Message Passing Interface
(MPI-1)

Jalel Chergui
Isabelle Dupays
Denis Girou
Stéphane Requena
Philippe Wautelet
# MPI-1 — Plan

1 – Introduction .......................................................... 6
  1.1 – Définitions ......................................................... 6
  1.2 – Concepts de l’échange de messages .......................... 10
  1.3 – