tranche, MPI_REAL ,donnees,nb_elements_recus,&
       deplacements, MPI_REAL ,2, MPI_COMM_WORLD ,code)
  IF (rang == 2) PRINT *, 'Moi, processus 2 je recois', donnees(1:nb_valeurs)
  CALL MPI_FINALIZE (code)
end program gatherv
```

```
> mpiexec -n 4 gatherv

 Moi, processus  0 envoie mon tableau valeurs :  1001. 1002. 1003.
 Moi, processus  2 envoie mon tableau valeurs :  1007. 1008.
 Moi, processus  3 envoie mon tableau valeurs :  1009. 1010.
 Moi, processus  1 envoie mon tableau valeurs :  1004. 1005. 1006.

 Moi, processus 2 je reçois   1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
1009. 1010.
```

4 – Communications collectives
4.8 – Échanges croisés : MPI_ALLTOALL()

FIGURE 18 – Échanges croisés : MPI_ALLTOALL()

Échanges croisés : MPI_ALLTOALL()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: comm, code

call MPI_ALLTOALL(message_emis,longueur_message_emis,type_message_emis,
              message_recu,longueur_message_recu,type_message_recu,comm,code)
```

Correspond à un MPI_GATHER() étendu à tous les processus : le ième processus envoie la jème tranche au jème processus qui le place à l'emplacement de la ième tranche.

Remarque :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.

```
1  program alltoall
2    use mpi
3    implicit none
4
5    integer, parameter                :: nb_valeurs=8
6    integer                           :: nb_procs,rang,longueur_tranche,i,code
7    real, dimension(nb_valeurs)       :: valeurs,donnees
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   valeurs(:)=(/(1000.+rang*nb_valeurs+i,i=1,nb_valeurs)/)
14   longueur_tranche=nb_valeurs/nb_procs
15
16   print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ', &
17           valeurs(1:nb_valeurs)
18
19   call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
20                     MPI_REAL,MPI_COMM_WORLD,code)
21
22   print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)
23
24   call MPI_FINALIZE(code)
25 end program alltoall
```

```
> mpiexec -n 4 alltoall
Moi, processus 1 envoie mon tableau valeurs :
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.
Moi, processus 0 envoie mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 2 envoie mon tableau valeurs :
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.
Moi, processus 3 envoie mon tableau valeurs :
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

Moi, processus 0, j'ai reçu 1001. 1002. 1009. 1010. 1017. 1018. 1025. 1026.
Moi, processus 2, j'ai reçu 1005. 1006. 1013. 1014. 1021. 1022. 1029. 1030.
Moi, processus 1, j'ai reçu 1003. 1004. 1011. 1012. 1019. 1020. 1027. 1028.
Moi, processus 3, j'ai reçu 1007. 1008. 1015. 1016. 1023. 1024. 1031. 1032.
```

4 – Communications collectives
4.9 – Réductions réparties

Réductions réparties

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur `SUM(A(:))` ou la recherche de l'élément de valeur maximum dans un vecteur `MAX(V(:))`.
- MPI propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus. Le résultat est obtenu sur un seul processus (`MPI_REDUCE()`) ou bien sur tous (`MPI_ALLREDUCE()`, qui est en fait équivalent à un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`).
- Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d'entre eux.
- Le sous-programme `MPI_SCAN()` permet en plus d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du communicateur et lui-même.
- Les sous-programmes `MPI_OP_CREATE()` et `MPI_OP_FREE()` permettent de définir des opérations de réduction personnelles.

Opérations

Table 3 – Principales opérations de réduction prédéfinies (il existe aussi d'autres opérations logiques)

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
MPI_MAXLOC	Recherche de l'indice du maximum
MPI_MINLOC	Recherche de l'indice du minimum
MPI_LAND	ET logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique

FIGURE 19 – Réduction répartie : `MPI_REDUCE()` avec l'opérateur somme

Réductions réparties : MPI_REDUCE()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur, type, rang_dest
integer :: operation, comm, code

call MPI_REDUCE(message_emis,message_recu,longueur,type,operation,rang_dest,comm,code)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type, pour les processus du communicateur comm,
2. stockage à l'adresse message_recu pour le processus de rang rang_dest.

```
program reduce
  use mpi
  implicit none
  integer :: nb_procs,rang,valeur,somme,code

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  if (rang == 0) then
     valeur=1000
  else
     valeur=rang
  endif

  call MPI_REDUCE(valeur,somme,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,code)

  if (rang == 0) then
     print *,'Moi, processus 0, j''ai pour valeur de la somme globale ',somme
  end if

  call MPI_FINALIZE(code)
end program reduce
```

```
> mpiexec -n 7 reduce

Moi, processus 0, j'ai pour valeur de la somme globale 1021
```

FIGURE 20 – Réduction répartie avec diffusion du résultat : `MPI_ALLREDUCE` (utilisation de l'opérateur produit)

Réductions réparties avec diffusion du résultat : MPI_ALLREDUCE()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur, type
integer :: operation, comm, code

call MPI_ALLREDUCE(message_emis,message_recu,longueur,type,operation,comm,code)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type, pour les processus du communicateur comm,
2. stockage à l'adresse message_recu pour tous les processus du communicateur comm.

```
program allreduce

  use mpi
  implicit none

  integer :: nb_procs,rang,valeur,produit,code

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  if (rang == 0) then
     valeur=10
  else
     valeur=rang
  endif

  call MPI_ALLREDUCE(valeur,produit,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD,code)

  print *,'Moi, processus ',rang,', j''ai reçu la valeur du produit global ',produit

  call MPI_FINALIZE(code)

end program allreduce
```

```
> mpiexec -n 7 allreduce

Moi, processus 6, j'ai reçu la valeur du produit global 7200
Moi, processus 2, j'ai reçu la valeur du produit global 7200
Moi, processus 0, j'ai reçu la valeur du produit global 7200
Moi, processus 4, j'ai reçu la valeur du produit global 7200
Moi, processus 5, j'ai reçu la valeur du produit global 7200
Moi, processus 3, j'ai reçu la valeur du produit global 7200
Moi, processus 1, j'ai reçu la valeur du produit global 7200
```

4 – Communications collectives
4.10 – Compléments

Compléments

- Les sous-programmes MPI_SCATTERV(), MPI_GATHERV(), MPI_ALLGATHERV() et MPI_ALLTOALLV() étendent MPI_SCATTER(), MPI_GATHER(), MPI_ALLGATHER() et MPI_ALLTOALL() au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.
- Pour toutes les opérations de reduction, le mot-clé MPI_IN_PLACE peut être utilisé pour que les données et résultats de l'opération soient stockés au même endroit : `call MPI_REDUCE(MPI_IN_PLACE,message_recu,...)`
- Deux nouveaux sous-programmes ont été ajoutés pour étendre les possibilités des sous-programmes collectifs dans quelques cas particuliers :
  - MPI_ALLTOALLW() : version de MPI_ALLTOALLV() où les déplacements sont exprimés en octets et non en éléments,
  - MPI_EXSCAN() : version *exclusive* de MPI_SCAN(), qui elle est inclusive.

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication
5.1 Modes d'envoi point à point 81
5.2 Appel bloquant 82
5.3 Appel non bloquant 95

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

5 – Modèles de communication
5.1 – Modes d'envoi point à point

Modes d'envoi point à point

Mode	*Bloquant*	*Non bloquant*
Envoi standard	`MPI_SEND()`	`MPI_ISEND()`
Envoi synchrone	`MPI_SSEND()`	`MPI_ISSEND()`
Envoi *bufferisé*	`MPI_BSEND()`	`MPI_IBSEND()`
Envoi en mode *ready*	`MPI_RSEND()`	`MPI_IRSEND()`
Réception	`MPI_RECV()`	`MPI_IRECV()`

5 – Modèles de communication
5.2 – Appel bloquant

Définition

- Un appel est bloquant si l'espace mémoire servant à la communication peut être réutilisé immédiatement après la sortie de l'appel.
- Les données envoyées peuvent être modifiées après l'appel bloquant.
- Les données reçues peuvent être lues après l'appel bloquant.

5 – Modèles de communication
5.2 – Appel bloquant
5.2.1 – Envois synchrones

Définition

Un envoi synchrone implique une synchronisation entre les processus concernés. Un envoi ne pourra commencer que lorsque sa réception sera postée. Il ne peut y avoir de communication que si les deux processus sont prêts à communiquer.

Protocole de *rendez-vous*

Le protocole de *rendez-vous* est généralement celui employé pour les envois en mode synchrone (dépend de l'implémentation). L'accusé de réception est optionnel.

Interface de MPI_SSEND()

```
TYPE(*), intent(in)  :: valeurs
integer, intent(in)  :: taille, type, dest, etiquette, comm
integer, intent(out) :: code

call MPI_SSEND(valeurs, taille, type, dest, etiquette, comm, code)
```

Avantages

- Consomment peu de ressources (pas de *buffer*)
- Rapides si le récepteur est prêt (pas de recopie dans un *buffer*)
- Garantie de la réception grâce à la synchronisation

Inconvénients

- Temps d'attente si le récepteur n'est pas là/pas prêt
- Risques d' inter-blocage

Exemple d'inter-blocage

Dans l'exemple suivant, on a un inter-blocage, car on est en mode synchrone, les deux processus sont bloqués sur le MPI_SSEND() car ils attendent le MPI_RECV() de l'autre processus. Or ce MPI_RECV() ne pourra se faire qu'après le déblocage du MPI_SSEND().

```
1  program ssendrecv
2    use mpi
3    implicit none
4    integer                                :: rang,valeur,num_proc,code
5    integer,parameter                      :: etiquette=110
6
7    call MPI_INIT(code)
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10   ! On suppose avoir exactement 2 processus
11   num_proc=mod(rang+1,2)
12
13   call MPI_SSEND(rang+1000,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,code)
14   call MPI_RECV(valeur,1,MPI_INTEGER, num_proc,etiquette,MPI_COMM_WORLD, &
15                 MPI_STATUS_IGNORE,code)
16
17   print *,'Moi, processus',rang,', j''ai reçu',valeur,'du processus',num_proc
18
19   call MPI_FINALIZE(code)
20 end program ssendrecv
```

5 – Modèles de communication
5.2 – Appel bloquant
5.2.2 – Envois *bufferisés*

Définition

Un envoi bufferisé implique la recopie des données dans un espace mémoire intermédiaire. Il n'y a alors pas de couplage entre les deux processus de la communication. La sortie de ce type d'envoi ne signifie donc pas que la réception a eu lieu.

Protocole avec *buffer* utilisateur du côté de l'émetteur

Dans cette approche, le *buffer* se trouve du côté de l'émetteur et est géré explicitement par l'application. Un *buffer* géré par MPI peut exister du côté du récepteur. De nombreuses variantes sont possibles. L'accusé de réception est optionnel.

Envois *bufferisés*

Les *buffers* doivent être gérés manuellement (avec appels à MPI_BUFFER_ATTACH() et MPI_BUFFER_DETACH()). Ils doivent être alloués en tenant compte des surcoûts mémoire des messages (en ajoutant la constante MPI_BSEND_OVERHEAD pour chaque instance de message).

Interfaces

```
TYPE(*), intent(in)  :: valeurs
integer, intent(in)  :: taille, type, dest, etiquette, comm
integer, intent(out) :: code
TYPE(*) :: buf
integer :: taille_buf

call MPI_BSEND(valeurs, taille, type, dest, etiquette, comm, code)
call MPI_BUFFER_ATTACH(buf, taille_buf, code)
call MPI_BUFFER_DETACH(buf, taille_buf, code)
```

Protocole *eager*

Le protocole *eager* est souvent employé pour les envois en mode standard (`MPI_SEND()`) pour les messages de petites tailles. Il peut aussi être utilisé pour les envois avec `MPI_BSEND()` avec des petits messages (dépend de l'implémentation) et en court-circuitant le *buffer* utilisateur du côté de l'émetteur. Dans cette approche, le *buffer* se trouve du côté du récepteur. L'accusé de réception est optionnel.

Avantages

- Pas besoin d'attendre le récepteur (recopie dans un *buffer*)
- Pas de risque de blocage (*deadlocks*)

Inconvénients

- Consomment plus de ressources (occupation mémoire par les *buffers* avec risques de saturation)
- Les *buffers* d'envoi doivent être gérés manuellement (souvent délicat de choisir une taille adaptée)
- Un peu plus lent que les envois synchrones si le récepteur est prêt
- Pas de garantie de la bonne réception (découplage envoi-réception)
- Risque de gaspillage d'espace mémoire si les *buffers* sont trop sur-dimensionnés
- L'application plante si les *buffers* sont trop petits
- Il y a aussi souvent des *buffers* cachés géré par l'implémentation MPI du côté de l'expéditeur et/ou du récepteur (et consommant des ressources mémoires)

Absence d'inter-blocage

Dans l'exemple suivant, on a pas d'inter-blocage, car on est en mode bufferisé. Une fois la copie faite dans le *buffer*, l'appel MPI_BSEND() retourne et on passe à l'appel MPI_RECV().

```
1  program bsendrecv
2    use mpi
3    implicit none
4    integer                                :: rang,valeur,num_proc,taille,surcout,code
5    integer,parameter                      :: etiquette=110, nb_elt=1
6    integer,dimension(:), allocatable      :: buffer
7
8    call MPI_INIT(code)
9    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
10
11   call MPI_TYPE_SIZE(MPI_INTEGER,taille,code)
12   ! Convertir taille MPI_BSEND_OVERHEAD (octets) en nombre d'integer
13   surcout = int(1+(MPI_BSEND_OVERHEAD*1.)/taille)
14   allocate(buffer(nb_elt+surcout))
15   call MPI_BUFFER_ATTACH( buffer,taille*(nb_elt+surcout),code)
16   ! On suppose avoir exactement 2 processus
17   num_proc=mod(rang+1,2)
18   call MPI_BSEND(rang+1000,nb_elt,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,code)
19   call MPI_RECV(valeur,nb_elt,MPI_INTEGER, num_proc,etiquette,MPI_COMM_WORLD, &
20                 MPI_STATUS_IGNORE,code)
21
22   print *,'Moi, processus',rang,', j''ai reçu',valeur,'du processus',num_proc
23   call MPI_BUFFER_DETACH(buffer,taille*(1+surcout),code)
24   call MPI_FINALIZE(code)
25 end program bsendrecv
```

5 – Modèles de communication
5.2 – Appel bloquant
5.2.3 – Envois standards

Envois standards

Un envoi standard se fait en appelant le sous-programme MPI_SEND(). Dans la plupart des implémentations, ce mode passe d'un mode *bufferisé* à un mode synchrone lorsque la taille des messages croît.

Interfaces

```
TYPE(*), intent(in)  :: valeurs
integer, intent(in)  :: taille, type, dest, etiquette, comm
integer, intent(out) :: code

call MPI_SEND(valeurs, taille, type, dest, etiquette, comm, code)
```

Ping−pong extranode balanced
Vargas − Standard
Vargas − Synchroneous
Vargas − Buffered
Babel − Standard
Babel − Synchroneous
Babel − Buffered
Bandwidth (MiB/s)
Message size (bytes)
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{2}
10^{4}
10^{6}
10^{8}

Avantages

- Souvent le plus performant (choix du mode le plus adapté par le constructeur)
- Le plus portable pour les performances

Inconvénients

- Peu de contrôle sur le mode réellement utilisé (souvent accessible via des variables d'environnement)
- Risque de *deadlock* selon le mode réel
- Comportement pouvant varier selon l'architecture et la taille du problème

5 – Modèles de communication
5.2 – Appel bloquant
5.2.5 – Envois en mode *ready*

Envois en mode *ready*

Un envoi en mode *ready* se fait en appelant le sous-programme `MPI_RSEND()`. Attention : il est obligatoire de faire ces appels seulement lorsque la réception est déjà postée. **Leur utilisation est fortement déconseillée**.

Avantages

- Légèrement plus performant que le mode synchrone car le protocole de synchronisation peut être simplifié

Inconvénients

- Erreurs si le récepteur n'est pas prêt lors de l'envoi

5 – Modèles de communication
5.3 – Appel non bloquant

Présentation

Le recouvrement des communications par des calculs est une méthode permettant de réaliser des opérations de communications en arrière-plan pendant que le programme continue de s'exécuter. Sur Ada, la latence d'une communication inter-nœud est de 1.5μs soit 4000 cycles processeur.

- Il est ainsi possible, si l'architecture matérielle et logicielle le permet, de masquer tout ou une partie des coûts de communications.
- Le recouvrement calculs-communications peut être vu comme un niveau supplémentaire de parallélisme.
- Cette approche s'utilise dans MPI par l'utilisation de sous-programmes non-bloquants (i.e. `MPI_ISEND()`, `MPI_IRECV()` et `MPI_WAIT()`).

Définition

Un appel non bloquant rend la main très rapidement, mais n'autorise pas la réutilisation immédiate de l'espace mémoire utilisé dans la communication. Il est nécessaire de s'assurer que la communication est bien terminée (avec `MPI_WAIT()` par exemple) avant de l'utiliser à nouveau.

Recouvrement partiel

Recouvrement total

Avantages

- Possibilité de masquer tout ou une partie des coûts des communications (si l'architecture le permet)
- Pas de risques de *deadlock*

Inconvénients

- Surcoûts plus importants (plusieurs appels pour un seul envoi ou réception, gestion des requêtes)
- Complexité plus élevée et maintenance plus compliquée
- Peu performant sur certaines machines (par exemple avec transfert commençant seulement à l'appel de `MPI_WAIT()`)
- Risque de perte de performance sur les noyaux de calcul (par exemple gestion différenciée entre la zone proche de la frontière d'un domaine et la zone intérieure entraînant une moins bonne utilisation des caches mémoires)
- Limité aux communications point à point (a été étendu aux collectives dans MPI 3.0)

Utilisation

L'envoi d'un message se fait en 2 étapes :

- Initier l'envoi ou la réception par un appel à un sous-programme commençant par `MPI_ISEND()` ou `MPI_IRECV()` (ou une de leurs variantes)
- Attendre la fin de la contribution locale par un appel à `MPI_WAIT()` (ou à une de ses variantes).

Les communications sont recouvertes par toutes les opérations qui se déroulent entre ces deux étapes. L'accès aux données en cours de réception est interdit avant la fin de l'appel à `MPI_WAIT()` (l'accès en lecture aux données en cours d'envoi est également interdit pour les implémentations MPI antérieures à la 2.2).

Interfaces

MPI_ISEND() MPI_ISSEND() et MPI_IBSEND() pour les envois non bloquants

```
TYPE(*), intent(in)  :: valeurs
integer, intent(in)  :: taille, type, dest, etiquette, comm
integer, intent(out) :: req, code

call MPI_ISEND(valeurs, taille, type, dest, etiquette, comm, req, code)
call MPI_ISSEND(valeurs, taille, type, dest, etiquette, comm, req, code)
call MPI_IBSEND(valeurs, taille, type, dest, etiquette, comm, req, code)
```

MPI_IRECV() pour les réceptions non bloquantes.

```
TYPE(*), intent(in)  :: valeurs
integer, intent(in)  :: taille, type, source, etiquette, comm
integer, intent(out) :: req, code

call MPI_IRECV(valeurs, taille, type, source, etiquette, comm, req, code)
```

Interfaces

MPI_WAIT() attend la fin d'une communication.MPI_TEST est la version non bloquante.

```
integer, intent(inout) :: req
integer, dimension(MPI_STATUS_SIZE), intent(out) :: statut
integer, intent(out) :: code
logical, intent(out) :: flag

call MPI_WAIT(req, statut, code)
call MPI_TEST(req, flag, statut, code)
```

MPI_WAITALL() attend la fin de toutes les communications. MPI_TESTALL est la version non bloquante.

```
integer, intent(in) :: taille
integer, dimension(taille) :: reqs
integer, dimension(MPI_STATUS_SIZE,taille), intent(out) :: statuts
integer, intent(out) :: code
logical, intent(out) :: flag

call MPI_WAITALL(taille, reqs, statuts, code)
call MPI_TESTALL(taille, reqs, statuts, flag, code)
```

N
W
E
S

```
1   SUBROUTINE debut_communication(u)
2     !Envoi au voisin N et reception du voisin S
3     CALL MPI_IRECV( u(,), 1, type_ligne, voisin(S), &
4          etiquette, comm2d, requete(1), code)
5     CALL MPI_ISEND( u(,), 1, type_ligne, voisin(N), &
6          etiquette, comm2d, requete(2), code)
7
8     !Envoi au voisin S et reception du voisin N
9     CALL MPI_IRECV( u(,), 1,type_ligne,voisin(N), &
10         etiquette, comm2d, requete(3), code)
11    CALL MPI_ISEND(  u(,), 1,type_ligne,voisin(S), &
12         etiquette,comm2d, requete(4),code)
13
14    !Envoi au voisin W et  reception du voisin E
15    CALL MPI_IRECV( u(,), 1, type_colonne , voisin(E), &
16         etiquette, comm2d, requete(5), code)
17    CALL MPI_ISEND( u(,), 1, type_colonne , voisin(W), &
18         etiquette,comm2d, requete(6),code)
19
20    !Envoi au voisin E et  reception du voisin W
21    CALL MPI_IRECV( u(,), 1, type_colonne, voisin(W), &
22         etiquette, comm2d, requete(7),code)
23    CALL MPI_ISEND( u(,), 1, type_colonne, voisin(E), &
24         etiquette, comm2d, requete(8),code)
25  END SUBROUTINE debut_communication
26  SUBROUTINE fin_communication(u)
27    CALL MPI_WAITALL(2*NB_VOISINS,requete,tab_statut,code)
28  END SUBROUTINE fin_communication
```

```
DO WHILE ((.NOT. convergence) .AND. (it < it_max))
  it = it +1
  u(sx:ex,sy:ey) = u_nouveau(sx:ex,sy:ey)

  !Echange des points aux interfaces pour u a l'iteration n
  CALL debut_communication( u )

  !Calcul de u a l'iteration n+1
  CALL calcul( u, u_nouveau, sx+1, ex-1, sy+1, ey-1)

  CALL fin_communication( u )

  ! Nord
  CALL calcul( u, u_nouveau, sx, sx, sy, ey)
  ! Sud
  CALL calcul( u, u_nouveau, ex, ex, sy, ey)
  ! Ouest
  CALL calcul( u, u_nouveau, sx, ex, sy, sy)
  ! Est
  CALL calcul( u, u_nouveau, sx, ex, ey, ey)

  !Calcul de l'erreur globale
  diffnorm =  erreur_globale (u, u_nouveau)
  !Arret du programme si on a atteint la precision machine obtenu
  !par la fonction F90 EPSILON
  convergence = ( diffnorm < eps )

END DO
```

Niveau de recouvrement sur différentes machines

Machine	*Niveau*
Blue Gene/Q, PAMID_THREAD_MULTIPLE=0	32%
Blue Gene/Q, PAMID_THREAD_MULTIPLE=1	100%
Ada, défaut	0%
Ada MP_USE_BULK_XFER=yes	100%

Mesures faites en recouvrant un noyau de calcul et un noyau de communication de mêmes durées et en utilisant différents schémas de communications (intra/extra-nœuds, par paires, processus aléatoires...).
Selon le schéma de communication, les résultats peuvent être totalement différents.

Un recouvrement de 0% signifie que la durée totale d'exécution vaut 2x la durée d'un noyau de calcul (ou communication).
Un recouvrement de 100% signifie que la durée totale vaut 1x la durée d'un noyau de calcul (ou communication).

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

6.1 Introduction 106
6.2 Types contigus 108
6.3 Types avec un pas constant 109
6.4 Validation des types de données dérivés 111
6.5 Exemples 112
6.6 Types homogènes à pas variable 118
6.7 Construction de sous-tableaux 124
6.8 Types hétérogènes 129
6.9 Taille des types de données 133
6.10 Conclusion 138

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

6 – Types de données dérivés
6.1 – Introduction

Introduction

- Dans les communications, les données échangées sont typées : `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc.
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_INDEXED()` ou `MPI_TYPE_CREATE_STRUCT()`.
- Les types dérivés permettent notamment l'échange de données non contiguës ou non homogènes en mémoire et de limiter le nombre d'appels aux sous-programmes de communications.

FIGURE 21 – Hiérarchie des constructeurs de type MPI

6 – Types de données dérivés
6.2 – Types contigus

Types contigus

- MPI_TYPE_CONTIGUOUS() crée une structure de données à partir d'un ensemble homogène de type préexistant de données contiguës en mémoire.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
call MPI_TYPE_CONTIGUOUS(5,MPI_REAL,nouveau_type,code)
```

FIGURE 22 – Sous-programme MPI_TYPE_CONTIGUOUS

```
integer, intent(in)  :: nombre, ancien_type
integer, intent(out) :: nouveau_type,code

call MPI_TYPE_CONTIGUOUS(nombre,ancien_type,nouveau_type,code)
```

6 – Types de données dérivés
6.3 – Types avec un pas constant

Types avec un pas constant

- `MPI_TYPE_VECTOR()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'**éléments**.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
call MPI_TYPE_VECTOR(6,1,5,MPI_REAL,nouveau_type,code)
```

FIGURE 23 – Sous-programme MPI_TYPE_VECTOR

```
integer, intent(in)  :: nombre_bloc,longueur_bloc
integer, intent(in)  :: pas ! donné en éléments
integer, intent(in)  :: ancien_type
integer, intent(out) :: nouveau_type,code

call MPI_TYPE_VECTOR(nombre_bloc,longueur_bloc,pas,ancien_type,nouveau_type,code)
```

Types avec un pas constant

- MPI_TYPE_CREATE_HVECTOR() crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'**octets**.
- Cette instruction est utile lorsque le type générique n'est plus un type de base (MPI_INTEGER, MPI_REAL,...) mais un type plus complexe construit à l'aide des sous-programmes MPI, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

```
integer, intent(in)                          :: nombre_bloc,longueur_bloc
integer(kind=MPI_ADDRESS_KIND), intent(in) :: pas ! donné en octets
integer, intent(in)                          :: ancien_type
integer, intent(out)                         :: nouveau_type, code

call MPI_TYPE_CREATE_HVECTOR(nombre_bloc,longueur_bloc,pas,
                             ancien_type,nouveau_type,code)
```

6 – Types de données dérivés
6.4 – Validation des types de données dérivés

Validation des types de données dérivés

- Les types dérivés doivent être validés avant d'être utilisé dans une communication. La validation s'effectue à l'aide du sous-programme MPI_TYPE_COMMIT().

```
integer, intent(inout) :: nouveau_type
integer, intent(out)   :: code

call MPI_TYPE_COMMIT(nouveau_type,code)
```

- Si on souhaite réutiliser le même nom pour définir un autre type dérivé, on doit au préalable le libérer en utilisant le sous-programme MPI_TYPE_FREE().

```
integer, intent(inout) :: nouveau_type
integer, intent(out)   :: code

call MPI_TYPE_FREE(nouveau_type,code)
```

6 – Types de données dérivés

6.5 – Exemples

6.5.1 – Type « colonne d'une matrice »

```
1  program colonne
2    use mpi
3    implicit none
4
5    integer, parameter                      :: nb_lignes=5,nb_colonnes=6
6    integer, parameter                      :: etiquette=100
7    real, dimension(nb_lignes,nb_colonnes) :: a
8    integer, dimension(MPI_STATUS_SIZE)     :: statut
9    integer                                 :: rang,code,type_colonne
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   ! Initialisation de la matrice sur chaque processus
15   a(:,:) = real(rang)
16
17   ! Définition du type type_colonne
18   call MPI_TYPE_CONTIGUOUS(nb_lignes,MPI_REAL,type_colonne,code)
19
20   ! Validation du type type_colonne
21   call MPI_TYPE_COMMIT(type_colonne,code)
```

```
22    ! Envoi de la première colonne
23    if ( rang == 0 ) then
24      call MPI_SEND(a(1,1),1,type_colonne,1,etiquette,MPI_COMM_WORLD,code)
25
26    ! Réception dans la dernière colonne
27    elseif ( rang == 1 ) then
28      call MPI_RECV(a(1,nb_colonnes),nb_lignes,MPI_REAL,0,etiquette,&
29                    MPI_COMM_WORLD,statut,code)
30    end if
31
32    ! Libère le type
33    call MPI_TYPE_FREE(type_colonne,code)
34
35    call MPI_FINALIZE(code)
36
37  end program colonne
```

6 – Types de données dérivés

6.5 – Exemples

6.5.2 – Type « ligne d'une matrice »

```
1  program ligne
2    use mpi
3    implicit none
4
5    integer, parameter                      :: nb_lignes=5,nb_colonnes=6
6    integer, parameter                      :: etiquette=100
7    real, dimension(nb_lignes,nb_colonnes):: a
8    integer, dimension(MPI_STATUS_SIZE)    :: statut
9    integer                                 :: rang,code,type_ligne
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   ! Initialisation de la matrice sur chaque processus
15   a(:,:) = real(rang)
16
17   ! Définition du type type_ligne
18   call MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)
19
20   ! Validation du type type_ligne
21   call MPI_TYPE_COMMIT(type_ligne,code)
```

```
22    ! Envoi
23    if ( rang == 0 ) then
24      call MPI_SEND(a(2,1),nb_colonnes,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
25
26    ! Réception dans l'avant-dernière ligne
27    elseif ( rang == 1 ) then
28      call MPI_RECV(a(nb_lignes-1,1),1,type_ligne,0,etiquette,&
29                    MPI_COMM_WORLD,statut,code)
30    end if
31
32    ! Libère le type type_ligne
33    call MPI_TYPE_FREE(type_ligne,code)
34
35    call MPI_FINALIZE(code)
36
37  end program ligne
```

6 – Types de données dérivés

6.5 – Exemples

6.5.3 – Type « bloc d'une matrice »

```
1  program bloc
2    use mpi
3    implicit none
4
5    integer, parameter                        :: nb_lignes=5,nb_colonnes=6
6    integer, parameter                        :: etiquette=100
7    integer, parameter                        :: nb_lignes_bloc=2,nb_colonnes_bloc=3
8    real, dimension(nb_lignes,nb_colonnes):: a
9    integer, dimension(MPI_STATUS_SIZE)    :: statut
10   integer                                   :: rang,code,type_bloc
11
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   ! Initialisation de la matrice sur chaque processus
16   a(:,:) = real(rang)
17
18   ! Création du type type_bloc
19   call MPI_TYPE_VECTOR(nb_colonnes_bloc,nb_lignes_bloc,nb_lignes,&
20                        MPI_REAL,type_bloc,code)
21
22   ! Validation du type type_bloc
23   call MPI_TYPE_COMMIT(type_bloc,code)
```

```
24    ! Envoi d'un bloc
25    if ( rang == 0 ) then
26      call MPI_SEND(a(1,1),1,type_bloc,1,etiquette,MPI_COMM_WORLD,code)
27
28    ! Réception du bloc
29    elseif ( rang == 1 ) then
30      call MPI_RECV(a(nb_lignes-1,nb_colonnes-2),1,type_bloc,0,etiquette,&
31                    MPI_COMM_WORLD,statut,code)
32    end if
33
34    ! Libération du type type_bloc
35    call MPI_TYPE_FREE(type_bloc,code)
36
37    call MPI_FINALIZE(code)
38
39  end program bloc
```

6 – Types de données dérivés
6.6 – Types homogènes à pas variable

Types homogènes à pas variable

- `MPI_TYPE_INDEXED()` permet de créer une structure de données composée d'une séquence de blocs contenant un nombre variable d'éléments et séparés par un pas variable en mémoire. Ce dernier est exprimé en **éléments**.
- `MPI_TYPE_CREATE_HINDEXED()` a la même fonctionnalité que `MPI_TYPE_INDEXED()` sauf que le pas séparant deux blocs de données est exprimé en **octets**.
  Cette instruction est utile lorsque le type générique n'est pas un type de base MPI (`MPI_INTEGER`, `MPI_REAL`, ...) mais un type plus complexe construit avec les sous-programmes MPI vus précédemment. On ne peut exprimer alors le pas en nombre d'éléments du type générique d'où le recours à `MPI_TYPE_CREATE_HINDEXED()`.
- Pour `MPI_TYPE_CREATE_HINDEXED()`, comme pour `MPI_TYPE_CREATE_HVECTOR()`, utilisez `MPI_TYPE_SIZE()` ou `MPI_TYPE_GET_EXTENT()` pour obtenir de façon portable la taille du pas en nombre d'octets.

nb=3, longueurs_blocs=(2,1,3), déplacements=(0,3,7)

FIGURE 24 – Le constructeur MPI_TYPE_INDEXED

```
integer,intent(in)                :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
! Attention les déplacements sont donnés en éléments
integer,intent(in),dimension(nb) :: deplacements
integer,intent(in)                :: ancien_type

integer,intent(out)               :: nouveau_type,code

call MPI_TYPE_INDEXED(nb,longueurs_blocs,deplacements,ancien_type,nouveau_type,code)
```

nb=4, longueurs_blocs=(2,1,2,1), déplacements=(2,10,14,24)

FIGURE 25 – Le constructeur MPI_TYPE_CREATE_HINDEXED

```
integer,intent(in)                                          :: nb
integer,intent(in),dimension(nb)                            :: longueurs_blocs
! Attention les déplacements sont donnés en octets
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: deplacements
integer,intent(in)                                          :: ancien_type

integer,intent(out)                                         :: nouveau_type,code

call MPI_TYPE_CREATE_HINDEXED(nb,longueurs_blocs,deplacements,
                              ancien_type,nouveau_type,code)
```

Exemple : matrice triangulaire

Dans l'exemple suivant, chacun des deux processus :

1. initialise sa matrice (nombres croissants positifs sur le processus 0 et négatifs décroissants sur le processus 1) ;
2. construit son type de données (*datatype*) : matrice triangulaire (supérieure pour le processus 0 et inférieure pour le processus 1) ;
3. envoie sa matrice triangulaire à l'autre et reçoit une matrice triangulaire qu'il stocke à la place de celle qu'il a envoyée via l'instruction `MPI_SENDRECV_REPLACE()` ;
4. libère ses ressources et quitte MPI.

Avant

Processus 0

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Processus 1

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

Après

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

FIGURE 26 – Échanges entre les 2 processus

```
1  program triangle
2    use mpi
3    implicit none
4
5    integer,parameter                    :: n=8,etiquette=100
6    real,dimension(n,n)                  :: a
7    integer,dimension(MPI_STATUS_SIZE)  :: statut
8    integer                              :: i,code
9    integer                              :: rang,type_triangle
10   integer,dimension(n)                 :: longueurs_blocs,deplacements
11
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   ! Initialisation de la matrice sur chaque processus
16   a(:,:) = reshape( (/ (sign(i,-rang),i=1,n*n) /), (/n,n/))
17
18   ! Création du type matrice triangulaire sup pour le processus 0
19   ! et du type matrice triangulaire inférieure pour le processus1
20   if (rang == 0) then
21      longueurs_blocs(:) = (/ (i-1,i=1,n) /)
22      deplacements(:)    = (/ (n*(i-1),i=1,n) /)
23   else
24      longueurs_blocs(:) = (/ (n-i,i=1,n) /)
25      deplacements(:)    = (/ (n*(i-1)+i,i=1,n) /)
26   endif
27
28   call MPI_TYPE_INDEXED(n,longueurs_blocs,deplacements,MPI_REAL,type_triangle,code)
29   call MPI_TYPE_COMMIT(type_triangle,code)
30
31   ! Permutation des matrices triangulaires supérieure et inférieure
32   call MPI_SENDRECV_REPLACE(a,1,type_triangle,mod(rang+1,2),etiquette,mod(rang+1,2), &
33                             etiquette,MPI_COMM_WORLD,statut,code)
34
35   ! Libération du type triangle
36   call MPI_TYPE_FREE(type_triangle,code)
37   call MPI_FINALIZE(code)
38 end program triangle
```

6 – Types de données dérivés
6.7 – Construction de sous-tableaux

Construction de sous-tableaux

Le sous-programme `MPI_TYPE_CREATE_SUBARRAY()` permet de créer un sous-tableau à partir d'un tableau.

Rappels sur le vocabulaire relatif aux tableaux en Fortran 95

- Le rang d'un tableau est son nombre de dimensions.
- L'étendue d'un tableau est son nombre d'éléments dans une dimension.
- Le profil d'un tableau est un vecteur dont chaque élément est l'étendue du tableau dans la dimension correspondante.

Soit par exemple le tableau `T(10,0:5,-10:10)`. Son rang est 3, son étendue dans la première dimension est 10, dans la seconde 6 et dans la troisième 21, son profil est le vecteur (10,6,21).

```
integer,intent(in)                    :: nb_dims
integer,dimension(ndims),intent(in) :: profil_tab,profil_sous_tab,coord_debut
integer,intent(in)                    :: ordre,ancien_type
integer,intent(out)                   :: nouveau_type,code
call MPI_TYPE_CREATE_SUBARRAY(nb_dims,profil_tab,profil_sous_tab,coord_debut,
                              ordre,ancien_type,nouveau_type,code)
```

Description des arguments

- `nb_dims` : rang du tableau
- `profil_tab` : profil du tableau à partir duquel on va extraire un sous-tableau
- `profil_sous_tab` : profil du sous-tableau
- `coord_debut` : coordonnées de départ si les indices du tableau commençaient à 0. Par exemple, si on veut que les coordonnées de départ du sous-tableau soient `tab(2,3)`, il faut que `coord_debut(:)=(/ 1,2 /)`
- `ordre` : ordre de stockage des éléments
  - `MPI_ORDER_FORTRAN` spécifie le mode de stockage en Fortran, c.-à-d. suivant les colonnes
  - `MPI_ORDER_C` spécifie le mode de stockage en C, c.-à-d. suivant les lignes

AVANT

1	5	9
2	6	10
3	7	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	-7	-11
-4	-8	-12

Processus 1

APRES

1	5	9
-7	-11	10
-8	-12	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	2	6
-4	3	7

Processus 1

FIGURE 27 – Échanges entre les 2 processus

```
1  program subarray
2
3    use mpi
4    implicit none
5
6    integer,parameter                          :: nb_lignes=4,nb_colonnes=3,&
7                                                   etiquette=1000,nb_dims=2
8    integer                                    :: code,rang,type_sous_tab,i
9    integer,dimension(nb_lignes,nb_colonnes)   :: tab
10   integer,dimension(nb_dims)                 :: profil_tab,profil_sous_tab,coord_debut
11   integer,dimension(MPI_STATUS_SIZE)         :: statut
12
13   call MPI_INIT(code)
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
15
16   ! Initialisation du tableau tab sur chaque processus
17   tab(:,:) = reshape( (/ (sign(i,-rang),i=1,nb_lignes*nb_colonnes) /) , &
18                       (/ nb_lignes,nb_colonnes /) )
19
```

```
20    ! Profil du tableau tab à partir duquel on va extraire un sous-tableau
21    profil_tab(:) = shape(tab)
22    ! La fonction F95 shape donne le profil du tableau passé en argument.
23    ! ATTENTION, si le tableau concerné n'a pas été alloué sur tous les processus,
24    ! il faut mettre explicitement le profil du tableau pour qu'il soit connu
25    ! sur tous les processus, soit profil_tab(:) = (/ nb_lignes,nb_colonnes) /)
26
27    ! Profil du sous-tableau
28    profil_sous_tab(:) = (/ 2,2 /)
29
30    ! Coordonnées de départ du sous-tableau
31    ! Pour le processus 0 on part de l'élément tab(2,1)
32    ! Pour le processus 1 on part de l'élément tab(3,2)
33    coord_debut(:) = (/ rang+1,rang /)
34
35    ! Création du type dérivé type_sous_tab
36    call MPI_TYPE_CREATE_SUBARRAY(nb_dims,profil_tab,profil_sous_tab,coord_debut,&
37                                  MPI_ORDER_FORTRAN,MPI_INTEGER,type_sous_tab,code)
38    call MPI_TYPE_COMMIT(type_sous_tab,code)
39
40    ! Permutation du sous-tableau
41    call MPI_SENDRECV_REPLACE(tab,1,type_sous_tab,mod(rang+1,2),etiquette,&
42                              mod(rang+1,2),etiquette,MPI_COMM_WORLD,statut,code)
43
44    call MPI_TYPE_FREE(type_sous_tab,code)
45
46    call MPI_FINALIZE(code)
47
48  end program subarray
```

6 – Types de données dérivés
6.8 – Types hétérogènes

- Le sous-programme MPI_TYPE_CREATE_STRUCT() permet de créer une séquence de blocs de données en précisant le **type**, le **nombre d'éléments** et le **pas** de chaque blocs.
- Il s'agit du constructeur de types le plus complet. Il généralise MPI_TYPE_INDEXED() en permettant de définir un **type** différent pour chaque bloc.

nb=5, longueurs_blocs=(3,1,5,1,1), déplacements=(0,7,11,21,26),
anciens_types=(type1,type2,type3,type1,type3)

FIGURE 28 – Le constructeur MPI_TYPE_CREATE_STRUCT

```
integer,intent(in)                                          :: nb
integer,intent(in),dimension(nb)                            :: longueurs_blocs
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: deplacements
integer,intent(in),dimension(nb)                            :: anciens_types

integer, intent(out)                                        :: nouveau_type,code

call MPI_TYPE_CREATE_STRUCT(nb,longueurs_blocs,deplacements,
                            anciens_types,nouveau_type,code)
```

Calcul du déplacement entre deux variables

- MPI_TYPE_CREATE_STRUCT() est utile notamment pour créer des types MPI correspondant à des types dérivés Fortran ou à des structures C.
- L'alignement en mémoire des structures de données hétérogènes dépend de l'architecture et du compilateur.
- Le déplacement entre deux éléments d'un type dérivé Fortran ou d'une structure C peut-être obtenu en calculant la différence entre leurs adresses mémoires.
- Le sous-programme MPI_GET_ADDRESS() permet de récupérer l'adresse d'une variable. C'est l'équivalent de l'opérateur de référencement (&) du C.
- Attention, même en C, il est préférable d'utiliser ce sous-programme MPI pour des raisons de portabilité.

```
<type>,intent(in)                              :: variable
integer(kind=MPI_ADDRESS_KIND),intent(out) :: adresse_variable
integer,intent(out)                            :: code

call MPI_GET_ADDRESS(variable,adresse_variable,code)
```

```
1  program Interaction_Particules
2
3    use mpi
4    implicit none
5
6    integer, parameter                          :: n=1000,etiquette=100
7    integer, dimension(MPI_STATUS_SIZE)          :: statut
8    integer                                     :: rang,code,type_particule,i
9    integer, dimension(4)                       :: types,longueurs_blocs
10   integer(kind=MPI_ADDRESS_KIND), dimension(4) :: deplacements,adresses
11
12   type Particule
13      character(len=5)                         :: categorie
14      integer                                  :: masse
15      real, dimension(3)                       :: coords
16      logical                                  :: classe
17   end type Particule
18   type(Particule), dimension(n)               :: p,temp_p
19
20   call MPI_INIT(code)
21   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
22
23   ! Construction du type de données
24   types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
25   longueurs_blocs = (/5,1,3,1/)
```

```
26    call MPI_GET_ADDRESS(p(1)%categorie,adresses(1),code)
27    call MPI_GET_ADDRESS(p(1)%masse,adresses(2),code)
28    call MPI_GET_ADDRESS(p(1)%coords,adresses(3),code)
29    call MPI_GET_ADDRESS(p(1)%classe,adresses(4),code)
30
31    ! Calcul des déplacements relatifs à l'adresse de départ
32    do i=1,4
33       deplacements(i)=adresses(i) - adresses(1)
34    end do
35    call MPI_TYPE_CREATE_STRUCT(4,longueurs_blocs,deplacements,types,type_particule, &
36                                code)
37    ! Validation du type structuré
38    call MPI_TYPE_COMMIT(type_particule,code)
39    ! Initialisation des particules pour chaque processus
40    ....
41    ! Envoi des particules de 0 vers 1
42    if (rang == 0) then
43       call MPI_SEND(p(1)%categorie,n,type_particule,1,etiquette,MPI_COMM_WORLD,code)
44    else
45       call MPI_RECV(temp_p(1)%categorie,n,type_particule,0,etiquette,MPI_COMM_WORLD, &
46                     statut,code)
47    endif
48
49    ! Libération du type
50    call MPI_TYPE_FREE(type_particule,code)
51
52    call MPI_FINALIZE(code)
53
54 end program Interaction_Particules
```

6 – Types de données dérivés
6.9 – Taille des types de données

- MPI_TYPE_SIZE() retourne le nombre d'octets nécessaire pour envoyer les éléments d'un type de données. Cette valeur ne tient pas compte des *trous* présents dans le type de données.

```
integer, intent(in)  :: type
integer, intent(out) :: taille, code

call MPI_TYPE_SIZE(type,taille,code)
```

- L'étendue d'un type est l'espace mémoire occupé par le type (en octets). Cette valeur intervient directement pour calculer la position du prochain élément en mémoire (c'est-à-dire le **pas** entre des éléments successifs).

```
integer, intent(in)                          :: type
integer(kind=MPI_ADDRESS_KIND),intent(out):: borne_inf,etendue
integer, intent(out)                         :: code

call MPI_TYPE_GET_EXTENT(type,borne_inf,etendue,code)
```

- L'étendue est un **paramètre** du type de données. Par défaut, c'est généralement l'intervalle entre le premier et le dernier élément mémoire du type (bornes incluses et en tenant compte de l'alignement mémoire). On peut modifier l'étendue d'un type pour créer un nouveau type adapté du précédent avec MPI_TYPE_CREATE_RESIZED(). Cela permet de choisir le **pas** entre des éléments successifs.

```
integer, intent(in)                          :: ancien_type
integer(kind=MPI_ADDRESS_KIND),intent(in) :: nouvelle_borne_inf,nouvelle_etendue
integer, intent(out)                         :: nouveau_type,code

call MPI_TYPE_CREATE_RESIZED(ancien_type,nouvelle_borne_inf,nouvelle_etendue,
                             nouveau_type,code)
```

Exemple 1 : MPI_TYPE_INDEXED(2,(/2,1/),(/1,4/),MPI_INTEGER,type,code)

Type dérivé :

Deux éléments successifs :

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.

`taille = 12 (3 entiers); borne_inf = 4 (1 entier); etendue = 16 (4 entiers)`

Exemple 2 : MPI_TYPE_VECTOR(3,1,nb_lignes,MPI_INTEGER,type_demi_ligne,code)

Vue 2D :

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

Vue 1D :

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.

`taille = 12 (3 entiers); borne_inf = 0; etendue = 44 (11 entiers)`

```
1  program demi_ligne
2    use mpi
3    implicit none
4    integer, parameter                          :: nb_lignes=5,nb_colonnes=6, &
5                                                   demi_ligne=nb_colonnes/2,etiquette=1000
6    integer, dimension(nb_lignes,nb_colonnes) :: a
7    integer                                     :: rang,i,taille_integer,code
8    integer                                     :: type_demi_ligne1,type_demi_ligne2
9    integer(kind=MPI_ADDRESS_KIND)              :: borne_inf1,etendue1,borne_inf2,etendue2
10   integer, dimension(MPI_STATUS_SIZE)         :: statut
11
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   ! Initialisation de la matrice A sur chaque processus
16   a(:,:) = reshape( (/ (sign(i,-rang),i=1,nb_lignes*nb_colonnes) /), &
17                     (/ nb_lignes,nb_colonnes /) )
18
19   ! Construction du type derivé type_demi_ligne1
20   call MPI_TYPE_VECTOR(demi_ligne,1,nb_lignes,MPI_INTEGER,type_demi_ligne1,code)
21
22   ! Connaître la taille du type de base MPI_INTEGER
23   call MPI_TYPE_SIZE(MPI_INTEGER,taille_integer,code)
24
25   ! Informations sur le type dérivé type_demi_ligne1
26   call MPI_TYPE_GET_EXTENT(type_demi_ligne1,borne_inf1,etendue1,code)
27   if (rang == 0) print *, "type_demi_ligne1: borne_inf=",borne_inf1,", etendue=",etendue1
28
29   ! Construction du type derivé type_demi_ligne2
30   borne_inf2 = 0
31   etendue2 = taille_integer
32   call MPI_TYPE_CREATE_RESIZED(type_demi_ligne1,borne_inf2,etendue2,&
33                                type_demi_ligne2,code)
```

```
34    ! Informations sur le type dérivé type_demi_ligne2
35    call MPI_TYPE_GET_EXTENT(type_demi_ligne2,borne_inf2,etendue2,code)
36    if (rang == 0) print *, "type_demi_ligne2: borne_inf=",borne_inf2,", etendue=",etendue2
37
38    ! Validation du type type_demi_ligne2
39    call MPI_TYPE_COMMIT(type_demi_ligne2,code)
40
41    if (rang == 0) then
42       ! Envoi de la matrice A au processus 1 avec le type type_demi_ligne2
43       call MPI_SEND(A(1,1),2,type_demi_ligne2,1,etiquette, &
44                     MPI_COMM_WORLD,code)
45    else
46       ! Réception pour le processus 1 dans la matrice A
47       call MPI_RECV(A(1,nb_colonnes-1),6,MPI_INTEGER,0,etiquette, &
48                     MPI_COMM_WORLD,statut,code)
49       print *,'Matrice A sur le processus 1'
50       do i=1,nb_lignes
51          print *,A(i,:)
52       end do
53    end if
54
55    call MPI_FINALIZE(code)
56 end program demi_ligne
```

```
> mpiexec -n 2 demi_ligne
 type_demi_ligne1: borne_inf=0, etendue=44
 type_demi_ligne2: borne_inf=0, etendue=4
```

```
Matrice A sur le processus 1
 -1  -6 -11 -16  1  12
 -2  -7 -12 -17  6 -27
 -3  -8 -13 -18 11 -28
 -4  -9 -14 -19  2 -29
 -5 -10 -15 -20  7 -30
```

6 – Types de données dérivés
6.10 – Conclusion

Conclusion

- Les types dérivés MPI sont de puissants mécanismes portables de description de données.
- Ils permettent, lorsqu'ils sont associés à des instructions comme `MPI_SENDRECV()`, de simplifier l'écriture de sous-programmes d'échanges interprocessus.
- L'association des types dérivés et des topologies (décrites dans l'un des prochains chapitres) fait de MPI l'outil idéal pour tous les problèmes de décomposition de domaines avec des maillages réguliers ou irréguliers.

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs
7.1 Introduction 140
7.2 Exemple 141
7.3 Communicateur par défaut 142
7.4 Groupes et communicateurs 143
7.5 Partitionnement d'un communicateur 144
7.6 Communicateur construit à partir d'un groupe 148
7.7 Topologies 149

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

7 – Communicateurs
7.1 – Introduction

Introduction

Il s'agit de partitionner un ensemble de processus afin de créer des sous-ensembles sur lesquels on puisse effectuer des opérations telles que des communications point à point, collectives, etc. Chaque sous-ensemble ainsi créé aura son propre espace de communication.

Figure 29 – Partitionnement d'un communicateur

7 – Communicateurs
7.2 – Exemple

Exemple

Par exemple, on veut diffuser un message collectif aux processus de rang pair et un autre aux processus de rang impair.

- Boucler sur des *send/recv* peut être très pénalisant surtout si le nombre de processus est élevé. De plus un test serait obligatoire dans la boucle pour savoir si le rang du processus auquel le processus émetteur doit envoyer le message est pair ou impair.
- Une solution est de créer un communicateur regroupant les processus pairs et un autre regroupant les processus impairs et d'initier les communications collectives à l'intérieur de ces groupes.

7 – Communicateurs
7.3 – Communicateur par défaut

Communicateur par défaut

- On ne peut créer un communicateur qu'à partir d'un autre communicateur. Le premier sera créé à partir de `MPI_COMM_WORLD`.
- En effet, à l'appel du sous-programme `MPI_INIT()`, un communicateur est créé par défaut.
- Son identificateur `MPI_COMM_WORLD` est un entier défini dans les fichiers d'en-tête.
- Il est créé pour toute la durée d'exécution du programme à l'appel du sous-programme `MPI_INIT()`
- Il ne peut être détruit que via l'appel à `MPI_FINALIZE()`
- Par défaut, il fixe donc la portée des communications point à point et collectives à tous les processus de l'application

7 – Communicateurs
7.4 – Groupes et communicateurs

Groupes et communicateurs

- Un communicateur est constitué :
  - d'un groupe, qui est un ensemble ordonné de processus ;
  - d'un contexte de communication mis en place à l'appel du sous-programme de construction du communicateur, qui permet de délimiter l'espace de communication.
- Les contextes de communication sont gérés par MPI (le programmeur n'a aucune action sur eux : c'est un attribut « caché »)
- Dans la bibliothèque MPI, divers sous-programmes existent pour construire des communicateurs : `MPI_COMM_CREATE()`, `MPI_COMM_DUP()`, `MPI_COMM_SPLIT()`
- Les constructeurs de communicateurs sont des opérateurs collectifs (qui engendrent des communications entre les processus)
- Les communicateurs que le programmeur crée peuvent être gérés dynamiquement et, de même qu'il est possible d'en créer, il est possible d'en détruire en utilisant le sous-programme `MPI_COMM_FREE()`

7 – Communicateurs
7.5 – Partitionnement d'un communicateur

Partitionnement d'un communicateur

Pour résoudre le problème de l'exemple, nous allons :

- partitionner le communicateur en processus de rang pair et d'autre part en processus de rang impair ;
- ne diffuser un message collectif qu'aux processus de rang pair et un autre qu'aux processus de rang impair.

FIGURE 30 – Création/destruction d'un communicateur

Partitionnement d'un communicateur avec MPI_COMM_SPLIT()

Le sous-programme MPI_COMM_SPLIT() permet de :

- partitionner un communicateur donné en autant de communicateurs que l'on veut
- donner le même nom à tous les communicateurs : il aura la valeur du communicateur dans lequel se trouve le processus courant
- Méthode :
  1. définir une valeur couleur associant à chaque processus le numéro du communicateur auquel il appartiendra
  2. définir une valeur clef permettant de numéroter les processus dans chaque communicateur
  3. créer la partition où chaque communicateur s'appelle nouveau_comm

```
integer, intent(in)  :: comm, couleur, clef
integer, intent(out) :: nouveau_comm, code
call MPI_COMM_SPLIT(comm,couleur,clef,nouveau_comm,code)
```

Un processus qui s'attribue une couleur égale à la valeur MPI_UNDEFINED aura pour nouveau_com le communicateur invalide MPI_COMM_NULL.

Exemple

Voyons comment procéder pour construire le communicateur qui va subdiviser l'espace de communication entre processus de rangs pairs et impairs, via le constructeur `MPI_COMM_SPLIT()`.

processus	a	b	c	d	e	f	g	h
rang_monde	0	1	2	3	4	5	6	7
couleur	0	1	0	1	0	1	0	1
clef	0	1	-1	3	4	-1	6	7
rang_pairs_imp	1	1	0	2	2	0	3	3

FIGURE 31 – Construction du communicateur `CommPairsImpairs` avec `MPI_COMM_SPLIT()`

```
1  program PairsImpairs
2    use mpi
3    implicit none
4
5    integer, parameter :: m=16
6    integer            :: clef,CommPairsImpairs
7    integer            :: rang_dans_monde,code
8    real, dimension(m) :: a
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang_dans_monde,code)
12
13   ! Initialisation du vecteur A
14   a(:)=0.
15   if(rang_dans_monde == 2) a(:)=2.
16   if(rang_dans_monde == 5) a(:)=5.
17
18   clef = rang_dans_monde
19   if (rang_dans_monde == 2 .OR. rang_dans_monde == 5 ) then
20      clef=-1
21   end if
22
23   ! Création des communicateurs pair et impair en leur donnant une même dénomination
24   call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(rang_dans_monde,2),clef,CommPairsImpairs,code)
25
26   ! Diffusion du message par le processus 0 de chaque communicateur aux processus
27   ! de son groupe
28   call MPI_BCAST(a,m,MPI_REAL,0,CommPairsImpairs,code)
29
30   ! Destruction des communicateurs
31   call MPI_COMM_FREE(CommPairsImpairs,code)
32   call MPI_FINALIZE(code)
33 end program PairsImpairs
```

7 – Communicateurs

7.6 – Communicateur construit à partir d'un groupe

Communicateur construit à partir d'un groupe

- On peut aussi construire un communicateur en définissant un groupe de processus.
  Procédure : appel à `MPI_COMM_GROUP()`, `MPI_GROUP_INCL()`, `MPI_COMM_CREATE()`, `MPI_GROUP_FREE()`
- Cette méthode présente dans le cas de l'exemple, divers inconvénients, car elle impose de :
  - nommer différemment les deux communicateurs (par exemple `comm_pair` et `comm_impair`) ;
  - passer par les groupes pour construire ces deux communicateurs ;
  - laisser MPI ordonner le rang des processus dans ces deux communicateurs ;
  - tester le communicateur dans lequel on se trouve.

7 – Communicateurs
7.7 – Topologies

Topologies

- Dans la plupart des applications, plus particulièrement dans les méthodes de décomposition de domaine où l'on fait correspondre le domaine de calcul à la grille de processus, il est intéressant de pouvoir disposer les processus suivant une topologie régulière
- MPI permet de définir des topologies virtuelles du type cartésien ou graphe
  - Topologies de type cartésien
    - chaque processus est défini dans une grille de processus ;
    - chaque processus a un voisin dans la grille ;
    - la grille peut être périodique ou non ;
    - les processus sont identifiés par leurs coordonnées dans la grille.
  - Topologies de type graphe
    - généralisation à des topologies plus complexes.

7 – Communicateurs
7.7 – Topologies
7.7.1 – Topologies cartésiennes

Topologies cartésiennes

- Une topologie cartésienne est définie à partir d'un communicateur donné comm_ancien, en appelant le sous-programme MPI_CART_CREATE().
- On définit :
  - un entier ndims représentant le nombre de dimensions de la grille
  - un tableau d'entiers dims de dimension ndims indiquant le nombre de processus dans chaque dimension
  - un tableau de logiques de dimension ndims indiquant la périodicité dans chaque dimension
  - un logique reorganisation indiquant la numérotation des processus

```
integer, intent(in)                    :: comm_ancien, ndims
integer, dimension(ndims),intent(in)   :: dims
logical, dimension(ndims),intent(in)   :: periods
logical, intent(in)                    :: reorganisation

integer, intent(out)                   :: comm_nouveau, code

call MPI_CART_CREATE(comm_ancien, ndims,dims,periods,reorganisation,comm_nouveau,code)
```

Exemple

Exemple sur une grille comportant 4 domaines suivant x et 2 suivant y, périodique en y.

```
use mpi
integer                    :: comm_2D, code
integer, parameter         :: ndims = 2
integer, dimension(ndims)  :: dims
logical, dimension(ndims)  :: periods
logical                    :: reorganisation

..............................................

dims(1) = 4
dims(2) = 2
periods(1) = .false.
periods(2) = .true.
reorganisation = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_2D,code)
```

Si `reorganisation = .false.` alors le rang des processus dans le nouveau communicateur (comm_2D) est le même que dans l'ancien communicateur (`MPI_COMM_WORLD`).
Si `reorganisation = .true.`, l'implémentation MPI choisit l'ordre des processus.

FIGURE 32 – Topologie cartésienne 2D périodique en y

Exemple 3D

Exemple sur une grille 3D comportant 4 domaines suivant x, 2 suivant y et 2 suivant z, non périodique.

```
use mpi
integer                    :: comm_3D,code
integer, parameter         :: ndims = 3
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical                    :: reorganisation

..............................................

dims(1) = 4
dims(2) = 2
dims(3) = 2
periods(:) = .false.
reorganisation = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_3D,code)
```

FIGURE 33 – Topologie cartésienne 3D non périodique

Distribution des processus

Dans une topologie cartésienne, le sous-programme MPI_DIMS_CREATE() retourne le nombre de processus dans chaque dimension de la grille en fonction du nombre total de processus.

```
integer, intent(in)                        :: nb_procs, ndims
integer, dimension(ndims),intent(inout) :: dims
integer, intent(out)                       :: code

call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)
```

Remarque : si les valeurs de dims en entrée valent toutes 0, cela signifie qu'on laisse à MPI le choix du nombre de processus dans chaque direction en fonction du nombre total de processus.

dims en entrée	call MPI_DIMS_CREATE	dims en sortie
(0,0)	(8,2,dims,code)	(4,2)
(0,0,0)	(16,3,dims,code)	(4,2,2)
(0,4,0)	(16,3,dims,code)	(2,4,2)
(0,3,0)	(16,3,dims,code)	error

Rang d'un processus

Dans une topologie cartésienne, le sous-programme MPI_CART_RANK() retourne le rang du processus associé aux coordonnées dans la grille.

```
integer, intent(in)                    :: comm_nouveau
integer, dimension(ndims),intent(in) :: coords
integer, intent(out)                   :: rang, code

call MPI_CART_RANK(comm_nouveau,coords,rang,code)
```

FIGURE 34 – Topologie cartésienne 2D périodique en y

```
coords(1)=dims(1)-1
do i=0,dims(2)-1
  coords(2) = i
  call MPI_CART_RANK(comm_2D,coords,rang(i),code)
end do
.............................................
i=0,en entrée coords=(3,0),en sortie rang(0)=6.
i=1,en entrée coords=(3,1),en sortie rang(1)=7.
```

Coordonnées d'un processus

Dans une topologie cartésienne, le sous-programme MPI_CART_COORDS() retourne les coordonnées d'un processus de rang donné dans la grille.

```
integer, intent(in)                    :: comm_nouveau, rang, ndims

integer, dimension(ndims),intent(out) :: coords
integer, intent(out)                   :: code

call MPI_CART_COORDS(comm_nouveau, rang, ndims, coords, code)
```

FIGURE 35 – Topologie cartésienne 2D périodique en y

```
if (mod(rang,2) == 0) then
  call MPI_CART_COORDS(comm_2D,rang,2,coords,code)
end if
.............................................
En entrée, les valeurs de rang sont : 0,2,4,6.
En sortie, les valeurs de coords sont :
(0,0),(1,0),(2,0),(3,0).
```

Rang des voisins

Dans une topologie cartésienne, un processus appelant le sous-programme MPI_CART_SHIFT() se voit retourner le rang de ses processus voisins dans une direction donnée.

```
integer, intent(in)  :: comm_nouveau, direction, pas
integer, intent(out) :: rang_precedent,rang_suivant
integer, intent(out) :: code

call MPI_CART_SHIFT(comm_nouveau, direction, pas, rang_precedent, rang_suivant, code)
```

- Le paramètre direction correspond à l'axe du déplacement (xyz).
- Le paramètre pas correspond au pas du déplacement.
- Si un rang n'a pas de voisin précédant (resp. suivant) dans la direction demandée, alors la valeur du rang précédant (resp. suivant) sera MPI_PROC_NULL.

FIGURE 36 – Appel du sous-programme MPI_CART_SHIFT()

```
call MPI_CART_SHIFT(comm_2D,0,1,rang_gauche,rang_droit,code)
.............................................
Pour le processus 2, rang_gauche=0, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_2D,1,1,rang_bas,rang_haut,code)
.............................................
Pour le processus 2, rang_bas=3, rang_haut=3
```

FIGURE 37 – Appel du sous-programme MPI_CART_SHIFT()

```
call MPI_CART_SHIFT(comm_3D,0,1,rang_gauche,rang_droit,code)
.............................................
Pour le processus 0, rang_gauche=-1, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_3D,1,1,rang_bas,rang_haut,code)
.............................................
Pour le processus 0, rang_bas=-1, rang_haut=2
```

```
call MPI_CART_SHIFT(comm_3D,2,1,rang_avant,rang_arriere,code)
.............................................
Pour le processus 0, rang_avant=-1, rang_arriere=1
```

```
1  program decomposition
2    use mpi
3    implicit none
4
5    integer                    :: rang_ds_topo,nb_procs
6    integer                    :: code,comm_2D
7    integer, dimension(4)      :: voisin
8    integer, parameter         :: N=1,E=2,S=3,W=4
9    integer, parameter         :: ndims = 2
10   integer, dimension (ndims) :: dims,coords
11   logical, dimension (ndims) :: periods
12   logical                    :: reorganisation
13
14   call MPI_INIT(code)
15
16   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
17
18   ! Connaître le nombre de processus suivant x et y
19   dims(:) = 0
20
21   call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)
```

```
22    ! Création grille 2D périodique en y
23    periods(1) = .false.
24    periods(2) = .true.
25    reorganisation = .false.
26
27    call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_2D,code)
28
29    ! Connaître mes coordonnées dans la topologie
30    call MPI_COMM_RANK(comm_2D,rang_ds_topo,code)
31    call MPI_CART_COORDS(comm_2D,rang_ds_topo,ndims,coords,code)
32
33    ! Recherche de mes voisins Ouest et Est
34    call MPI_CART_SHIFT(comm_2D,0,1,voisin(W),voisin(E),code)
35
36    ! Recherche de mes voisins Sud et Nord
37    call MPI_CART_SHIFT(comm_2D,1,1,voisin(S),voisin(N),code)
38
39    call MPI_FINALIZE(code)
40
41  end program decomposition
```

7 – Communicateurs
7.7 – Topologies
7.7.2 – Subdiviser une topologie cartésienne

Subdiviser une topologie cartésienne

- La question est de savoir comment dégénérer une topologie cartésienne 2D ou 3D de processus en une topologie cartésienne respectivement 1D ou 2D.
- Pour MPI, dégénérer une topologie cartésienne 2D (ou 3D) revient à créer autant de communicateurs qu'il y a de lignes ou de colonnes (resp. de plans) dans la grille cartésienne initiale.
- L'intérêt majeur est de pouvoir effectuer des opérations collectives restreintes à un sous-ensemble de processus appartenant à :
  - une même ligne (ou colonne), si la topologie initiale est 2D ;
  - un même plan, si la topologie initiale est 3D.

FIGURE 38 – Deux exemples de distribution de données dans une topologie 2D dégénérée

Subdiviser une topologie cartésienne

Il existe deux façons de faire pour dégénérer une topologie :

- en utilisant le sous-programme général `MPI_COMM_SPLIT()` ;
- en utilisant le sous-programme `MPI_CART_SUB()` prévu à cet effet.

```
logical,intent(in),dimension(NDim) :: conserve_dims
integer,intent(in)                 :: CommCart
integer,intent(out)                :: CommCartD, code
call MPI_CART_SUB(CommCart,conserve_dims,CommCartD,code)
```

FIGURE 39 – Représentation initiale d'un tableau V dans la grille 2D et représentation finale après la distribution de celui-ci sur la grille 2D dégénérée

```
1  program CommCartSub
2    use mpi
3    implicit none
4
5    integer                    :: Comm2D,Comm1D,rang,code
6    integer,parameter          :: NDim2D=2
7    integer,dimension(NDim2D)  :: Dim2D,Coord2D
8    logical,dimension(NDim2D)  :: Periode,conserve_dims
9    logical                    :: Reordonne
10   integer,parameter          :: m=4
11   real, dimension(m)         :: V(:)=0.
12   real                       :: W=0.
```

```
13    call MPI_INIT(code)
14
15    ! Création de la grille 2D initiale
16    Dim2D(1) = 4
17    Dim2D(2) = 3
18    Periode(:) = .false.
19    ReOrdonne = .false.
20    call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Periode,ReOrdonne,Comm2D,code)
21    call MPI_COMM_RANK(Comm2D,rang,code)
22    call MPI_CART_COORDS(Comm2D,rang,NDim2D,Coord2D,code)
23
24    ! Initialisation du vecteur V
25    if (Coord2D(1) == 1) V(:)=real(rang)
26
27    ! Chaque ligne de la grille doit être une topologie cartésienne 1D
28    conserve_dims(1) = .true.
29    conserve_dims(2) = .false.
30    ! Subdivision de la grille cartésienne 2D
31    call MPI_CART_SUB(Comm2D,conserve_dims,Comm1D,code)
32
33    ! Les processus de la colonne 2 distribuent le vecteur V aux processus de leur ligne
34    call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,code)
35
36    print '("Rang : ",I2," ; Coordonnees : (",I1,",",I1,") ; W = ",F2.0)', &
37          rang,Coord2D(1),Coord2D(2),W
38
39    call MPI_FINALIZE(code)
40  end program CommCartSub
```

```
> mpiexec -n 12 CommCartSub
Rang :  0 ; Coordonnees : (0,0) ; W = 3.
Rang :  1 ; Coordonnees : (0,1) ; W = 4.
Rang :  3 ; Coordonnees : (1,0) ; W = 3.
Rang :  8 ; Coordonnees : (2,2) ; W = 5.
Rang :  4 ; Coordonnees : (1,1) ; W = 4.
Rang :  5 ; Coordonnees : (1,2) ; W = 5.
Rang :  6 ; Coordonnees : (2,0) ; W = 3.
Rang : 10 ; Coordonnees : (3,1) ; W = 4.
Rang : 11 ; Coordonnees : (3,2) ; W = 5.
Rang :  9 ; Coordonnees : (3,0) ; W = 3.
Rang :  2 ; Coordonnees : (0,2) ; W = 5.
Rang :  7 ; Coordonnees : (2,1) ; W = 4.
```

1 Introduction
2 Environnement
3 Communications point à point
4 Communications collectives
5 Modèles de communication
6 Types de données dérivés
7 Communicateurs
8 MPI-IO
8.1 Introduction . . . 172
8.2 Ouverture et fermeture d'un fichier . . . 177
8.3 Lectures/écritures : généralités . . . 180
8.4 Lectures/écritures individuelles . . . 184
8.5 Lectures/écritures collectives . . . 197
8.6 Positionnement explicite des pointeurs dans un fichier . . . 208
8.7 Vues sur les fichiers . . . 211
8.8 Lectures/écritures non bloquantes . . . 225
8.9 Conseils . . . 234
8.10 Definitions . . . 235
9 Conclusion
10 Annexes
11 Index

8 – MPI-IO
8.1 – Introduction

Optimisation des entrées-sorties

- Très logiquement, les applications qui font des calculs volumineux manipulent également des quantités importantes de données externes, et génèrent donc un nombre conséquent d'entrées-sorties.
- Le traitement efficace de celles-ci influe donc parfois très fortement sur les performances globales des applications.
- L'optimisation des entrées-sorties de codes parallèles se fait par la combinaison :
  - de leur parallélisation, pour éviter de créer un goulet d'étranglement en raison de leur sérialisation ;
  - de techniques mises en œuvre explicitement au niveau de la programmation (lectures / écritures non-bloquantes) ;
  - d'opérations spécifiques prises en charge par le système d'exploitation (regroupement des requêtes, gestion des tampons d'entrées-sorties, etc.).
- L'utilisation d'une bibliothèque facilite l'optimisation des entrées-sorties.

L'interface MPI-IO

- La norme MPI-2 définie un ensemble de fonctions permettant de réaliser des entrées-sorties parallèles.
- L'interface est calquée sur celle utilisée pour l'échange de messages MPI. Par exemple, les opérations collectives et non-bloquantes sur les fichiers sont gérées de façon similaire à ce que propose MPI pour les messages entre processus. La définition des données accédées suivant les processus se fait par l'utilisation de types de données (de base ou bien dérivés).
- Bien sûr, de nombreux éléments (descripteurs de fichiers, attributs ...) rappellent les interfaces d'entrées-sorties natives des langages de programmation.

Enjeux

- L'interface de haut niveau a pour but d'offrir simplicité, expressivité et souplesse à l'utilisateur, tout en autorisant des implémentations performantes prenant en compte les spécificités matérielles et logicielles des machines cibles.
- L'utilisateur peut exprimer lui-même un certain nombre d'optimisations (opérations collectives et non-bloquantes).
- De nombreuses optimisations essentielles sont implémentées de façon transparente dans les bibliothèques.

Exemple d'optimisation séquentielle implémentée par les bibliothèques

- Pour obtenir de bonnes performances, il est préférable de limiter le nombre de requêtes (latence) et de lire de large bloc de données.
- Lorsqu'un seul processus accède à de nombreux petits blocs discontinus, il est possible de regrouper les requêtes pour plus de performances.
- Une bibliothèque MPI-IO peut implémenter cette optimisation de manière transparente.

Requêtes sur de petits blocs non contigus d'un fichier

Lecture d'un grand bloc contigu et transfert dans une zone mémoire tampon

Copies mémoire des éléments requis dans les variables du programme

FIGURE 40 – Mécanisme de *passoire* (*data sieving*) dans le cas d'accès nombreux, par un seul processus, à de petits blocs discontinus

Exemple d'optimisation parallèle

Lorsqu'un ensemble de processus accède à des blocs discontinus (cas des tableaux distribués, par exemple), la bibliothèque d'I/O peut optimiser l'opération en maximisant l'accès contiguë aux données et en utilisant des communications collectives de redistribution.

FIGURE 41 – Lecture en deux phases, par un ensemble de processus

8 – MPI-IO
8.2 – Ouverture et fermeture d'un fichier

Ouverture et fermeture d'un fichier

- L'ouverture et la fermeture d'un fichier sont des opérations *collectives*.
- Tous les processus du communicateur au sein duquel un fichier est ouvert participeront aux opérations collectives ultérieures d'accès aux données.
- L'ouverture d'un fichier renvoie un descripteur. C'est un objet opaque qui est ensuite utilisé comme référence dans toutes les opérations portant sur le fichier.
- A l'ouverture, les attributs décrivent les droits d'accès, le mode d'ouverture, la destruction éventuelle à la fermeture, etc. Les attributs doivent être précisés en utilisant des constantes prédéfinies et peuvent être combinés entre eux.
- Nous ne décrivons ici que les sous-programmes d'ouverture et de fermeture mais d'autres sous-programme de gestion de fichiers sont disponibles (obtention des caractéristiques, suppression, pré-allocation, etc.). Par exemple, le sous-programme `MPI_FILE_GET_INFO()` permet d'obtenir des informations sur un fichier ouvert (les informations disponibles varient d'une implémentation à l'autre).

```
1  program open01
2
3    use mpi
4    implicit none
5
6    integer :: descripteur,code
7
8    call MPI_INIT(code)
9
10   call MPI_FILE_OPEN(MPI_COMM_WORLD,"fichier.txt", &
11                      MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,descripteur,code)
12
13   call MPI_FILE_CLOSE(descripteur,code)
14   call MPI_FINALIZE(code)
15
16 end program open01
```

```
> ls -l fichier.txt

-rw-------   1 nom       grp   0 Feb 08 12:13 fichier.txt
```

Table 4 – Attributs pouvant être positionnés lors de l'ouverture des fichiers

Attribut	Signification
MPI_MODE_RDONLY	seulement en lecture
MPI_MODE_RDWR	en lecture et écriture
MPI_MODE_WRONLY	seulement en écriture
MPI_MODE_CREATE	création du fichier s'il n'existe pas
MPI_MODE_EXCL	erreur si le fichier existe
MPI_MODE_UNIQUE_OPEN	erreur si le fichier est déjà ouvert par une autre application
MPI_MODE_SEQUENTIAL	accès séquentiel
MPI_MODE_APPEND	pointeurs en fin de fichier (mode ajout)
MPI_MODE_DELETE_ON_CLOSE	destruction après la fermeture

8 – MPI-IO

8.3 – Lectures/écritures : généralités

Généralités

- Les transferts de données entre fichiers et zones mémoire des processus se font via des appels explicites à des sous-programmes de lecture et d'écriture.
- On distingue trois propriétés des accès aux fichiers :
  - le positionnement, qui peut être explicite (en spécifiant un déplacement par rapport au début du fichier) ou implicite, via des pointeurs gérés par le système (ces pointeurs peuvent être de deux types : soit individuels à chaque processus, soit partagés par tous les processus) ;
  - la synchronisation, les accès pouvant être de type bloquants ou non bloquants ;
  - le regroupement, les accès pouvant être collectifs (c'est-à-dire effectués par tous les processus du communicateur au sein duquel le fichier a été ouvert) ou propres seulement à un ou plusieurs processus.
- Il est possible de mélanger les types d'accès effectués à un même fichier au sein d'une application.

Table 5 – Résumé des types d'accès possibles

Positionnement	**Synchronisation**	**Regroupement**	
		individuel	*collectif*
adresses explicites	bloquantes	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	non bloquantes	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END

suite page suivante

Positionnement	**Synchronisation**	**Regroupement**	
		individuel	*collectif*
pointeurs implicites individuels	bloquantes	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	non bloquantes	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
pointeurs implicites partagés	bloquantes	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	non bloquantes	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Le mécanisme de vue

- Par défaut, les fichiers sont lus comme une simple suite d'octets mais MPI-IO dispose d'un mécanisme permettant une abstraction de plus haut niveau du contenu des fichiers : il est possible de décrire des structure de données complexes et de s'en servir comme gabarit lors de l'accès aux fichiers.
- Pour l'instant, il faut seulement savoir qu'un type élémentaire de données sert d'unité de base à ces constructions et que par défaut, le type élémentaire est l'octet.
- Ce mécanisme de vue sera décrit en détails plus tard.

8 – MPI-IO

8.4 – Lectures/écritures individuelles

8.4.1 – Via des déplacements explicites

Déplacements explicites

- Le déplacement est explicite lorsque la position à partir de laquelle on travaille sur le fichier est définie explicitement lors de l'opération de lecture ou d'écriture.
- La position dans un fichier s'exprime toujours comme un multiple du type élémentaire de la vue courante. Par défaut, la position s'exprime donc en octets.
- La taille de la zone de lecture/ecriture est exprimée en nombre d'occurences d'un type de données (ex : `MPI_INTEGER`). La taille du type doit être un multiple du type élémentaire.

```
1  program write_at
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: i,rang,descripteur,code,nb_octets_entier
7    integer(kind=MPI_OFFSET_KIND)          :: position_fichier
8    integer, dimension(nb_valeurs)        :: valeurs
9    integer, dimension(MPI_STATUS_SIZE)  :: statut
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   valeurs(:)= (/(i+rang*100,i=1,nb_valeurs)/)
15   print *, "Écriture processus",rang,":",valeurs(:)
16
17   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
18                      MPI_INFO_NULL,descripteur,code)
19
20   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
21   position_fichier=rang*nb_valeurs*nb_octets_entier
22
23   call MPI_FILE_WRITE_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
24                          statut,code)
25
26   call MPI_FILE_CLOSE(descripteur,code)
27   call MPI_FINALIZE(code)
28 end program write_at
```

FIGURE 42 – Exemple d'utilisation de `MPI_FILE_WRITE_AT()`

```
> mpiexec -n 2 write_at

Écriture processus 0 :    1,    2,    3,    4,    5,    6,    7,    8,    9,   10
Écriture processus 1 :  101,  102,  103,  104,  105,  106,  107,  108,  109,  110
```

```
1  program read_at
2
3    use mpi
4    implicit none
5
6    integer, parameter                    :: nb_valeurs=10
7    integer                               :: rang,descripteur,code,nb_octets_entier
8    integer(kind=MPI_OFFSET_KIND)          :: position_fichier
9    integer, dimension(nb_valeurs)        :: valeurs
10   integer, dimension(MPI_STATUS_SIZE)  :: statut
11
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                      descripteur,code)
17
18   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
19
20   position_fichier=rang*nb_valeurs*nb_octets_entier
21   call MPI_FILE_READ_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
22                         statut,code)
23   print *, "Lecture processus",rang,":",valeurs(:)
24
25   call MPI_FILE_CLOSE(descripteur,code)
26   call MPI_FINALIZE(code)
27
28 end program read_at
```

FIGURE 43 – Exemple d'utilisation de `MPI_FILE_READ_AT()`

```
> mpiexec -n 2 read_at

Lecture processus 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

8 – MPI-IO

8.4 – Lectures/écritures individuelles

8.4.2 – Via des déplacements implicites individuels

Déplacements implicites individuels

- Un pointeur individuel est géré par le système, et ceci par fichier et par processus.
- Pour un processus donné, deux accès successifs au même fichier permettent donc d'accéder automatiquement aux éléments consécutifs de celui-ci.
- Dans tous ces sous-programmes, les pointeurs partagés ne sont jamais accédés ou modifiés explicitement.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.

```
1  program read01
2
3    use mpi
4    implicit none
5
6    integer, parameter                    :: nb_valeurs=10
7    integer                               :: rang,descripteur,code
8    integer, dimension(nb_valeurs)        :: valeurs
9    integer, dimension(MPI_STATUS_SIZE) :: statut
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur,code)
16
17   call MPI_FILE_READ(descripteur,valeurs,6,MPI_INTEGER,statut,code)
18   call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut,code)
19
20   print *, "Lecture processus",rang,":",valeurs(:)
21
22   call MPI_FILE_CLOSE(descripteur,code)
23   call MPI_FINALIZE(code)
24
25 end program read01
```

FIGURE 44 – Exemple 1 d'utilisation de `MPI_FILE_READ()`

```
> mpiexec -n 2 read01

Lecture processus 1 :  1,  2,  3,  4,  5,  6,  7,  8,  9,  10
Lecture processus 0 :  1,  2,  3,  4,  5,  6,  7,  8,  9,  10
```

```
1  program read02
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: rang,descripteur,code
7    integer, dimension(nb_valeurs)        :: valeurs=0
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur,code)
15
16   if (rang == 0) then
17      call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut,code)
18   else
19      call MPI_FILE_READ(descripteur,valeurs,8,MPI_INTEGER,statut,code)
20      call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut,code)
21   end if
22
23   print *, "Lecture processus",rang,":",valeurs(1:8)
24
25   call MPI_FILE_CLOSE(descripteur,code)
26   call MPI_FINALIZE(code)
27 end program read02
```

FIGURE 45 – Exemple 2 d'utilisation de `MPI_FILE_READ()`

```
> mpiexec -n 2 read02

Lecture processus 0 :  1,   2,   3,   4,   5,   0,   0,   0
Lecture processus 1 :  9,  10, 101, 102, 103,   6,   7,   8
```

8 – MPI-IO

8.4 – Lectures/écritures individuelles

8.4.3 – Via des déplacements implicites partagés

Déplacements implicites partagés

- Il existe un et un seul pointeur partagé par fichier, commun à tous les processus du communicateur dans lequel le fichier a été ouvert.
- Tous les processus qui font une opération d'entrée-sortie utilisant le pointeur partagé doivent employer la même vue du fichier.
- Si on utilise les variantes non collectives des sous-programmes, l'ordre de lecture n'est pas déterministe. Si le traitement doit être déterministe, il faut explicitement gérer l'ordonnancement des processus ou utiliser les variantes collectives.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.
- Dans tous ces sous-programmes, les pointeurs individuels ne sont jamais accédés ou modifiés.

```
1  program read_shared01
2
3    use mpi
4    implicit none
5
6    integer                              :: rang,descripteur,code
7    integer, parameter                   :: nb_valeurs=10
8    integer, dimension(nb_valeurs)       :: valeurs
9    integer, dimension(MPI_STATUS_SIZE) :: statut
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur,code)
16
17   call MPI_FILE_READ_SHARED(descripteur,valeurs,4,MPI_INTEGER,statut,code)
18   call MPI_FILE_READ_SHARED(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
19
20   print *, "Lecture processus",rang,":",valeurs(:)
21
22   call MPI_FILE_CLOSE(descripteur,code)
23   call MPI_FINALIZE(code)
24
25 end program read_shared01
```

FIGURE 46 – Exemple 2 d'utilisation de `MPI_FILE_READ_SHARED()`

```
> mpiexec -n 2 read_shared01

Lecture processus  1 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus  0 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

8 – MPI-IO
8.5 – Lectures/écritures collectives

Lectures/écritures collectives

- Tous les processus du communicateur au sein duquel un fichier est ouvert participent aux opérations collectives d'accès aux données.
- Les opérations collectives sont généralement plus performantes que les opérations individuelles, parce qu'elles autorisent davantage de techniques d'optimisation mises en œuvre automatiquement ;
- Les accès sont effectués dans l'ordre des rangs des processus. Le traitement est donc dans ce cas déterministe.

8 – MPI-IO

8.5 – Lectures/écritures collectives

8.5.1 – Via des déplacements explicites

```
program read_at_all
  use mpi
  implicit none

  integer, parameter                    :: nb_valeurs=10
  integer                               :: rang,descripteur,code,nb_octets_entier
  integer(kind=MPI_OFFSET_KIND)          :: position_fichier
  integer, dimension(nb_valeurs)        :: valeurs
  integer, dimension(MPI_STATUS_SIZE)  :: statut

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     descripteur,code)

  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
  position_fichier=rang*nb_valeurs*nb_octets_entier
  call MPI_FILE_READ_AT_ALL(descripteur,position_fichier,valeurs,nb_valeurs, &
                            MPI_INTEGER,statut,code)
  print *, "Lecture processus",rang,":",valeurs(:)

  call MPI_FILE_CLOSE(descripteur,code)
  call MPI_FINALIZE(code)
end program read_at_all
```

FIGURE 47 – Exemple d'utilisation de `MPI_FILE_READ_AT_ALL()`

```
> mpiexec -n 2 read_at_all

Lecture processus 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

8 – MPI-IO

8.5 – Lectures/écritures collectives

8.5.2 – Via des déplacements implicites individuels

```
1  program read_all01
2    use mpi
3    implicit none
4
5    integer                              :: rang,descripteur,code
6    integer, parameter                   :: nb_valeurs=10
7    integer, dimension(nb_valeurs)       :: valeurs
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur,code)
15
16   call MPI_FILE_READ_ALL(descripteur,valeurs,4,MPI_INTEGER,statut,code)
17   call MPI_FILE_READ_ALL(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
18
19   print *, "Lecture processus ",rang,":",valeurs(:)
20
21   call MPI_FILE_CLOSE(descripteur,code)
22   call MPI_FINALIZE(code)
23 end program read_all01
```

FIGURE 48 – Exemple 1 d'utilisation de `MPI_FILE_READ_ALL()`

```
> mpiexec -n 2 read_all01

Lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Lecture processus 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
1  program read_all02
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: rang,descripteur,indice1,indice2,code
7    integer, dimension(nb_valeurs)        :: valeurs=0
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
13                      descripteur,code)
14
15   if (rang == 0) then
16      indice1=3
17      indice2=6
18   else
19      indice1=5
20      indice2=9
21   end if
22
23   call MPI_FILE_READ_ALL(descripteur,valeurs(indice1),indice2-indice1+1, &
24                          MPI_INTEGER,statut,code)
25   print *, "Lecture processus",rang,":",valeurs(:)
26
27   call MPI_FILE_CLOSE(descripteur,code)
28   call MPI_FINALIZE(code)
29 end program read_all02
```

FIGURE 49 – Exemple 2 d'utilisation de `MPI_FILE_READ_ALL()`

```
> mpiexec -n 2 read_all02

Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

```
1  program read_all03
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: rang,descripteur,code
7    integer, dimension(nb_valeurs)        :: valeurs=0
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur,code)
15
16   if (rang == 0) then
17      call MPI_FILE_READ_ALL(descripteur,valeurs(3),4,MPI_INTEGER,statut,code)
18   else
19      call MPI_FILE_READ_ALL(descripteur,valeurs(5),5,MPI_INTEGER,statut,code)
20   end if
21
22   print *, "Lecture processus",rang,":",valeurs(:)
23
24   call MPI_FILE_CLOSE(descripteur,code)
25   call MPI_FINALIZE(code)
26 end program read_all03
```

FIGURE 50 – Exemple 3 d'utilisation de `MPI_FILE_READ_ALL()`

```
> mpiexec -n 2 read_all03

Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

8 – MPI-IO

8.5 – Lectures/écritures collectives

8.5.3 – Via des déplacements implicites partagés

```
1  program read_ordered
2    use mpi
3    implicit none
4
5    integer                              :: rang,descripteur,code
6    integer, parameter                   :: nb_valeurs=10
7    integer, dimension(nb_valeurs)       :: valeurs
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur,code)
15
16   call MPI_FILE_READ_ORDERED(descripteur,valeurs,4,MPI_INTEGER,statut,code)
17   call MPI_FILE_READ_ORDERED(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
18
19   print *, "Lecture processus",rang,":",valeurs(:)
20
21   call MPI_FILE_CLOSE(descripteur,code)
22   call MPI_FINALIZE(code)
23 end program read_ordered
```

FIGURE 51 – Exemple d'utilisation de `MPI_FILE_ORDERED()`

```
> mpiexec -n 2 read_ordered

Lecture processus 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Lecture processus 0 : 1, 2, 3, 4,   9,  10, 101, 102, 103, 104
```

8 – MPI-IO

8.6 – Positionnement explicite des pointeurs dans un fichier

Positionnement explicite des pointeurs dans un fichier

- Les sous-programmes `MPI_FILE_GET_POSITION()` et `MPI_FILE_GET_POSITION_SHARED()` permettent de connaître respectivement la valeur courante des pointeurs individuels et celle du pointeur partagé.
- Il est possible de positionner explicitement les pointeurs individuels à l'aide du sous-programme `MPI_FILE_SEEK()`, et de même le pointeur partagé avec le sous-programme `MPI_FILE_SEEK_SHARED()`.
- Il y a trois modes possibles pour modifier la valeur d'un pointeur :
  - `MPI_SEEK_SET` permet de définir un déplacement absolu ;
  - `MPI_SEEK_CUR` permet un déplacement relativement à la position courante ;
  - `MPI_SEEK_END` positionne le pointeur à la fin du fichier, à laquelle un déplacement éventuel est ajouté.
- Avec `MPI_SEEK_CUR`, on peut spécifier une valeur négative, ce qui permet de revenir en arrière dans le fichier.

```
program seek
  use mpi
  implicit none
  integer, parameter                    :: nb_valeurs=10
  integer                               :: rang,descripteur,nb_octets_entier,code
  integer(kind=MPI_OFFSET_KIND)         :: position_fichier
  integer, dimension(nb_valeurs)        :: valeurs
  integer, dimension(MPI_STATUS_SIZE)   :: statut

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     descripteur,code)

  call MPI_FILE_READ(descripteur,valeurs,3,MPI_INTEGER,statut,code)
  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
  position_fichier=8*nb_octets_entier
  call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_CUR,code)
  call MPI_FILE_READ(descripteur,valeurs(4),3,MPI_INTEGER,statut,code)
  position_fichier=4*nb_octets_entier
  call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_SET,code)
  call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut,code)

  print *, "Lecture processus",rang,":",valeurs(:)

  call MPI_FILE_CLOSE(descripteur,code)
  call MPI_FINALIZE(code)
end program seek
```

FIGURE 52 – Exemple d'utilisation de `MPI_FILE_SEEK()`

```
> mpiexec -n 2 seek

Lecture processus 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Lecture processus 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```

8 – MPI-IO
8.7 – Vues sur les fichiers
8.7.1 – Définition des vues

Définition des vues

- C'est un mécanisme permettant de décrire un schéma d'accès aux fichiers.
- Une vue est définie par trois variables : un déplacement initial, un type élémentaire de données et un motif.
- L'accès aux fichiers s'effectue par répétition du motif, une fois le positionnement initial effectué.

FIGURE 53 – Type élémentaire de donnée et motif

Définition des vues

- Les vues sont construites à l'aide de types dérivés MPI.
- Il est possible de définir des trous dans une vue, de façon à ne pas tenir compte de certaines parties des données.
- La vue par défaut consiste en une simple suite d'octets (déplacement initial nul, *type_ élém* et motif égaux à `MPI_BYTE`).

Vues multiples

- Un processus donné peut définir et utiliser successivement plusieurs vues d'un même fichier.
- Les processus peuvent avoir des vues différentes du fichier, de façon à accéder à des parties complémentaires de celui-ci.

Figure 54 – Exemple de définition de motifs différents selon les processus

Remarques :

- Un pointeur partagé n'est utilisable avec une vue que si tous les processus ont la même vue.
- Si le fichier est ouvert en écriture, les zones décrites par les types élémentaires et les motifs ne peuvent se recouvrir, même partiellement.

Changement de la vue sur un fichier : MPI_FILE_SET_VIEW()

```
integer :: descripteur
integer(kind=MPI_OFFSET_KIND) :: deplacement_initial
integer :: type_elem
integer :: motif
character(len=*) :: mode
integer :: info
integer :: code

call MPI_FILE_SET_VIEW(descripteur,
                       deplacement_initial,type_elem,motif,
                       mode,info,code)
```

- C'est une **opération collective** à l'ensemble des processus impliqués dans l'accès au fichier. Chaque processus peut définir **un déplacement initial et un motif différent**. L'étendu du type élémentaire doit être identique.
- Les pointeurs individuels et le pointeur partagé **sont réinitialisés au début de la vue**.

Notes :

- Les types dérivés utilisés doivent avoir été validés au préalable à l'aide du sous-programme MPI_TYPE_COMMIT().
- Il y a trois représentations possibles des données (mode) : "native", "internal" ou "external32".

Exemple 1 : lecture d'un fichier par blocs de deux éléments

```
> mpiexec -n 2 read_view01

Lecture processus 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Lecture processus 0 : 1, 2, 5, 6,   9,  10, 103, 104, 107, 108
```

Exemple 1 (suite) : création des vues sur le fichier

dép_initial **0**

type_élém MPI_INTEGER

motif proc. 0

motif proc. 1

```
if (rang == 0) coord=1
if (rang == 1) coord=3

call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
                              MPI_ORDER_FORTRAN,MPI_INTEGER,motif,code)
call MPI_TYPE_COMMIT(motif,code)

! Ne pas omettre de passer par une variable intermédiaire de représentation
! MPI_OFFSET_KIND, pour des raisons de portabilité
deplacement_initial=0

call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
                       "native",MPI_INFO_NULL,code)
```

```
program read_view01
  use mpi
  implicit none
  integer, parameter                    :: nb_valeurs=10
  integer                               :: rang,descripteur,coord,motif,code
  integer(kind=MPI_OFFSET_KIND)         :: deplacement_initial
  integer, dimension(nb_valeurs)        :: valeurs
  integer, dimension(MPI_STATUS_SIZE)   :: statut

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  if (rang == 0) coord=1
  if (rang == 1) coord=3

  call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
                                MPI_ORDER_FORTRAN,MPI_INTEGER,motif,code)
  call MPI_TYPE_COMMIT(motif,code)

  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     descripteur,code)

  deplacement_initial=0
  call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
                         "native",MPI_INFO_NULL,code)
  call MPI_FILE_READ(descripteur,valeurs,nb_valeurs,MPI_INTEGER,statut,code)

  print *, "Lecture processus",rang,":",valeurs(:)

  call MPI_FILE_CLOSE(descripteur,code)
  call MPI_FINALIZE(code)

end program read_view01
```

Exemple 2 : utilisation successive de plusieurs vues

```
program read_view02

  use mpi
  implicit none

  integer, parameter                    :: nb_valeurs=10
  integer                               :: rang,descripteur,code, &
                                           motif_1,motif_2,nb_octets_entier
  integer(kind=MPI_OFFSET_KIND)          :: deplacement_initial
  integer, dimension(nb_valeurs)        :: valeurs
  integer, dimension(MPI_STATUS_SIZE)  :: statut

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

```
15    call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/0/), &
16                                  MPI_ORDER_FORTRAN,MPI_INTEGER,motif_1,code)
17    call MPI_TYPE_COMMIT(motif_1,code)
18
19    call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/1/),(/2/), &
20                                  MPI_ORDER_FORTRAN,MPI_INTEGER,motif_2,code)
21    call MPI_TYPE_COMMIT(motif_2,code)
22
23    call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
24                       descripteur,code)
25
26    ! Lecture en utilisant la premiere vue
27    deplacement_initial=0
28    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif_1, &
29                           "native",MPI_INFO_NULL,code)
30    call MPI_FILE_READ(descripteur,valeurs,4,MPI_INTEGER,statut,code)
31    call MPI_FILE_READ(descripteur,valeurs(5),3,MPI_INTEGER,statut,code)
32
33    ! Lecture en utilisant la seconde vue
34    call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
35    deplacement_initial=2*nb_octets_entier
36    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif_2, &
37                           "native",MPI_INFO_NULL,code)
38    call MPI_FILE_READ(descripteur,valeurs(8),3,MPI_INTEGER,statut,code)
39
40    print *, "Lecture processus",rang,":",valeurs(:)
41
42    call MPI_FILE_CLOSE(descripteur,code)
43    call MPI_FINALIZE(code)
44  end program read_view02
```

```
> mpiexec -n 2 read_view02

Lecture processus 1 : 1, 2, 5, 6, 9, 10, 13, 5, 8, 11
Lecture processus 0 : 1, 2, 5, 6, 9, 10, 13, 5, 8, 11
```

Exemple 3 : gestion des trous dans les types de données

```
program read_view_03_indexed

  use mpi
  implicit none

  integer, parameter                          :: nb_valeurs=9
  integer                                     :: rang,descripteur,nb_octets_entier,code
  integer                                     :: motif_temp,motif
  integer(kind=MPI_OFFSET_KIND)               :: deplacement_initial
  integer(kind=MPI_ADDRESS_KIND)              :: borne_inf,etendue
  integer, dimension(2)                       :: longueurs,deplacements
  integer, dimension(nb_valeurs)              :: valeurs
  integer, dimension(MPI_STATUS_SIZE)         :: statut

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

```

```
18    ! motif_temp: type MPI d'étendu 4*MPI_INTEGER
19    deplacements(1)=0
20    longueurs(1)=2
21    deplacements(2)=3
22    longueurs(2)=1
23    call MPI_TYPE_INDEXED(2,longueurs,deplacements,MPI_INTEGER,motif_temp,code)
24
25    ! motif: type MPI d'étendu 5*MPI_INTEGER
26    call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
27    call MPI_TYPE_GET_EXTENT(motif_temp,borne_inf,etendue,code)
28    etendue = etendue + nb_octets_entier
29    call MPI_TYPE_CREATE_RESIZED(motif_temp,borne_inf,borne_inf+etendue,motif,code)
30    call MPI_TYPE_COMMIT(motif,code)
31
32    call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
33                       descripteur,code)
34
35    deplacement_initial=0
36    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
37                           "native",MPI_INFO_NULL,code)
38
39    call MPI_FILE_READ(descripteur,valeurs,9,MPI_INTEGER,statut,code)
40
41    print *,"Lecture processus",rang,":",valeurs(:)
42
43    call MPI_FILE_CLOSE(descripteur,code)
44    call MPI_FINALIZE(code)
45
46  end program read_view03_indexed
```

```
> mpiexec -n 2 read_view03

Lecture, processus 0 : 1, 2, 4, 6, 7, 9, 11, 12, 14
Lecture, processus 1 : 1, 2, 4, 6, 7, 9, 11, 12, 14
```

Exemple 3 : implémentation alternative, utilisant un type structure.

```
program read_view03_struct

  [...]

  call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/2/),(/0/),MPI_ORDER_FORTRAN, &
       MPI_INTEGER,temp_motif1,code)

  call MPI_TYPE_CREATE_SUBARRAY(1,(/2/),(/1/),(/0/),MPI_ORDER_FORTRAN, &
       MPI_INTEGER,temp_motif2,code)

  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)

  deplacements(1) = 0
  deplacements(2) = 3*nb_octets_entier

  call MPI_TYPE_CREATE_STRUCT(2,(/1,1/),deplacements,&
       (/temp_motif1,temp_motif2/),motif,code)
  call MPI_TYPE_COMMIT(motif,code)

  [...]

end program read_view03_struct
```

8 – MPI-IO
8.8 – Lectures/écritures non bloquantes

Lectures/écritures non bloquantes

- L'intérêt est de faire un recouvrement entre les calculs et les entrées-sorties.
- Les entrées-sorties non bloquantes sont implémentées suivant le modèle utilisé pour les communications non bloquantes entre processus.
- Un accès non-bloquant doit donner lieu ultérieurement à un test explicite de complétude ou à une mise en attente (via `MPI_TEST()`, `MPI_WAIT()`, etc.)

8 – MPI-IO

8.8 – Lectures/écritures non bloquantes

8.8.1 – Via des déplacements explicites

```
1  program iread_at
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: i,nb_iterations=0,rang,nb_octets_entier, &
7                                              descripteur,requete,code
8    integer(kind=MPI_OFFSET_KIND)          :: position_fichier
9    integer, dimension(nb_valeurs)        :: valeurs
10   integer, dimension(MPI_STATUS_SIZE)  :: statut
11   logical                               :: termine
12
13   call MPI_INIT(code)
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

```
1   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
2                      descripteur,code)
3
4   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
5
6   position_fichier=rang*nb_valeurs*nb_octets_entier
7   call MPI_FILE_IREAD_AT(descripteur,position_fichier,valeurs,nb_valeurs, &
8                          MPI_INTEGER,requete,code)
9
10  do while (nb_iterations < 5000)
11     nb_iterations=nb_iterations+1
12     ! Calculs recouvrant le temps demandé par l'opération de lecture
13     ...
14     call MPI_TEST(requete,termine,statut,code)
15     if (termine) exit
16  end do
17  print *,"Après",nb_iterations,"iterations, lecture processus",rang,":",valeurs
18
19  call MPI_FILE_CLOSE(descripteur,code)
20  call MPI_FINALIZE(code)
21
22 end program iread_at
```

FIGURE 55 – Exemple d'utilisation de `MPI_FILE_IREAD_AT()`

```
> mpiexec -n 2 iread_at

Après 1 iterations, lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Après 1 iterations, lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

8 – MPI-IO

8.8 – Lectures/écritures non bloquantes

8.8.2 – Via des déplacements implicites individuels

```
1  program iread
2    use mpi
3    implicit none
4
5    integer, parameter                    :: nb_valeurs=10
6    integer                               :: rang,descripteur,requete,code
7    integer, dimension(nb_valeurs)        :: valeurs
8    integer, dimension(MPI_STATUS_SIZE) :: statut
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur,code)
15
16   call MPI_FILE_IREAD(descripteur,valeurs,nb_valeurs,MPI_INTEGER,requete,code)
17   ! Calcul recouvrant le temps demandé par l'opération de lecture
18   ...
19   call MPI_WAIT(requete,statut,code)
20   print *, "Lecture processus",rang,":",valeurs(:)
21
22   call MPI_FILE_CLOSE(descripteur,code)
23   call MPI_FINALIZE(code)
24 end program iread
```

FIGURE 56 – Exemple 1 d'utilisation de `MPI_FILE_IREAD()`

```
> mpiexec -n 2 iread

Lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Lecture processus 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Lectures/écritures collectives et non bloquantes

- Il est possible d'effectuer des opérations qui soient à la fois collectives et non bloquantes.
- Celles-ci nécessitent un appel à deux sous-programmes distincts, l'un pour déclencher l'opération et l'autre pour la terminer.
- Il ne peut y avoir qu'une seule opération collective non bloquante en cours à la fois par processus.
- Entre les deux phases de l'opération collective non-bloquante, il est possible de faire des opérations non collectives sur le fichier, mais la zone mémoire concernée par l'opération collective ne peut être modifiée.

```
1  program read_ordered_begin_end
2
3    use mpi
4    implicit none
5
6    integer                              :: rang,descripteur,code
7    integer, parameter                   :: nb_valeurs=10
8    integer, dimension(nb_valeurs)       :: valeurs
9    integer, dimension(MPI_STATUS_SIZE) :: statut
10
11   call MPI_INIT(code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur,code)
16
17   call MPI_FILE_READ_ORDERED_BEGIN(descripteur,valeurs,4,MPI_INTEGER,code)
18   print *, "Processus numéro    :",rang
19   call MPI_FILE_READ_ORDERED_END(descripteur,valeurs,statut,code)
20
21   print *, "Lecture processus",rang,":",valeurs(1:4)
22
23   call MPI_FILE_CLOSE(descripteur,code)
24   call MPI_FINALIZE(code)
25
26 end program read_ordered_begin_end
```

FIGURE 57 – Exemple d'utilisation de `MPI_FILE_READ_ORDERED_BEGIN()`

```
> mpiexec -n 2 read_ordered_begin_end

Processus numéro    : 0
Lecture processus 0 : 1, 2, 3, 4
Processus numéro    : 1
Lecture processus 1 : 5, 6, 7, 8
```

8 – MPI-IO
8.9 – Conseils

Conclusion

MPI-IO offre une interface de haut niveau et un ensemble de fonctionnalités très riche. Il est possible de réaliser des opérations complexes et de tirer partie des optimisations implémentées dans la bibliothèque. MPI-IO offre aussi une bonne portabilité.

Conseils

- L'utilisation des sous-programmes à positionnement explicite dans les fichiers ne sont à employer que dans des cas particuliers, l'utilisation implicite de pointeurs individuels ou partagés offrant une interface de plus haut niveau.
- Lorsque les opérations font intervenir l'ensemble (ou un sous-ensemble de processus identifiable par un sous-communicateur MPI), il faut généralement privilégier la forme collective des opérations.
- Exactement comme pour le traitement des messages lorsque ceux-ci représentent une part importante de l'application, le non-bloquant est une voie privilégiée d'optimisation à mettre en œuvre par les programmeurs, mais ceci ne doit être implémenté qu'après qu'on se soit assuré du comportement correct de l'application en mode bloquant.

8 – MPI-IO
8.10 – Definitions

Définitions (fichiers)

fichier (*file*) : un fichier MPI est un ensemble ordonné de données typées. Un fichier est ouvert collectivement par tous les processus d'un communicateur. Toutes les opérations d'entrées-sorties collectives ultérieures se feront dans ce cadre.

descripteur (*file handle*) : le descripteur est un objet opaque créé à l'ouverture et détruit à la fermeture d'un fichier. Toutes les opérations sur un fichier se font en spécifiant comme référence son descripteur.

pointeur (*file pointer*) : ils sont tenus à jour automatiquement par MPI et déterminent des positions à l'intérieur du fichier. Il y en a de deux sortes : les pointeurs individuels qui sont propres à chaque processus ayant ouvert le fichier ; les pointeurs partagés qui sont communs à tous les processus ayant ouvert le fichier.

position (*offset*) : c'est la position dans le fichier, exprimée en nombre de *type_élém*, relativement à la vue courante (les *trous* définis dans la vue ne sont pas comptabilisés pour calculer les positions).

Définitions (vues)

déplacement initial (*displacement*) : c'est une adresse absolue, exprimée en octets, par rapport au début du fichier et à partir de laquelle une vue commence.

type élémentaire de données — type_élém (*etype*) : c'est l'unité utilisée pour calculer les positionnements et pour accéder aux données. Ce peut être n'importe quel type de donnée MPI, prédéfini ou bien créé en tant que type dérivé.

motif (*filetype*) : c'est un masque qui constitue la base du partitionnement d'un fichier entre processus (si le fichier est vu comme un *pavage* à une dimension, le motif est la *tuile élémentaire* qui sert au pavage). C'est ou bien un type élémentaire de données, ou bien un type dérivé MPI construit comme une répétition d'occurrences d'un tel type élémentaire de données (les *trous* — parties de fichiers non accédées — doivent également être un multiple du *type_ élém* utilisé).

vue (*view*) : une vue définit la manière dont les données d'un fichier sont accédées et interprétées. C'est un ensemble ordonné de types élémentaires de données.

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index

9 – Conclusion

Conclusion

- Utiliser les communications point-à-point bloquantes, ceci avant de passer aux communications non-bloquantes. Il faudra alors essayer de faire du recouvrement calcul/communications.
- Utiliser les fonctions d'entrées-sorties bloquantes, ceci avant de passer aux entrées-sorties non-bloquantes. De même, il faudra alors faire du recouvrement calcul/entrées-sorties.
- Écrire les communications comme si les envois étaient synchrones (`MPI_SSEND()`).
- Éviter les barrières de synchronisation (`MPI_BARRIER()`), surtout sur les fonctions collectives qui sont bloquantes.
- La programmation mixte MPI/OpenMP peut apporter des gains d'extensibilité, mais pour que cette approche fonctionne bien, il est évidemment nécessaire d'avoir de bonnes performances OpenMP à l'intérieur de chaque processus MPI. Un cours est dispensé à l'IDRIS (`https://cours.idris.fr/`).

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes
10.1 Communications collectives 241
10.2 Types de données dérivés 243
10.3 Optimisations 260
10.4 Communicateurs 264
10.5 Gestion de processus 280
10.6 RMA 305
10.7 MPI-IO 332

11 Index

10 – Annexes

Il s'agit ici de programmes concernant différentes fonctionnalités de MPI qui sont :

- ☞ moins fréquentes d'utilisation (création de sa propre opération de réduction, type dérivés spécifiques, topologie de type graphe, communications persistantes) ;
- ☞ qui ne sont pas disponibles sur l'ensemble des machines (gestion dynamique de processus, mode client serveur).

10 – Annexes
10.1 – Communications collectives

Dans cet exemple, on se propose de créer sa propre opération de réduction, produit de vecteurs de nombres complexes.

```
program ma_reduction
  use mpi
  implicit none
  integer                 :: rang,code,i,mon_operation
  integer, parameter      :: n=4
  complex, dimension(n) :: a,resultat
  external mon_produit

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
  ! Initialisation du vecteur A sur chaque processus
  a(:) = (/ (cmplx(rang+i,rang+i+1),i=1,n) /)
  ! Création de l'opération commutative mon_operation
  call MPI_OP_CREATE(mon_produit,.true.,mon_operation,code)
  ! Collecte sur le processus 0 du produit global
  call MPI_REDUCE(a,resultat,n,MPI_COMPLEX,mon_operation,0,MPI_COMM_WORLD,code)

  ! Affichage du résultat
  if (rang == 0) then
     print *,'Valeur du produit',resultat
  end if
  call MPI_FINALIZE(code)
end program ma_reduction
```

```
24 ! Définition du produit terme à terme de deux vecteurs de nombres complexes
25
26 integer function mon_produit(vecteur1,vecteur2,longueur,type_donnee) result(inutilise)
27   implicit none
28
29   complex,dimension(longueur) :: vecteur1,vecteur2
30   integer                     :: longueur,type_donnee,i
31
32   do i=1,longueur
33      vecteur2(i) = cmplx(real(vecteur1(i))*real(vecteur2(i)) - &
34                          aimag(vecteur1(i))*aimag(vecteur2(i)), &
35                        real(vecteur1(i))*aimag(vecteur2(i)) + &
36                          aimag(vecteur1(i))*real(vecteur2(i)))
37   end do
38
39   inutilise=0
40
41 end function mon_produit
```

```
> mpiexec -n 5 ma_reduction

Valeur du produit (155.,-2010.), (-1390.,-8195.), (-7215.,-23420.), (-22000.,-54765.)
```

10 – Annexes
10.2 – Types de données dérivés
10.2.1 – Distribution d'un tableau sur plusieurs processus

☞ Le sous-programme MPI_TYPE_CREATE_DARRAY() permet de générer un tableau sur un ensemble de processus suivant une distribution par blocs ou cyclique.

```
integer,intent(in)                      :: nb_procs,rang,nb_dims
integer,dimension(nb_dims),intent(in) :: profil_tab,mode_distribution
integer,dimension(nb_dims),intent(in) :: profil_sous_tab,distribution_procs
integer,intent(in)                      :: ordre,ancien_type
integer,intent(out)                     :: nouveau_type,code
call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution,
                            profil_sous_tab,distribution_procs,ordre,ancien_type,
                            nouveau_type,code)
```

- nb_dims : rang du tableau
- nb_procs : nombre total de processus
- rang : rang de chaque processus
- profil_tab : profil du tableau à distribuer
- mode_distribution : mode de distribution dans chaque dimension du tableau, soit :
  1. `MPI_DISTRIBUTE_BLOCK` indique une distribution par blocs
  2. `MPI_DISTRIBUTE_CYCLIC` indique une distribution cyclique
  3. `MPI_DISTRIBUTE_NONE` indique qu'il n'y a pas de distribution
- profil_sous_tab : profil d'un bloc
- distribution_procs : nombre de processus dans chaque dimension

Quelques remarques :

- ☞ l'ordre des processus est le même que pour les topologies ;
- ☞ pour que l'appel au sous-programme soit correct, on doit avoir `nb_procs` = $\prod_{i=1}^{nb_dims}$ `distribution_procs(i)` ;
- ☞ lorsqu'une dimension `i` est distribuée par blocs, via le paramètre `MPI_DISTRIBUTE_BLOCK`, la règle suivante doit être respectée `profil_sous_tab(i) * distribution_procs(i)` $\geq$ `profil_tab(i)` ;
- ☞ lorsqu'une dimension `i` n'est pas distribuée, via le paramètre `MPI_DISTRIBUTE_NONE`, le nombre de processus choisi dans cette dimension doit valoir `1` (`distribution_procs(i) = 1`) et le profil du bloc dans cette dimension (`profil_sous_tab(i)`) est ignoré.

Distribution par blocs d'un tableau suivant 4 processus

- ☞ `nb_procs = 4, distribution_procs(:) = (/ 2,2 /)`
- ☞ `profil_tab(:) = (/ 4,6 /), profil_sous_tab(:) = (/ 2,3 /)`
- ☞ `mode_distribution(:)=(/MPI_DISTRIBUTE_BLOCK,MPI_DISTRIBUTE_BLOCK/)`

FIGURE 58 – Définition du type dérivé sur chaque processus pour une distribution par blocs

```
1  program darray_bloc
2    use mpi
3    implicit none
4
5    integer,parameter                          :: nb_lignes=4,nb_colonnes=6, &
6                                                  nb_dims=2,etiquette1=1000,etiquette2=1001
7    integer                                    :: nb_procs,code,rang,i,type_bloc
8    integer,dimension(nb_lignes,nb_colonnes)   :: tab
9    integer,dimension(nb_dims)                 :: profil_tab,mode_distribution, &
10                                                  profil_sous_tab,distribution_procs
11   integer,dimension(MPI_STATUS_SIZE)         :: statut
12
13   call MPI_INIT(code)
14   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
16
17   ! Initialisation du tableau tab sur chaque processus
18   tab(:,:)=reshape((/(i*(rang+1),i=1,nb_lignes*nb_colonnes)/),(/nb_lignes,nb_colonnes/))
19
20   ! Profil du tableau tab
21   profil_tab(:) = shape(tab)
22
23   ! Mode de distribution
24   mode_distribution(:) = (/ MPI_DISTRIBUTE_BLOCK,MPI_DISTRIBUTE_BLOCK /)
25
26   ! Profil d'un bloc
27   profil_sous_tab(:) = (/ 2,3 /)
```

```
28    ! Nombre de processus dans chaque dimension
29    distribution_procs(:) = (/ 2,2 /)
30    ! Création du type dérivé type_bloc
31    call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution, &
32                                profil_sous_tab, distribution_procs,MPI_ORDER_FORTRAN, &
33                                MPI_INTEGER,type_bloc,code)
34    call MPI_TYPE_COMMIT(type_bloc,code)
35    select case(rang)
36      case(0)
37        ! Le processus 0 envoie son tableau tab au processus 1
38        call MPI_SEND(tab,1,type_bloc,1,etiquette1,MPI_COMM_WORLD,code)
39      case(1)
40        ! Le processus 1 reçoit son tableau tab du processus 0
41        call MPI_RECV(tab,1,type_bloc,0,etiquette1,MPI_COMM_WORLD,statut,code)
42      case(2)
43        ! Le processus 2 envoie son tableau tab au processus 3
44        call MPI_SEND(tab,1,type_bloc,3,etiquette2,MPI_COMM_WORLD,code)
45      case(3)
46        ! Le processus 3 reçoit son tableau tab du processus 2
47        call MPI_RECV(tab,1,type_bloc,2,etiquette2,MPI_COMM_WORLD,statut,code)
48    end select
49    ! Affichage du tableau tab sur chaque processus
50    ......................................
51    call MPI_TYPE_FREE(type_bloc,code)
52    call MPI_FINALIZE(code)
53  end program darray_bloc
```

```
> mpiexec -n 4 darray_bloc

Tableau tab obtenu sur le processus 0
  1,   5,   9,  13,  17,  21
  2,   6,  10,  14,  18,  22
  3,   7,  11,  15,  19,  23
  4,   8,  12,  16,  20,  24

Tableau tab obtenu sur le processus 1
  2,  10,  18,   1,   5,   9
  4,  12,  20,   2,   6,  10
  6,  14,  22,  30,  38,  46
  8,  16,  24,  32,  40,  48

Tableau tab obtenu sur le processus 2
  3,  15,  27,  39,  51,  63
  6,  18,  30,  42,  54,  66
  9,  21,  33,  45,  57,  69
 12,  24,  36,  48,  60,  72

Tableau tab obtenu sur le processus 3
  4,  20,  36,  52,  68,  84
  8,  24,  40,  56,  72,  88
 12,  28,  44,   9,  21,  33
 16,  32,  48,  12,  24,  36
```

Distribution cyclique d'un tableau suivant 4 processus

- `nb_procs = 4`, `distribution_procs(:) = (/ 2,2 /)`
- `profil_tab(:) = (/ 4,6 /)`, `profil_sous_tab(:) = (/ 1,2 /)`
- `mode_distribution(:)=(/MPI_DISTRIBUTE_CYCLIC,MPI_DISTRIBUTE_CYCLIC/)`

FIGURE 59 – Définition du type dérivé sur chaque processus pour une distribution cyclique

```
1  program darray_cyclique
2    use mpi
3    implicit none
4
5    integer,parameter                          :: nb_lignes=4,nb_colonnes=6, &
6                                                  nb_dims=2,etiquette1=1000,etiquette2=1001
7    integer                                    :: nb_procs,code,rang,i,type_cyclique
8    integer,dimension(nb_lignes,nb_colonnes)   :: tab
9    integer,dimension(nb_dims)                 :: profil_tab,mode_distribution, &
10                                                  profil_sous_tab,distribution_procs
11   integer,dimension(MPI_STATUS_SIZE)         :: statut
12
13   call MPI_INIT(code)
14   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
16
17   ! Initialisation du tableau tab sur chaque processus
18   tab(:,:)=reshape((/(i*(rang+1),i=1,nb_lignes*nb_colonnes)/),(/nb_lignes,nb_colonnes/))
19
20   ! Profil du tableau tab
21   profil_tab(:) = shape(tab)
22
23   ! Mode de distribution
24   mode_distribution(:) = (/ MPI_DISTRIBUTE_CYCLIC,MPI_DISTRIBUTE_CYCLIC /)
25
26   ! Profil d'un bloc
27   profil_sous_tab(:) = (/ 1,2 /)
```

```
28    ! Nombre de processus dans chaque dimension
29    distribution_procs(:) = (/ 2,2 /)
30
31    ! Création du type dérivé type_cyclique
32    call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution, &
33                                profil_sous_tab, distribution_procs,MPI_ORDER_FORTRAN, &
34                                MPI_INTEGER,type_cyclique,code)
35    call MPI_TYPE_COMMIT(type_cyclique,code)
36
37    select case(rang)
38      case(0)
39        ! Le processus 0 envoie son tableau tab au processus 2
40        call MPI_SEND(tab,1,type_cyclique,2,etiquette1,MPI_COMM_WORLD,code)
41      case(2)
42        ! Le processus 2 reçoit son tableau tab du processus 0
43        call MPI_RECV(tab,1,type_cyclique,0,etiquette1,MPI_COMM_WORLD,statut,code)
44      case(1)
45        ! Le processus 1 envoie son tableau tab au processus 3
46        call MPI_SEND(tab,1,type_cyclique,3,etiquette2,MPI_COMM_WORLD,code)
47      case(3)
48        ! Le processus 3 reçoit son tableau tab du processus 1
49        call MPI_RECV(tab,1,type_cyclique,1,etiquette2,MPI_COMM_WORLD,statut,code)
50     end select
51
52    ! Affichage du tableau tab sur chaque processus
53    ......................................
54
55    call MPI_TYPE_FREE(type_cyclique,code)
56
57    call MPI_FINALIZE(code)
58  end program darray_cyclique
```

```
> mpiexec -n 4 darray_cyclique

Tableau tab obtenu sur le processus 0
  1,   5,   9,  13,  17,  21
  2,   6,  10,  14,  18,  22
  3,   7,  11,  15,  19,  23
  4,   8,  12,  16,  20,  24

Tableau tab obtenu sur le processus 1
  2,  10,  18,  26,  34,  42
  4,  12,  20,  28,  36,  44
  6,  14,  22,  30,  38,  46
  8,  16,  24,  32,  40,  48

Tableau tab obtenu sur le processus 2
  3,  15,  27,  39,  51,  63
  1,   5,  30,  42,  17,  21
  9,  21,  33,  45,  57,  69
  3,   7,  36,  48,  19,  23

Tableau tab obtenu sur le processus 3
  4,  20,  36,  52,  68,  84
  8,  24,  18,  26,  72,  88
 12,  28,  44,  60,  76,  92
 16,  32,  22,  30,  80,  96
```

Autres exemples

- ☞ `nb_procs = 4`, `distribution_procs(:) = (/ 2,2 /)`
- ☞ `profil_tab(:) = (/ 4,3 /)`, `profil_sous_tab(:) = (/ 2,2 /)`
- ☞ `mode_distribution(:)=(/MPI_DISTRIBUTE_BLOCK,MPI_DISTRIBUTE_BLOCK/)`

FIGURE 60 – Définition du type dérivé sur chaque processus

- ☞ `nb_procs = 3, distribution_procs(:) = (/ 3,1 /)`
- ☞ `profil_tab(:) = (/ 5,5 /), profil_sous_tab(:) = (/ 1,5 /)`
- ☞ `mode_distribution(:)=(/MPI_DISTRIBUTE_CYCLIC,MPI_DISTRIBUTE_NONE/)`

FIGURE 61 – Définition du type dérivé sur chaque processus

10 – Annexes
10.2 – Types de données dérivés
10.2.2 – Types dérivés numériques

- Le langage Fortran 95 introduit deux fonctions intrinsèques **selected_int_kind()** et **selected_real_kind()** qui permettent de définir la précision et/ou l'étendue d'un nombre entier, réel ou complexe
- MPI devait donc assurer la portabilité de ces types de données en définissant essentiellement les sous-programmes suivants : `MPI_TYPE_CREATE_F90_INTEGER()`, `MPI_TYPE_CREATE_F90_REAL()` et `MPI_TYPE_CREATE_F90_COMPLEX()`
- Ces sous-programmes renvoient des types dérivés MPI

Rappels (extrait du cours Fortran 95 de l'IDRIS)

- La fonction intrinsèque **selected_int_kind(e)** reçoit en argument un nombre entier e positif et retourne une valeur qui correspond au sous-type permettant de représenter les entiers n tels que $-10^{+e} < n < 10^{+e}$
- La fonction intrinsèque **selected_real_kind(p,e)** admet deux arguments optionnels positifs p et e (toutefois l'un des deux doit obligatoirement être fourni) indiquant respectivement la précision (nombre de chiffres décimaux significatifs) et l'étendue (la plage des nombres représentables en machine) désirées. Elle retourne un entier correspondant au sous-type permettant de représenter les réels x tels que $10^{-e} < |x| < 10^{+e}$. (Fin de l'extrait)
  Exemple : on souhaite représenter le nombre réel 12345.1234568 sur $p = 8$ chiffres significatifs avec une étendue par défaut. Au mieux, ce nombre aurait la valeur $x = 12345.123$ en machine. (Fin des rappels)

```
1  program precision
2    use mpi
3    implicit none
4    integer, parameter            :: n=101, preci=12
5    integer                       :: rang, mon_complex, code
6    ! Le sous-type k représentera une précision d'au moins 12 chiffres significatifs
7    integer, parameter            :: k=selected_real_kind(preci)
8    complex(kind=k), dimension(n) :: donnee
9
10   call MPI_INIT(code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
12
13   ! Construction du type MPI mon_complex associé à la précision demandée
14   call MPI_TYPE_CREATE_F90_COMPLEX(preci, MPI_UNDEFINED, mon_complex, code)
15
16   if (rang == 0) donnee(:) = cmplx(rang, rang, kind=k)
17
18   ! Utilisation du type mon_complex
19   call MPI_BCAST(donnee, n, mon_complex, 0, MPI_COMM_WORLD, code)
20
21   call MPI_FINALIZE(code)
22 end program precision
```

Remarques

- En réalité, les types générés par ces sous-programmes sont prédéfinis par MPI
- Par conséquent, ils ne peuvent pas être libérés avec `MPI_TYPE_FREE()`
- De plus, il n'est pas nécessaire de les valider avec `MPI_TYPE_COMMIT()`

10 – Annexes
10.3 – Optimisations

Dans un programme, il arrive parfois que l'on soit contraint de boucler un certain nombre de fois sur un envoi et une réception de message où la valeur des données manipulées change mais pas leurs adresses en mémoire ni leurs nombres ni leurs types. En outre, l'appel à un sous-programme de communication à chaque itération peut être très pénalisant à la longue d'où l'intérêt des communications persistantes. Elles consistent à :

1. créer un schéma persistant de communication une fois pour toutes (à l'extérieur de la boucle) ;
2. activer réellement la requête d'envoi ou de réception dans la boucle ;
3. libérer, si nécessaire, la requête en fin de boucle.

envoi *standard*	`MPI_SEND_INIT()`
envoi *synchroneous*	`MPI_SSEND_INIT()`
envoi *buffered*	`MPI_BSEND_INIT()`
réception *standard*	`MPI_RECV_INIT()`

```
23    if (rang == 0) then
24       do k = 1, 1000
25          call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
26          call sgetrf(na, na, a, na, pivota, code)
27          call MPI_WAIT(requete0,statut,code)
28          c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
29       end do
30    elseif (rang == 1) then
31       do k = 1, 1000
32          call sgetrf(na, na, a, na, pivota, code)
33          call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
34          call sgetrf(nb, nb, b, nb, pivotb, code)
35          call MPI_WAIT(requete1,statut,code)
36          a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
37       end do
38    end if
```

```
> mpiexec -n 2 AOptimiser
Temps : 235 secondes
```

L'utilisation d'un schéma persistant de communication permet de cacher la latence et de réduire les surcoûts induits par chaque appel aux sous-programmes de communication dans la boucle. Le gain peut être important lorsque ce mode de communication est réellement implémenté.

```
23    if (rang == 0) then
24       call MPI_SSEND_INIT(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
25       do k = 1, 1000
26          call MPI_START(requete0,code)
27          call sgetrf(na, na, a, na, pivota, code)
28          call MPI_WAIT(requete0,statut,code)
29          c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
30       end do
31    elseif (rang == 1) then
32       call MPI_RECV_INIT(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
33       do k = 1, 1000
34          call sgetrf(na, na, a, na, pivota, code)
35          call MPI_START(requete1,code)
36          call sgetrf(nb, nb, b, nb, pivotb, code)
37          call MPI_WAIT(requete1,statut,code)
38          a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
39       end do
40    end if
```

```
> mpiexec -n 2 AOptimiser
Temps :  235 secondes
```

Ici, l'implémentation MPI et/ou l'infrastructure matérielle de la machine ne permettent malheureusement pas une utilisation efficace du mode persistant.

Remarques :

- ☞ Une communication activée par `MPI_START()` sur une requête créée par l'un des sous-programmes `MPI_xxxx_INIT()` est équivalente à une communication non bloquante `MPI_Ixxxx()`.
- ☞ Pour redéfinir un nouveau schéma persistant avec la même requête, il faut auparavant libérer celle associée à l'ancien schéma en appelant le sous-programme `MPI_REQUEST_FREE`(`requete`,code).
- ☞ Ce sous-programme ne libèrera la requête `requete` qu'une fois que la communication associée sera réellement terminée.

10 – Annexes
10.4 – Communicateurs
10.4.1 – Intra et intercommunicateurs

- Les communicateurs que nous avons construits jusqu'à présent sont des intracommunicateurs car ils ne permettent pas que des processus appartenant à des communicateurs distincts puissent communiquer entre eux.
- Des processus appartenant à des intracommunicateurs distincts ne peuvent communiquer que s'il existe un lien de communication entre ces intracommunicateurs.
- Un intercommunicateur est un communicateur qui permet l'établissement de ce lien de communication.
- Le sous-programme MPI `MPI_INTERCOMM_CREATE()` permet de construire des intercommunicateurs.
- Le couplage des modèles océan/atmosphère illustre bien l'utilité des intra et intercommunicateurs...

10 – Annexes
10.4 – Communicateurs
10.4.1 – Intra et intercommunicateurs

Exemple récapitulatif sur les intra et intercommunicateurs

FIGURE 62 – Couplage océan/atmosphère

```
1  program OceanAtmosphere
2    use mpi
3    implicit none
4
5    integer,parameter :: tag1=1111, tag2=2222
6    integer           :: RangMonde, NombreIntraComm, couleur, code, &
7                         IntraComm, CommOceanAtmosphere, CommVisuAtmosphere
8
9    call MPI_INIT(code)
10   call MPI_COMM_RANK(MPI_COMM_WORLD,RangMonde,code)
11
12   ! Construction des 3 IntraCommunicateurs
13   NombreIntraComm = 3
14   couleur = mod(RangMonde,NombreIntraComm) ! = 0,1,2
15   call MPI_COMM_SPLIT(MPI_COMM_WORLD,couleur,RangMonde,IntraComm,code)
```

```
16    ! Construction des deux InterCommunicateurs et et appel des sous-programmes de calcul
17    select case(couleur)
18     case(0)
19      ! InterCommunicateur OceanAtmosphere pour que le groupe 0 communique
20      ! avec le groupe 1
21      call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,1,tag1,CommOceanAtmosphere, &
22                                code)
23      call ocean(IntraComm,CommOceanAtmosphere)
24
25     case(1)
26      ! InterCommunicateur OceanAtmosphere pour que le groupe 1 communique
27      ! avec le groupe 0
28      call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,0,tag1,CommOceanAtmosphere, &
29                                code)
30
31      ! InterCommunicateur CommVisuAtmosphere pour que le groupe 1 communique
32      ! avec le groupe 2
33      call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,2,tag2,CommVisuAtmosphere,code)
34      call atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)
35
36     case(2)
37      ! InterCommunicateur CommVisuAtmosphere pour que le groupe 2 communique
38      ! avec le groupe 1
39      call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,1,tag2,CommVisuAtmosphere,code)
40      call visualisation(IntraComm,CommVisuAtmosphere)
41    end select
42
43  end program OceanAtmosphere
```

```
44 subroutine ocean(IntraComm,CommOceanAtmosphere)
45   use mpi
46   implicit none
47
48   integer,parameter                    :: n=1024,tag1=3333
49   real,dimension(n)                    :: a,b,c
50   integer                              :: rang,code,germe(1),IntraComm,CommOceanAtmosphere
51   integer,dimension(MPI_STATUS_SIZE) :: statut
52   integer,intrinsic                    :: irtc
53
54   ! Les processus 0, 3, 6 dédiés au modèle océanographique effectuent un calcul
55   germe(1)=irtc()
56   call random_seed(put=germe)
57   call random_number(a)
58   call random_number(b)
59   call random_number(c)
60   a(:) = b(:) * c(:)
61
62   ! Les processus impliqués dans le modèle océan effectuent une opération collective
63   call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_SUM,IntraComm,code)
64
65   ! Rang du processus dans IntraComm
66   call MPI_COMM_RANK(IntraComm,rang,code)
67
68   ! Échange de messages avec les processus associés au modèle atmosphérique
69   call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
70                             CommOceanAtmosphere,statut,code)
71
72   ! Le modèle océanographique tient compte des valeurs atmosphériques
73   a(:) = b(:) * c(:)
74 end subroutine ocean
```

```
75 subroutine atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)
76   use mpi
77   implicit none
78
79   integer,parameter                    :: n=1024,tag1=3333,tag2=4444
80   real,dimension(n)                    :: a,b,c
81   integer                              :: rang,code,germe(1),IntraComm, &
82                                           CommOceanAtmosphere,CommVisuAtmosphere
83   integer,dimension(MPI_STATUS_SIZE) :: statut
84   integer,intrinsic                    :: irtc
85
86   ! Les processus 1, 4, 7 dédiés au modèle atmosphérique effectuent un calcul
87   germe(1)=irtc()
88   call random_seed(put=germe)
89
90   call random_number(a)
91   call random_number(b)
92   call random_number(c)
93
94   a(:) = b(:) + c(:)
```

```
95    ! Les processus dédiés au modèle atmosphère effectuent une opération collective
96    call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_MAX,IntraComm,code)
97
98    ! Rang du processus dans IntraComm
99    call MPI_COMM_RANK(IntraComm,rang,code)
100
101   ! Échange de messages avec les processus dédiés au modèle océanographique
102   call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
103                             CommOceanAtmosphere,statut,code)
104
105   ! Le modèle atmosphère tient compte des valeurs océanographiques
106   a(:) = b(:) * c(:)
107
108   ! Envoi des résultats aux processus dédiés à la visualisation
109   if (rang == 0 .or. rang == 1) then
110      call MPI_SSEND(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,code)
111   end if
112
113 end subroutine atmosphere
```

```
114 subroutine visualisation(IntraComm,CommVisuAtmosphere)
115   use mpi
116   implicit none
117
118   integer,parameter                    :: n=1024,tag2=4444
119   real,dimension(n)                    :: a,b,c
120   integer                              :: rang,code,IntraComm,CommVisuAtmosphere
121   integer,dimension(MPI_STATUS_SIZE) :: statut
122
123   ! Les processus 2 et 5 sont chargés de la visualisation
124   call MPI_COMM_RANK(IntraComm,rang,code)
125
126   ! Réception des valeurs du champ à tracer
127   call MPI_RECV(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,statut,code)
128
129   print*,'Moi, processus ',rang,' je trace mon champ A : ',a(:)
130
131 end subroutine visualisation
```

10 – Annexes
10.4 – Communicateurs
10.4.2 – Graphe de processus

Il arrive cependant que dans certaines applications (géométries complexes), la décomposition de domaine ne soit plus une grille régulière mais un graphe dans lequel un sous-domaine peut avoir un ou plusieurs voisins quelconques. Le sous-programme MPI_DIST_GRAPH_CREATE() permet alors de définir une topologie de type graphe.

```
integer, intent(in)                                :: comm_ancien,n,info
integer, dimension(:),intent(in)                   :: source,degres
integer, dimension(nb_voisins_max),intent(in)      :: liste_voisins,poids
logical, intent(in)                                :: reorganisation

integer, intent(out)                               :: comm_nouveau, code

call MPI_DIST_GRAPH_CREATE(comm_ancien,n,sources,degres,liste_voisins,poids,&
                           info,reorganisation, comm_nouveau,code)
```

Les tableaux d'entiers **poids** et **liste_voisins** permettent de définir le poids attribué et la liste des voisins ceci pour chacun des nœuds.

FIGURE 63 – Graphe de processus

Numéro de processus	liste_voisins
0	1
1	0,5,2,3
2	1,3,4
3	1,2,4
4	3,2,5
5	1,4

```
poids (:) = 1;liste_voisins = (/ 1, 0,5,2,3, 1,3,4, 1,2,4, 3,2,5, 1,4 /)
```

Deux autres fonctions sont utiles pour connaître :

- le nombre de voisins pour un processus donné :

```
integer, intent(in)                                :: comm_nouveau
integer, intent(in)                                :: rang
integer, intent(out)                               :: nb_voisins
integer, intent(out)                               :: code

call MPI_GRAPH_NEIGHBORS_COUNT(comm_nouveau,rang,nb_voisins,code)
```

- la liste des voisins pour un processus donné :

```
integer, intent(in)                                :: comm_nouveau
integer, intent(in)                                :: rang
integer, intent(in)                                :: nb_voisins
integer, dimension(nb_voisins_max), intent(out) :: voisins
integer, intent(out)                               :: code

call MPI_GRAPH_NEIGHBORS(comm_nouveau,rang,nb_voisins,voisins,code)
```

```
1  program graphe
2
3    use mpi
4    implicit none
5
6    integer                                :: rang,rang_monde,code,nb_processus,comm_graphe,&
7                                              n, nb_voisins,i,iteration=0
8    integer, parameter                     :: etiquette=100
9    integer, dimension(16)                 :: liste_voisins,poids
10   integer, allocatable,dimension(:)      :: voisins,sources,degres
11   integer, dimension(MPI_STATUS_SIZE):: statut
12   real                                   :: propagation, & ! Propagation du feu
13                                                            ! depuis les voisins
14                                              feu=0.,      & ! Valeur du feu
15                                              bois=1.,     & ! Rien n'a encore brûlé
16                                              arret=1.       ! Tout a brûlé si arret <= 0.01
17
18   call MPI_INIT(code)
19   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_processus,code)
20   call MPI_COMM_RANK(MPI_COMM_WORLD,rang_monde,code)
21
22   allocate(sources(0:nb_processus-1),degres(0:nb_processus-1))
23
24   if (rang_monde==0) then
25      n=nb_processus
26   else
27      n=0
28   end if
```

```
29    do i=0,nb_processus-1
30       sources(i)=i
31    end do
32
33    degres(:)= (/ 1,4,3,3,3,2 /)
34
35    liste_voisins(:)= (/ 1,0,5,2,3,1,3,4,1,2,4,3,2,5,1,4 /)
36
37    poids(:) = 1
38
39    call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD,n,sources,degres,liste_voisins,poids, &
40                               MPI_INFO_NULL,.false.,comm_graphe,code)
41    call MPI_COMM_RANK(comm_graphe,rang,code)
42
43    if (rang == 2) feu=1.          ! Le feu se déclare arbitrairement sur la parcelle 2
44
45    call MPI_GRAPH_NEIGHBORS_COUNT(comm_graphe,rang,nb_voisins,code)
46
47    allocate(voisins(nb_voisins)) ! Allocation du tableau voisins
48
49    call MPI_GRAPH_NEIGHBORS(comm_graphe,rang,nb_voisins,voisins,code)
```

```
50   do while (arret > 0.01)       ! On arrête dès qu'il n'y a plus rien à brûler
51
52      do i=1,nb_voisins          ! On propage le feu aux voisins
53         call MPI_SENDRECV(minval((/1.,feu/)),1,MPI_REAL,voisins(i),etiquette, &
54                           propagation,       1,MPI_REAL,voisins(i),etiquette, &
55                           comm_graphe,statut,code)
56         ! Le feu se développe en local sous l'influence des voisins
57         feu=1.2*feu + 0.2*propagation*bois
58         bois=bois/(1.+feu)      ! On calcule ce qui reste de bois sur la parcelle
59      end do
60
61      call MPI_ALLREDUCE(bois,arret,1,MPI_REAL,MPI_SUM,comm_graphe,code)
62
63      iteration=iteration+1
64      print '("Itération ",i2," parcelle ",i2," bois=",f5.3)',iteration,rang,bois
65      call MPI_BARRIER(comm_graphe,code)
66      if (rang  == 0) print '("--")'
67   end do
68
69   deallocate(voisins)
70
71   call MPI_FINALIZE(code)
72
73 end program graphe
```

```
> mpiexec -n 6 graphe
Iteration  1 parcelle  0 bois=1.000
Iteration  1 parcelle  3 bois=0.602
Iteration  1 parcelle  5 bois=0.953
Iteration  1 parcelle  4 bois=0.589
Iteration  1 parcelle  1 bois=0.672
Iteration  1 parcelle  2 bois=0.068
--
.................................
Iteration 10 parcelle  0 bois=0.008
Iteration 10 parcelle  1 bois=0.000
Iteration 10 parcelle  3 bois=0.000
Iteration 10 parcelle  5 bois=0.000
Iteration 10 parcelle  2 bois=0.000
Iteration 10 parcelle  4 bois=0.000
--
```

(a) Itération 0 (b) Itération 1 (c) Itération 2

(d) Itération 3 (e) Itération 4 (f) Itération 10

FIGURE 64 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt

10 – Annexes
10.5 – Gestion de processus
10.5.1 – Introduction

- La gestion dynamique des processus a été l'un des apports majeurs de MPI-2
- C'est la possibilité de créer (et dans certaines conditions de supprimer) des processus durant l'exécution de l'application
- Le démarrage d'une application reste dépendant de l'environnement d'exécution qui sera défini par le constructeur

L'activation d'un ou plusieurs processus peut se faire selon deux modes bien distincts :

❶ Le mode maître-ouvriers : l'un au moins des processus d'une application active un ou plusieurs autres processus. Les processus ouvriers ainsi activés dynamiquement exécutent un code soit identique (modèle SPMD) soit différent (modèle MPMD) du processus maître qui les a généré.

❷ Le mode client-serveur : un ou plusieurs processus d'une application serveur (lancée au préalable) sont en attente de connexion d'un ou plusieurs processus d'une application cliente (lancée plus tard). Une fois la connexion effectuée, un lien de communication est établi entre les processus des deux applications.

10 – Annexes
10.5 – Gestion de processus
10.5.2 – Mode maître-ouvriers

Activation d'un programme unique

Dans l'exemple que nous allons décrire, nous suivons le modèle MPMD où un programme parallèle « maître » active, avec le sous-programme `MPI_COMM_SPAWN()`, plusieurs copies d'un programme parallèle unique « ouvriers ». Ce sous-programme est collectif. Il est bloquant pour tous les processus appartenant au communicateur incluant le processus « maître », celui qui active réellement les processus ouvriers. Nous aurons également besoin du sous-programme `MPI_INTERCOMM_MERGE()` qui permet de fusionner dans un même intracommunicateur deux communicateurs liés par un intercommunicateur donné.

```
> mpiexec -n 3 -max_np 7 maitre
```

FIGURE 65 – Utilisation de MPI_COMM_SPAWN()

```
1  program maitre
2    use mpi
3    implicit none
4
5    integer :: nb_procs_maitres,nb_procs_ouvriers=4,nb_procs,rang,code
6    integer :: inter_comm,intra_comm,rang_maitre=1
7    logical :: drapeau=.false.
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs_maitres, code)
11
12   ! Activation des processus ouvriers
13   call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
14                       rang_maitre, MPI_COMM_WORLD,inter_comm,MPI_ERRCODES_IGNORE,code)
15
16   ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
17   ! des processus seront ordonnés selon la valeur de l'argument drapeau
18   call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)
19
20   call MPI_COMM_SIZE(intra_comm, nb_procs, code)
21   call MPI_COMM_RANK(intra_comm, rang, code)
22
23   print *,"maitre    de rang ", rang, "; intra_comm de taille ",nb_procs, &
24           "; mon MPI_COMM_WORLD de taille ", nb_procs_maitres
25
26   call MPI_FINALIZE(code)
27 end program maitre
```

```
program ouvriers
  use mpi
  implicit none
  integer :: nb_procs_ouvriers, nb_procs, rang, code
  integer :: inter_comm, intra_comm
  logical :: drapeau=.true.

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs_ouvriers, code)

  ! Ai-je un processus maître ?
  call MPI_COMM_GET_PARENT(inter_comm, code)
  if (inter_comm == MPI_COMM_NULL) then
     print *,'Pas de processus maître'
     call MPI_FINALIZE(code)
     stop
  end if

  ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
  ! des processus seront ordonnés selon la valeur de l'argument drapeau
  call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)

  call MPI_COMM_SIZE(intra_comm, nb_procs, code)
  call MPI_COMM_RANK(intra_comm, rang, code)

  print *,"ouvrier de rang ", rang, "; intra_comm de taille ",nb_procs, &
          "; mon MPI_COMM_WORLD de taille : ", nb_procs_ouvriers

  call MPI_FINALIZE(code)
end program ouvriers
```

```
> mpiexec -n 3 -max_np 7 maitre
maitre  de rang 0 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
maitre  de rang 2 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
ouvrier de rang 5 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
ouvrier de rang 4 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
ouvrier de rang 6 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
maitre  de rang 1 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
ouvrier de rang 3 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
```

Noter que, dans ce cas, la fusion des communicateurs ne modifie pas le rang des processus associés au programme maître.

Signification de MPI_COMM_SELF

MPI_COMM_SELF est un communicateur prédéfini par MPI. À l'appel de MPI_COMM_SPAWN(), ce communicateur inclut un et un seul processus. Ce processus est celui qui active les processus ouvriers. MPI_COMM_SELF n'incluera donc que le processus maître.

```
...
! Activation des processus ouvriers
rang_maitre=1
nb_procs_ouvriers=4
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
                    rang_maitre, MPI_COMM_SELF, inter_comm,MPI_ERRCODES_IGNORE,code)
...
```

```
> mpiexec -n 3 -max_np 7 maitre
```

FIGURE 66 – Signification de MPI_COMM_SELF

Signification de MPI_INFO_NULL

- Ce paramètre est aussi utilisé dans d'autres contextes, notamment dans les entrées/sorties parallèles avec MPI-IO.
- S'il est spécifié à l'appel du sous-programme MPI_COMM_SPAWN() (ou bien MPI_COMM_SPAWN_MULTIPLE() que l'on introduira par la suite), il indique le mode de recherche par défaut des programmes « ouvriers ». Les constructeurs peuvent toutefois définir d'autres valeurs plus spécifiques à leur environnement.
- Le mode de recherche par défaut voudra dire généralement que les programmes « ouvriers » se trouvent sur la machine locale et dans le même répertoire que le programme « maître »
- Pour modifier ces valeurs par défaut, il faut utiliser les sous-programmes MPI_INFO_CREATE(), MPI_INFO_SET() et MPI_INFO_FREE()

```
integer :: rang_maitre=1, nb_procs_ouvriers=4, info_spawn
...
! Redéfinition du mode de recherche des programmes ouvriers
call MPI_INFO_CREATE(info_spawn, code)
call MPI_INFO_SET(info_spawn, "host", "aleph.idris.fr", code)
call MPI_INFO_SET(info_spawn, "wdir", "/workdir/idris/rech/rgrp001", code)

! Activation des processus ouvriers
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, info_spawn, &
                    rang_maitre, MPI_COMM_SELF, inter_comm,MPI_ERRCODES_IGNORE,code)

! Libération du paramètre info_spawn
call MPI_INFO_FREE(info_spawn, code)
...
```

Signification de MPI_UNIVERSE_SIZE

MPI_UNIVERSE_SIZE est une clef MPI dont on peut connaître la valeur grâce au sous-programme MPI_COMM_GET_ATTR(). Si la version de MPI utilisée l'implémente, il est associé au nombre total de processus qu'un utilisateur peut activer.

```
...
! Nombre de processus maximum que l'on peut activer
call MPI_COMM_GET_ATTR(MPI_COMM_WORLD,MPI_UNIVERSE_SIZE,nb_procs_total,logique,code)

if (logique) then
   ! Ici nb_procs_ouvriers vaudra 7-3=4
   nb_procs_ouvriers = nb_procs_total - nb_procs_maitres
else
   print *,"MPI_UNIVERSE_SIZE n''est pas supporté"
   nb_procs_ouvriers = 4
end if

! Activation des processus ouvriers
rang_maitre=1
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
                    rang_maitre, MPI_COMM_WORLD,inter_comm,MPI_ERRCODES_IGNORE,code)
...
```

Activation de programmes multiples

Dans ce second exemple, nous suivons le modèle MPMD où un programme parallèle « maître » active avec le sous-programme `MPI_COMM_SPAWN_MULTIPLE()` plusieurs copies de 4 programmes parallèles différents « ouvriers1 », ..., « ouvriers4 ». Ce sous-programme est collectif. Il est bloquant pour tous les processus appartenant au communicateur incluant le processus « maître », celui qui active réellement l'ensemble des processus ouvriers.
Dans ce cas, pour des raisons de performance, il est conseillé de ne pas appeler le sous-programme `MPI_COMM_SPAWN()` autant de fois qu'il y a de programmes ouvriers mais plutôt d'appeler le sous-programme `MPI_COMM_SPAWN_MULTIPLE()` une seule fois pour activer l'ensemble des programmes ouvriers.

```
1  program maitre
2    use mpi
3    implicit none
4
5    integer :: inter_comm,intra_comm, rang_maitre=1,code
6    logical :: drapeau=.false.
7    ! On souhaite activer 4 programmes ouvriers
8    integer, parameter :: nb_prog_ouvriers=4
9    character(len=12), dimension(nb_prog_ouvriers) :: ouvriers
10   integer, dimension(nb_prog_ouvriers) :: nb_procs_ouvriers=(/3,2,1,2/),infos
11   ! Un code d'erreur par programme et par processus activé
12   integer, dimension(8) :: codes_erreurs   ! 8=3+2+1+2
13
14   call MPI_INIT(code)
15
16   ouvriers(:)      = (/"ouvriers1", "ouvriers2", "ouvriers3", "ouvriers4"/)
17   infos(:)         = MPI_INFO_NULL
18   codes_erreurs(:) = MPI_ERRCODES_IGNORE
19
20   ! Activation de plusieurs programmes ouvriers
21   call MPI_COMM_SPAWN_MULTIPLE(nb_prog_ouvriers,ouvriers,MPI_ARGVS_NULL, &
22                                nb_procs_ouvriers,infos,rang_maitre,MPI_COMM_WORLD, &
23                                inter_comm,codes_erreurs,code)
24
25   ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
26   ! des processus seront ordonnés selon la valeur de l'argument drapeau
27   call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)
28
29   ! Inclure ici le code correspondant aux calculs à faire par les processus maîtres
30   ...
31
32   call MPI_FINALIZE(code)
33 end program maitre
```

```
> mpiexec -n 3 -max_np 11 maitre
```

FIGURE 67 – Utilisation de MPI_COMM_SPAWN_MULTIPLE()

Remarques

- ☞ MPI_COMM_SPAWN() et MPI_COMM_SPAWN_MULTIPLE() sont des sous-programmes collectifs qui doivent être appelés par l'ensemble des processus du communicateur incluant le processus maître
- ☞ Attention à l'ordre des processus dans le nouvel intracommunicateur issu de la fusion des deux communicateurs associés à l'intercommunicateur renvoyé par MPI_COMM_SPAWN() ou MPI_COMM_SPAWN_MULTIPLE()
- ☞ Contrairement à ce que l'on aurait obtenu si MPI_COMM_SPAWN() avait été utilisé pour activer plusieurs programmes, MPI_COMM_SPAWN_MULTIPLE() inclut tous les processus de tous les programmes ouvriers dans le même communicateur MPI_COMM_WORLD
- ☞ Tous les arguments de MPI_COMM_SPAWN_MULTIPLE() ont la même signification que ceux de MPI_COMM_SPAWN()
- ☞ Dans MPI_COMM_SPAWN_MULTIPLE(), certains arguments sont toutefois transformés en tableaux du fait de la multiplicité des programmes ouvriers à activer
- ☞ Avec MPI_COMM_SPAWN_MULTIPLE(), les variables MPI_INFO_NULL, MPI_COMM_SELF et MPI_UNIVERSE_SIZE conservent les mêmes caractéristiques que celles que l'on a vues avec MPI_COMM_SPAWN()

10 – Annexes
10.5 – Gestion de processus
10.5.3 – Mode client-serveur

Deux programmes indépendants peuvent établir entre eux un lien de communication alors que leurs processus ne partagent aucun communicateur. Cette situation peut se produire :

- ☞ lorsque deux parties d'une application démarrent indépendamment l'une de l'autre et veulent, à un moment de leur vie, échanger des informations ;
- ☞ lorsqu'une application parallèle serveur accepte des connexions de plusieurs applications parallèles clientes ;
- ☞ lorsqu'un outil de visualisation veut s'attacher à un processus en cours d'exécution pour extraire certaines informations.

L'environnement (machines, systèmes d'exploitation, etc.) dans lequel s'exécute l'application serveur peut être différent de celui des applications clientes.

FIGURE 68 – Schéma d'application client-serveur

Processus serveur

Pour accepter un lien de communication avec le processus client, le processus serveur passe par trois étapes :

1. ouverture d'un port de communication : `MPI_OPEN_PORT()` ;
2. publication d'un nom arbitraire de connexion : `MPI_PUBLISH_NAME()` ;
3. acceptation de la connexion : `MPI_COMM_ACCEPT()`

Pour fermer ce lien de communication, de même :

1. fermeture de la connexion avec le processus client : `MPI_COMM_DISCONNECT()` ;
2. retrait du nom de connexion : `MPI_UNPUBLISH_NAME()` ;
3. fermeture du port de communication : `MPI_CLOSE_PORT()`.

Le processus serveur exécutera la séquence de code suivante :

```
...
integer                          :: rang_serveur=2, inter_comm, code
character(len=MPI_MAX_PORT_NAME) :: nom_de_port

...
if ( rang == rang_serveur ) then
   call MPI_OPEN_PORT(MPI_INFO_NULL,nom_de_port,code)
   call MPI_PUBLISH_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)
end if

call MPI_COMM_ACCEPT(nom_de_port, MPI_INFO_NULL, rang_serveur, MPI_COMM_WORLD, &
                     inter_comm, code)

! Inclure ici le code du serveur
...
call MPI_COMM_DISCONNECT(inter_comm, code)

if ( rang == rang_serveur ) then
   call MPI_UNPUBLISH_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)
   call MPI_CLOSE_PORT(nom_de_port, code)
end if
...
```

Processus client

Le client doit tout d'abord se connecter au port de communication du serveur, ce qui se réalise en deux étapes :

1. recherche du port de communication associé au nom publié par le serveur : MPI_LOOKUP_NAME() ;
2. connexion avec le serveur : MPI_COMM_CONNECT().

Ensuite, pour interrompre la connexion avec le serveur, le client devra obligatoirement appeler le sous-programme MPI_COMM_DISCONNECT().

```
...
integer                          :: rang_client=1, inter_comm, code
character(len=MPI_MAX_PORT_NAME) :: nom_de_port

...
if ( rang == rang_client ) &
   call MPI_LOOKUP_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)

call MPI_COMM_CONNECT(nom_de_port, MPI_INFO_NULL, rang_client, MPI_COMM_WORLD, &
                      inter_comm, code)

! Inclure ici le code du client
...

call MPI_COMM_DISCONNECT(inter_comm, code)
...
```

Remarques

- ☞ MPI_COMM_CONNECT(), MPI_COMM_ACCEPT() et MPI_COMM_DISCONNECT() sont des sous-programmes collectifs (donc bloquants), bien qu'un seul processus participe à la connexion de part et d'autre
- ☞ MPI_CLOSE_PORT() libère le port de communication (le serveur devient injoignable) alors que MPI_COMM_DISCONNECT() ne fait que rompre le lien de communication entre deux intracommunicateurs pour qu'éventuellement un autre lien puisse s'établir sur le même port
- ☞ MPI_COMM_SELF peut être utilisé à la place de MPI_COMM_WORLD dans les appels aux sous-programmes MPI_COMM_ACCEPT() et MPI_COMM_CONNECT(). Dans ce cas, la connexion s'établit entre deux intracommunicateurs ne contenant chacun que le processus appelant l'un ou l'autre sous-programme.
- ☞ Sans le mécanisme des sous-programmes MPI_PUBLISH_NAME() et MPI_LOOKUP_NAME(), on aurait été amené à préciser explicitement au processus client par un moyen quelconque (sur l'entrée standard ou par l'intermédiaire d'un fichier), le nom du port de communication renvoyé par le processus serveur

10 – Annexes
10.5 – Gestion de processus
10.5.4 – Suppression de processus

- S'il est possible de créer des processus, on devrait pouvoir les supprimer
- Or, il n'existe pas de sous-programme MPI spécifique pour supprimer un processus généré en cours d'exécution
- En revanche, il est toujours possible de diriger (ex. par échange de messages) l'exécution de ce processus vers une « terminaison normale »
- Un processus MPI se termine normalement à l'appel du sous-programme `MPI_FINALIZE()` et à la fin de l'exécution du programme principal
- Il existe trois contraintes :
  1. le nouveau communicateur `MPI_COMM_WORLD` généré ne doit contenir que le processus dont on veut se débarrasser ;
  2. il ne doit exister aucun lien de communication (intercommunicateur) entre le communicateur `MPI_COMM_WORLD` contenant le processus père (ou serveur) et celui contenant le processus fils (ou client) à supprimer ;
  3. tout intracommunicateur contenant le processus à détruire doit être invalidé avant la terminaison du processus fils (ou client).

- Il faut également noter que
  - il n'est pas possible de se débarrasser d'un seul processus « ouvrier » si son communicateur `MPI_COMM_WORLD` inclut d'autres processus ;
  - dans ce cas, la terminaison ne s'effectue « proprement » que si tous les processus de `MPI_COMM_WORLD` appellent le sous-programme `MPI_FINALIZE()` et atteignent normalement la fin de l'exécution.

10 – Annexes
10.5 – Gestion de processus
10.5.5 – Compléments

- Dans certains cas, comme celui de MPI_UNIVERSE_SIZE, les implémentations ont des clefs spécifiques dont la valeur peut être connue grâce au sous-programme :

```
integer, intent(in)                          :: comm, clef
integer(kind=MPI_ADDRESS_KIND), intent(out) :: valeur
logical, intent(out)                         :: logique
integer, intent(out)                         :: code

call MPI_COMM_GET_ATTR(comm, clef, valeur, logique, code)
```

- On peut cependant modifier la valeur associée à une clef définie au préalable, en utilisant le sous-programme :

```
integer, intent(inout)                      :: comm
integer, intent(in)                         :: clef
integer(kind=MPI_ADDRESS_KIND), intent(in) :: valeur
integer, intent(out)                        :: code

call MPI_COMM_SET_ATTR(comm, clef, valeur, code)
```

- Plus généralement, on peut définir un couple (clef, valeur) spécifique à son application par l'intermédiaire des sous-programmes MPI_COMM_CREATE_KEYVAL() et MPI_COMM_SET_ATTR()

10 – Annexes
10.6 – RMA

Introduction

Diverses approches existent pour transférer des données entre deux processus distincts. Parmi les plus utilisées, on trouve :

- les communications point à point par échange de messages (MPI, etc.) ;
- les communications par copies de mémoire à mémoire (accès direct à la mémoire d'un processus distant). Appelées RMA pour *Remote Memory Access* ou OSC pour *One Sided Communication*, c'est l'un des apports majeurs de MPI 2.

Rappel : concept de l'échange de messages

FIGURE 69 – L'échange de messages

Dans le concept de l'échange de messages, un émetteur (source) va envoyer un message à un processus destinataire (cible) qui va faire la démarche de recevoir ce message. Cela nécessite que l'émetteur comme le destinataire prennent part à la communication. Ceci peut être contraignant et difficile à mettre en œuvre dans certains algorithmes (par exemple lorsqu'il faut gérer un compteur global).

Concept des copies de mémoire à mémoire

Le concept de communication par copies de mémoire à mémoire n'est pas nouveau, MPI ayant simplement unifié les solutions constructeurs déjà existantes (telles que shmem (CRAY), lapi (IBM), ...) en proposant ses propres primitives RMA. Via ces sous-programmes, un processus a directement accès (en lecture, écriture ou mise à jour) à la mémoire d'un autre processus distant. Dans cette approche, le processus distant n'a donc pas à intervenir dans la procédure de transfert des données.

Les principaux avantages sont les suivants :

- des performances améliorées lorsque le matériel le permet,
- une programmation plus simple de certains algorithmes.

Approche RMA de MPI

L'approche RMA de MPI peut être divisée en trois étapes successives :

1. définition sur chaque processus d'une zone mémoire (fenêtre mémoire locale) visible et susceptible d'être accédée par des processus distants ;
2. déclenchement du transfert des données directement de la mémoire d'un processus à celle d'un autre processus. Il faut alors spécifier le type, le nombre et la localisation initiale et finale des données.
3. achèvement des transferts en cours par une étape de synchronisation, les données étant alors réellement disponibles pour les calculs.

10 – Annexes
10.6 – RMA
10.6.1 – Notion de fenêtre mémoire

Notion de fenêtre mémoire

Tous les processus prenant part à une opération de copie de mémoire à mémoire doivent spécifier quelle partie de leur mémoire va être accessible aux autres processus ; c'est la notion de fenêtre mémoire. L'opération collective MPI_WIN_CREATE() permet la création d'un objet MPI fenêtre. Cet objet est composé, pour chaque processus, d'une zone mémoire spécifique appelée fenêtre mémoire locale. Au niveau de chaque processus, une fenêtre mémoire locale est caractérisée par son adresse de départ, sa taille en octets (qui peut être nulle) et la taille de l'unité de déplacement à l'intérieur de cette fenêtre (en octets). Ces caractéristiques peuvent être différentes sur chacun des processus.

Interface

```
TYPE(*), intent(in)  :: adresse
integer, intent(in)  :: deplacement, info, comm
integer(kind=MPI_ADDRESS_KIND) :: taille
integer, intent(out) :: zone, code

call MPI_WIN_CREATE(adresse, taille, deplacement, info, comm, zone, code)
```

Exemple

FIGURE 70 – Création de deux objets MPI fenêtre, win1 et win2

Gestion des fenêtres

- Une fois les transferts terminés, on doit libérer la fenêtre avec le sous-programme `MPI_WIN_FREE()`.
- `MPI_WIN_GET_ATTR()` permet de connaître les caractéristiques d'une fenêtre mémoire locale en utilisant les mots clés `MPI_WIN_BASE`, `MPI_WIN_SIZE` ou `MPI_WIN_DISP_UNIT`.

Remarque :

- Le choix de l'unité de déplacement associée à la fenêtre mémoire locale est important (indispensable dans un environnement hétérogène et facilitant le codage dans tous les cas). L'obtention de la taille d'un type MPI se fait en appelant le sous-programme `MPI_TYPE_SIZE()`.

```
program fenetre

  use mpi
  implicit none

  integer :: code, rang, taille_reel, win, n=4
  integer (kind=MPI_ADDRESS_KIND) :: dim_win, taille, base, unite
  real(kind=kind(1.d0)), dimension(:), allocatable :: win_local
  logical :: flag

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
  call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION,taille_reel,code)

  if (rang==0) n=0
  allocate(win_local(n))
  dim_win = taille_reel*n

  call MPI_WIN_CREATE(win_local, dim_win, taille_reel, MPI_INFO_NULL, &
                      MPI_COMM_WORLD, win, code)

  call MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, taille, flag, code)
  call MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, code)
  call MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, unite, flag, code)
  call MPI_WIN_FREE(win,code)
  print *,"processus", rang,"taille, base, unite = ", &
         taille, base, unite
  call MPI_FINALIZE(code)
end program fenetre
```

```
> mpiexec -n 3 fenetre
 processus  1  taille, base, unite =   32  17248330400  8
 processus  0  taille, base, unite =   0  2  8
 processus  2  taille, base, unite =   32  17248330400  8
```

10 – Annexes
10.6 – RMA
10.6.2 – Transfert des données

Transfert des données

MPI permet à un processus de lire (MPI_GET()), d'écrire (MPI_PUT()) et de mettre à jour (MPI_ACCUMULATE()) des données situées dans la fenêtre mémoire locale d'un processus distant.

On nomme origine le processus qui fait l'appel au sous-programme d'initialisation du transfert et cible le processus qui possède la fenêtre mémoire locale qui va être utilisée dans la procédure de transfert.

Lors de l'initialisation du transfert, le processus cible n'appelle aucun sous-programme MPI. Toutes les informations nécessaires sont spécifiées sous forme de paramètres lors de l'appel au sous-programme MPI par l'origine.

Paramètres

En particulier, on trouve :

- des paramètres ayant rapport à l'origine :
  - le type des éléments ;
  - leur nombre ;
  - l'adresse mémoire du premier élément.
- des paramètres ayant rapport à la cible :
  - le rang de la cible ainsi que l'objet MPI fenêtre, ce qui détermine de façon unique une fenêtre mémoire locale ;
  - un déplacement dans cette fenêtre locale ;
  - le nombre et le type des données à transférer.

```
program exemple_put
integer :: nb_orig=10, nb_cible=10, cible=1, win, code
integer (kind=MPI_ADDRESS_KIND) :: deplacement=40
integer, dimension(10) :: B
...
call MPI_PUT(B,nb_orig,MPI_INTEGER,cible,deplacement,nb_cible,MPI_INTEGER,win,code)
...
end program exemple_put
```

Exemple d'un MPI_PUT

Remarques

- La syntaxe de MPI_GET est identique à celle de MPI_PUT, seul le sens de transfert des données étant inversé.
- Les sous-programmes de transfert de données RMA sont des primitives non bloquantes (choix délibéré de MPI).
- Sur le processus cible, les seules données accessibles sont celles contenues dans la fenêtre mémoire locale.
- MPI_ACCUMULATE() admet parmi ses paramètres une opération qui doit être soit du type MPI_REPLACE, soit l'une des opérations de réduction prédéfinies : MPI_SUM, MPI_PROD, MPI_MAX, etc. Ce ne peut en aucun cas être une opération définie par l'utilisateur.

10 – Annexes
10.6 – RMA
10.6.3 – Achèvement du transfert : la synchronisation

Achèvement du transfert : la synchronisation

Le transfert des données débute après l'appel à l'un des sous-programmes non bloquants (`MPI_PUT()`, ...). Mais quand le transfert est-il terminé et les données réellement disponibles ?

$$\Downarrow$$

Après une synchronisation qui est à la charge du programmeur.

Ces synchronisations peuvent être classées en deux types :

- synchronisation de type cible active (opération collective, tous les processus associés à la fenêtre prenant part à la synchronisation) ;
- synchronisation de type cible passive (seul le processus origine appelle le sous-programme de synchronisation).

10 – Annexes
10.6 – RMA
10.6.3 – Achèvement du transfert : la synchronisation

Synchronisation de type cible active

- Se fait en utilisant le sous-programme MPI `MPI_WIN_FENCE()`.
- `MPI_WIN_FENCE()` est une opération collective sur tous les processus associés à l'objet MPI fenêtre.
- `MPI_WIN_FENCE()` agit comme une barrière de synchronisation. Elle attend la fin de tous les transferts de données (RMA ou non) utilisant la fenêtre mémoire locale et initiés depuis le dernier appel à `MPI_WIN_FENCE()`.
- Cette primitive va permettre de séparer les parties calcul du code (où l'on utilise des données de la fenêtre mémoire locale via des *load* ou des *store*) des parties de transfert de données de type RMA.
- Un argument *assert* de la primitive `MPI_WIN_FENCE()`, de type entier, permet d'affiner son comportement en vue de meilleures performances. Diverses valeurs sont prédéfinies `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, `MPI_MODE_NOPRECEDE`, `MPI_MODE_NOSUCCEED`. Une valeur de zéro pour cet argument est toujours valide.

Remarques

- Le fait d'avoir choisi des sous-programmes RMA d'initialisation du transfert non bloquants et une synchronisation pour l'achèvement des transferts en cours autorise l'implémentation à regrouper lors de l'exécution divers transferts vers la même cible en un transfert unique. L'effet de la latence est ainsi réduit et les performances améliorées.
- Le caractère collectif de la synchronisation a pour conséquence qu'on n'a pas réellement affaire à ce que l'on appelle du « *One Sided Communication* »... En fait tous les processus du communicateur vont devoir prendre part à la synchronisation, ce qui perd de son intérêt !

Du bon usage de MPI_WIN_FENCE

- Il faut s'assurer qu'entre deux appels successifs à MPI_WIN_FENCE() il n'y a soit que des affectations locales (*load/store*) sur des variables contenues dans la fenêtre mémoire locale du processus, soit que des opérations RMA de type MPI_PUT() ou MPI_ACCUMULATE(), mais jamais les deux en même temps !

0
MPI_WIN_FENCE()
|
MPI_WIN_FENCE()
MPI_PUT()
MPI_WIN_FENCE()

1
MPI_WIN_FENCE()
`win_loc(:) = win_loc(:) + 1.0`
MPI_WIN_FENCE()
①
MPI_WIN_FENCE()

Le programme précédent est-il conforme au bon usage de MPI_WIN_FENCE() ?

Tout dépend de la portion de code représentée par ①. Si celle-ci n'engendre pas de *load/store* sur la fenêtre locale (affectation ou utilisation d'une variable stockée dans la fenêtre locale), alors c'est bon ; dans le cas contraire, le résultat est indéfini.

```
program ex_fence
  use mpi
  implicit none

  integer, parameter :: assert=0
  integer :: code, rang, taille_reel, win, i, nb_elements, cible, m=4, n=4
  integer (kind=MPI_ADDRESS_KIND) :: deplacement, dim_win
  real(kind=kind(1.d0)), dimension(:), allocatable :: win_local, tab

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
  call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION,taille_reel,code)

  if (rang==0) then
     n=0
     allocate(tab(m))
  endif

  allocate(win_local(n))
  dim_win = taille_reel*n

  call MPI_WIN_CREATE(win_local, dim_win, taille_reel, MPI_INFO_NULL, &
                      MPI_COMM_WORLD, win, code)
```

```
if (rang==0) then
   tab(:) = (/ (i, i=1,m) /)
else
   win_local(:) = 0.0
end if

call MPI_WIN_FENCE(assert,win,code)
if (rang==0) then
   cible = 1; nb_elements = 2; deplacement = 1
   call MPI_PUT(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, deplacement, &
                nb_elements, MPI_DOUBLE_PRECISION, win, code)
end if

call MPI_WIN_FENCE(assert,win,code)
if (rang==0) then
   tab(m) = sum(tab(1:m-1))
else
   win_local(n) = sum(win_local(1:n-1))
endif

call MPI_WIN_FENCE(assert,win,code)
if (rang==0) then
   nb_elements = 1; deplacement = m-1
   call MPI_GET(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, deplacement, &
                nb_elements, MPI_DOUBLE_PRECISION, win, code)
end if
```

```
  call MPI_WIN_FENCE(assert,win,code)
  if (rang==0) then
     tab(m) = sum(tab(1:m-1))
  else
     win_local(:) = win_local(:) + 1
  endif

  call MPI_WIN_FENCE(assert,win,code)
  if (rang==0) then
     nb_elements = m-1; deplacement = 1
     call MPI_ACCUMULATE(tab(2), nb_elements, MPI_DOUBLE_PRECISION, cible, &
                         deplacement, nb_elements, MPI_DOUBLE_PRECISION, &
                         MPI_SUM, win, code)
  end if

  call MPI_WIN_FENCE(assert,win,code)
  call MPI_WIN_FREE(win,code)

  if (rang==0) then
     print *,"processus", rang, "tab=",tab(:)
  else
     print *,"processus", rang, "win_local=",win_local(:)
  endif

  call MPI_FINALIZE(code)
end program ex_fence
```

Exemple récapitulatif

FIGURE 71 – Exemple récapitulatif correspondant au code ex_fence

Quelques précisions et restrictions...

- Il est possible de travailler sur des fenêtres mémoire locales différentes qui se recouvrent, même si cela n'est pas recommandé (une telle utilisation impliquant de trop nombreuses restrictions). Dans la suite on supposera ne pas être dans ce cas.
- Il faut toujours séparer par un appel à `MPI_WIN_FENCE()` un *store* et l'appel à un sous-programme `MPI_PUT()` ou `MPI_ACCUMULATE()` accédant à la même fenêtre mémoire locale même à des endroits différents ne se recouvrant pas.
- Entre deux appels successifs au sous-programme `MPI_WIN_FENCE()`, on a les contraintes suivantes :
  - les sous-programmes `MPI_PUT()` n'admettent pas le recouvrement à l'intérieur d'une même fenêtre mémoire locale. En d'autres termes, les zones mémoires mises en jeu lors d'appels à plusieurs sous-programmes `MPI_PUT()` agissant sur la même fenêtre mémoire locale, ne doivent pas se recouvrir ;

Quelques précisions et restrictions...

- les sous-programmes `MPI_ACCUMULATE()` admettent le recouvrement à l'intérieur d'une même fenêtre mémoire locale, à la condition que les types des données et l'opération de réduction utilisés soient identiques lors de tous ces appels ;
- les sous-programmes `MPI_PUT()` et `MPI_ACCUMULATE()` utilisés consécutivement n'admettent pas le recouvrement à l'intérieur d'une même fenêtre mémoire locale ;
- un *load* et un appel au sous-programme `MPI_GET()` peuvent accéder concurremment à n'importe quelle partie de la fenêtre locale, pourvu qu'elle n'ait pas été mise à jour auparavant soit par un *store*, soit lors de l'appel à un sous-programme de type `MPI_PUT()` ou `MPI_ACCUMULATE()`.

10 – Annexes
10.6 – RMA
10.6.3 – Achèvement du transfert : la synchronisation

Synchronisation de type cible passive

- Se fait via les appels aux sous-programmes MPI `MPI_WIN_LOCK()` et `MPI_WIN_UNLOCK()`.
- Contrairement à la synchronisation par `MPI_WIN_FENCE()` (qui est une opération collective de type barrière), ici seul le processus origine va participer à la synchronisation. De ce fait tous les appels nécessaires au transfert des données (initialisation du transfert, synchronisation) ne font intervenir que le processus origine ; c'est du vrai « *One Sided Communication* ».
- Les opérations de *lock* et d'*unlock* ne s'appliquent qu'à une fenêtre mémoire locale donnée (i.e. identifiée par un numéro de processus cible et un objet MPI fenêtre). La période qui commence au *lock* et se termine à l'*unlock* est appelée une période d'accès à la fenêtre mémoire locale. Ce n'est que durant cette période que le processus origine va avoir accès à la fenêtre mémoire locale du processus cible.

Synchronisation de type cible passive

- Pour l'utiliser, il suffit pour le processus origine d'entourer l'appel aux primitives RMA d'initialisation de transfert de données par `MPI_WIN_LOCK()` et `MPI_WIN_UNLOCK()`. Pour le processus cible, aucun appel de sous-programmes `MPI` n'est à faire.
- Lorsque `MPI_WIN_UNLOCK()` rend la main, tous les transferts de données initiés après le `MPI_WIN_LOCK()` sont terminés.
- Le premier argument de `MPI_WIN_LOCK()` permet de spécifier si le fait de faire plusieurs accès simultanés via des opérations de RMA sur une même fenêtre mémoire locale est autorisé (`MPI_LOCK_SHARED`) ou non (`MPI_LOCK_EXCLUSIVE`).
- Une utilisation basique des synchronisations de type cible passive consiste à créer des versions bloquantes des RMA (*put, get, accumulate*) sans que la cible ait besoin de faire appel à des sous-programmes `MPI`.

```
subroutine get_bloquant(orig_addr, orig_count, orig_datatype, target_rank,  &
                        target_disp, target_count, target_datatype, win, code)
integer, intent(in) :: orig_count, orig_datatype, target_rank, target_count, &
                       target_datatype, win
integer, intent(out) :: code
integer(kind=MPI_ADDRESS_KIND), intent(in) :: target_disp
real(kind=kind(1.d0)), dimension(:) :: orig_addr

call MPI_WIN_LOCK(MPI_LOCK_SHARED, target_rank, 0, win, code)
call MPI_GET(orig_addr, orig_count, orig_datatype, target_rank, target_disp, &
             target_count, target_datatype, win, code)
call MPI_WIN_UNLOCK(target_rank, win, code)
end subroutine get_bloquant
```

Remarque concernant les codes Fortran

Pour être portable, lors de l'utilisation des synchronisations de type cible passive (`MPI_WIN_LOCK()`, `MPI_WIN_UNLOCK()`), il faut allouer la fenêtre mémoire avec `MPI_ALLOC_MEM()`. Cette fonction admet comme argument des pointeurs de type C (i.e. pointeurs Fortran CRAY, qui ne font pas partie de la norme Fortran95). Dans le cas où ces derniers ne sont pas disponibles, il faut utiliser un programme C pour faire l'allocation de la fenêtre mémoire...

10 – Annexes
10.6 – RMA
10.6.4 – Conclusions

Conclusions

- Les concepts RMA de MPI sont compliqués à mettre en œuvre sur des applications non triviales. Une connaissance approfondie de la norme est nécessaire pour ne pas tomber dans les nombreux pièges.
- Les performances peuvent être très variables d'une implémentation à l'autre.
- L'intérêt du concept RMA de MPI réside essentiellement dans l'approche cible passive. C'est seulement dans ce cas que l'utilisation des sous-programmes RMA est réellement indispensable (application nécessitant qu'un processus accède à des données appartenant à un processus distant sans interruption de ce dernier...).

10 – Annexes
10.7 – MPI-IO

Il est possible d'obtenir certaines informations spécifiques sur un fichier.

```
program open02
  use mpi
  implicit none
  integer           :: rang,descripteur,attribut,longueur,code
  character(len=80) :: libelle
  logical           :: defini

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  call MPI_FILE_OPEN(MPI_COMM_WORLD,"fichier.txt",MPI_MODE_RDWR + MPI_MODE_CREATE, &
                     MPI_INFO_NULL,descripteur,code)
  call MPI_FILE_GET_INFO(descripteur,attribut,code)
  call MPI_INFO_GET_VALUELEN(attribut,"cb_nodes",longueur,defini,code)
  if (defini) then
     call MPI_INFO_GET(attribut,"cb_nodes",longueur,libelle,defini,code)
     if (rang==0) print *,"Fichier ‘fichier.txt’ sur ",libelle(1:longueur)," processus"
  end if
  call MPI_INFO_FREE(attribut,code)
  call MPI_FILE_CLOSE(descripteur,code)

  call MPI_FINALIZE(code)
end program open02
```

```
> mpiexec -n 2 open02
Fichier ‘fichier.txt’ sur 2 processus
```

1 Introduction

2 Environnement

3 Communications point à point

4 Communications collectives

5 Modèles de communication

6 Types de données dérivés

7 Communicateurs

8 MPI-IO

9 Conclusion

10 Annexes

11 Index
11.1 Index des constantes MPI 334
11.2 Index des sous-programmes MPI 337

mpi 25
mpi.h 25
MPI_ADDRESS_KIND . . 110, 120, 129–131, 133, 134, 136, 221, 304, 309, 312, 315, 321, 329
MPI_ANY_SOURCE 38, 44
MPI_ANY_TAG 38, 44
MPI_ARGV_NULL 284, 287, 290, 291
MPI_ARGVS_NULL 293
MPI_BSEND_OVERHEAD 87, 90
MPI_BYTE 212
MPI_CHARACTER 131
MPI_COMM_NULL 145, 285
MPI_COMM_SELF 287, 288, 290, 295, 301
MPI_COMM_WORLD . . . 26–29, 35, 41, 42, 44, 49, 52, 55, 58, 61, 64, 65, 68, 74, 77, 85, 90, 112–117, 123, 127, 128, 131, 132, 136, 137, 142, 147, 151, 153, 163, 164, 169, 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217–219, 221, 222, 226, 227, 229, 232, 241, 247, 248, 251, 252, 258, 261, 262, 266, 267, 275, 276, 284–286, 291, 293, 295, 299–303, 312, 321, 332
MPI_COMPLEX 106, 241
MPI_DISTRIBUTE_BLOCK 244–247, 254
MPI_DISTRIBUTE_CYCLIC 244, 250, 251, 255
MPI_DISTRIBUTE_NONE 244, 245, 255
MPI_DOUBLE_PRECISION 312, 321–323
MPI_ERRCODES_IGNORE 284, 287, 290, 291, 293
MPI_IN_PLACE 79
MPI_INFO_NULL . 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 216, 217, 219, 222, 227, 229, 232, 276, 284, 287, 291, 293, 295, 299, 300, 312, 321, 332

MPI_INTEGER 35, 41, 42, 44, 52, 74, 77, 85, 90, 106, 110, 118, 128, 131, 135–137, 184, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 216, 217, 219, 222, 224, 227, 229, 232, 248, 252, 315
MPI_LOCK_EXCLUSIVE 328
MPI_LOCK_SHARED 328, 329
MPI_LOGICAL 131
MPI_MAX 270, 316
MPI_MAX_PORT_NAME 299, 300
MPI_MODE_CREATE 178, 185, 332
MPI_MODE_NOPRECEDE 318
MPI_MODE_NOPUT 318
MPI_MODE_NOSTORE 318
MPI_MODE_NOSUCCEED 318
MPI_MODE_RDONLY .187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 219, 222, 227, 229, 232
MPI_MODE_RDWR 178, 332
MPI_MODE_WRONLY 185
MPI_OFFSET_KIND 185, 187, 198, 209, 214, 217, 218, 221, 226
MPI_ORDER_C 125
MPI_ORDER_FORTRAN 125, 128, 216, 217, 219, 224, 248, 252
MPI_PROC_NULL 38, 160
MPI_PROD 77, 316
MPI_REAL 55, 58, 61, 65, 68, 106, 108–110, 112–116, 118, 123, 131, 147, 169, 261, 262, 268, 270, 271, 277
MPI_REPLACE 316

MPI_SEEK_CUR 208, 209
MPI_SEEK_END 208
MPI_SEEK_SET 208, 209
MPI_SOURCE 44
MPI_STATUS_IGNORE 38, 41, 85, 90
MPI_STATUS_SIZE 34, 35, 39, 43, 44, 100, 112, 114, 116, 123, 127, 131, 136, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 218, 221, 226, 229, 232, 247, 251, 268, 269, 271, 275
MPI_SUM 74, 268, 277, 316, 323
MPI_TAG 44
MPI_UNDEFINED 145, 258
MPI_UNIVERSE_SIZE 291, 295, 304
MPI_WIN_BASE 311, 312
MPI_WIN_DISP_UNIT 311, 312
MPI_WIN_SIZE 311, 312
mpif.h 25

MPI_ABORT 27, 27
MPI_ACCUMULATE 313, 316, 320, 323, 325, 326
MPI_ALLGATHER 48, 60, 60, 61, 79
MPI_ALLGATHERV 79
MPI_ALLOC_MEM 330
MPI_ALLREDUCE 48, 70, 76, 76, 77, 268, 270, 277
MPI_ALLTOALL 48, 67, 67, 68, 79
MPI_ALLTOALLV 79
MPI_ALLTOALLW 79
MPI_BARRIER 48, 49, 49, 238, 277
MPI_BCAST 48, 51, 51, 52, 60, 70, 147, 258
MPI_BSEND 81, 87, 88, 90
MPI_BUFFER_ATTACH 87, 87, 90
MPI_BUFFER_DETACH 87, 87, 90
MPI_CART_COORDS 158, 158, 159, 164, 169
MPI_CART_CREATE 150, 150, 151, 153, 164, 169
MPI_CART_RANK 156, 156, 157
MPI_CART_SHIFT 160, 160–162, 164
MPI_CART_SUB 167, 167, 169
MPI_CLOSE_PORT 298, 299, 301
MPI_COMM_ACCEPT 298, 299, 301
MPI_COMM_CONNECT 300, 301
MPI_COMM_CREATE 143, 148
MPI_COMM_CREATE_KEYVAL 304
MPI_COMM_DISCONNECT 298–301
MPI_COMM_DUP 143

MPI_COMM_FREE 143, 147
MPI_COMM_GET_ATTR 291, 304
MPI_COMM_GET_PARENT 285
MPI_COMM_GROUP 148
MPI_COMM_RANK .. 28, 28, 29, 35, 41, 44, 52, 55, 58, 61, 64, 68, 74, 77, 85, 90, 112, 114, 116, 123, 127, 131, 136, 147, 164, 169, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 218, 221, 226, 229, 232, 241, 247, 251, 258, 266, 268, 270, 271, 275, 276, 284, 285, 312, 321, 332
MPI_COMM_SET_ATTR 304
MPI_COMM_SIZE 28, 28, 29, 44, 55, 58, 61, 64, 68, 74, 77, 163, 247, 251, 275, 284, 285
MPI_COMM_SPAWN 282–284, 287, 289–292, 295
MPI_COMM_SPAWN_MULTIPLE 289, 292–295
MPI_COMM_SPLIT 143, 145, 145–147, 167, 266
MPI_DIMS_CREATE 155, 155, 163
MPI_DIST_GRAPH_CREATE 272, 276
MPI_EXSCAN 79
MPI_FILE_CLOSE 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 219, 222, 227, 229, 232, 332
MPI_FILE_GET_INFO 177, 332
MPI_FILE_GET_POSITION 208
MPI_FILE_GET_POSITION_SHARED 208
MPI_FILE_IREAD 182, 229
MPI_FILE_IREAD_AT 181, 227
MPI_FILE_IREAD_SHARED 182
MPI_FILE_IWRITE 182

MPI_FILE_IWRITE_AT 181
MPI_FILE_IWRITE_SHARED 182
MPI_FILE_OPEN . 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 219, 222, 227, 229, 232, 332
MPI_FILE_READ 182, 190, 192, 209, 217, 219, 222
MPI_FILE_READ_ALL 182, 200, 202, 204
MPI_FILE_READ_ALL_BEGIN 182
MPI_FILE_READ_ALL_END 182
MPI_FILE_READ_AT 181, 187
MPI_FILE_READ_AT_ALL 181, 198
MPI_FILE_READ_AT_ALL_BEGIN 181
MPI_FILE_READ_AT_ALL_END 181
MPI_FILE_READ_ORDERED 182, 206
MPI_FILE_READ_ORDERED_BEGIN 182, 232
MPI_FILE_READ_ORDERED_END 182, 232
MPI_FILE_READ_SHARED 182, 195
MPI_FILE_SEEK 208, 209
MPI_FILE_SEEK_SHARED 208
MPI_FILE_SET_VIEW 214, 214, 216, 217, 219, 222
MPI_FILE_WRITE 182
MPI_FILE_WRITE_ALL 182
MPI_FILE_WRITE_ALL_BEGIN 182
MPI_FILE_WRITE_ALL_END 182
MPI_FILE_WRITE_AT 181, 185
MPI_FILE_WRITE_AT_ALL 181

MPI_FILE_WRITE_AT_ALL_BEGIN 181
MPI_FILE_WRITE_AT_ALL_END 181
MPI_FILE_WRITE_ORDERED 182
MPI_FILE_WRITE_ORDERED_BEGIN 182
MPI_FILE_WRITE_ORDERED_END 182
MPI_FILE_WRITE_SHARED 182
MPI_FINALIZE ... 25, 25, 29, 35, 41, 44, 52, 55, 58, 61, 65, 68, 74, 77, 85, 90, 113, 115, 117, 123, 128, 132, 137, 142, 147, 164, 169, 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 219, 222, 227, 229, 232, 241, 248, 252, 258, 277, 284, 285, 293, 302, 303, 312, 323, 332
MPI_GATHER 48, 57, 57, 58, 60, 63, 67, 79
MPI_GATHERV 63, 63, 65, 79
MPI_GET 313, 316, 322, 326, 329
MPI_GET_ADDRESS 130, 132
MPI_GRAPH_NEIGHBORS 274, 276
MPI_GRAPH_NEIGHBORS_COUNT 274, 276
MPI_GROUP_FREE 148
MPI_GROUP_INCL 148
MPI_IBSEND 81, 99, 99
MPI_INFO_CREATE 289, 290
MPI_INFO_FREE 289, 290, 332
MPI_INFO_GET 332
MPI_INFO_GET_VALUELEN 332
MPI_INFO_SET 289, 290

MPI_INIT 25, 25, 29, 35, 41, 44, 52, 55, 58, 61, 64, 68, 74, 77, 85, 90, 112, 114, 116, 123, 127, 131, 136, 142, 147, 163, 169, 178, 185, 187, 190, 192, 195, 198, 200, 202, 204, 206, 209, 217, 218, 221, 226, 229, 232, 241, 247, 251, 258, 266, 275, 284, 285, 293, 312, 321, 332
MPI_INTERCOMM_CREATE 264, 267
MPI_INTERCOMM_MERGE 282, 284, 285, 293
MPI_IRECV 81, 95, 98, 99, 99, 102, 261
MPI_IRSEND 81
MPI_ISEND 81, 95, 98, 99, 99, 102
MPI_ISSEND 81, 99, 99, 261
MPI_LOOKUP_NAME 300, 301
MPI_OP_CREATE 70, 241
MPI_OP_FREE 70
MPI_OPEN_PORT 298, 299
MPI_PUBLISH_NAME 298, 299, 301
MPI_PUT 313, 315–317, 320, 322, 325, 326
MPI_RECV 34, 34, 35, 42, 44, 81, 85, 90, 113, 115, 117, 132, 137, 248, 252, 271
MPI_RECV_INIT 262
MPI_REDUCE 48, 70, 73, 73, 74, 79, 241
MPI_REQUEST_FREE 263
MPI_RSEND 81, 94
MPI_SCAN 70, 79
MPI_SCATTER 48, 54, 54, 55, 79, 169
MPI_SCATTERV 79
MPI_SEND 33, 33–35, 42, 44, 81, 88, 91, 113, 115, 117, 132, 137, 248, 252

MPI_SENDRECV 38, 39, 39–42, 138, 277
MPI_SENDRECV_REPLACE 38, 43, 43, 121, 123, 128, 268, 270
MPI_SSEND 81, 84, 84, 85, 238, 270
MPI_SSEND_INIT 262
MPI_START 262, 263
MPI_TEST 100, 100, 225, 227
MPI_TESTALL 100, 100
MPI_TYPE_COMMIT . 111, 111, 112, 114, 116, 123, 128, 132, 137, 214, 216, 217, 219, 222, 224, 248, 252, 259
MPI_TYPE_CONTIGUOUS 106, 108, 108, 112
MPI_TYPE_CREATE_DARRAY 243, 248, 252
MPI_TYPE_CREATE_F90_COMPLEX 256, 258
MPI_TYPE_CREATE_F90_INTEGER 256
MPI_TYPE_CREATE_F90_REAL 256
MPI_TYPE_CREATE_HINDEXED 118, 120, 120
MPI_TYPE_CREATE_HVECTOR 110, 110, 118
MPI_TYPE_CREATE_RESIZED 134, 134, 136, 222
MPI_TYPE_CREATE_STRUCT 106, 129, 129, 130, 132, 224
MPI_TYPE_CREATE_SUBARRAY 124, 125, 128, 216, 217, 219, 224
MPI_TYPE_FREE 111, 111, 113, 115, 117, 123, 128, 132, 248, 252, 259
MPI_TYPE_GET_EXTENT 118, 133, 136, 137, 222
MPI_TYPE_INDEXED 106, 118, 119, 119, 123, 129, 135, 222
MPI_TYPE_SIZE 90, 118, 133, 133, 136, 185, 187, 198, 209, 219, 222, 224, 227, 311, 312, 321
MPI_TYPE_VECTOR 106, 109, 109, 114, 116, 135, 136
MPI_UNPUBLISH_NAME 298, 299

MPI_WAIT 95, 97, 98, 100, 100, 225, 229, 261, 262
MPI_WAITALL 100, 100, 102
MPI_WIN_CREATE 309, 309, 312, 321
MPI_WIN_FENCE 318, 320, 322, 323, 325, 327
MPI_WIN_FREE 311, 312, 323
MPI_WIN_GET_ATTR 311, 312
MPI_WIN_LOCK 327–330
MPI_WIN_UNLOCK 327–330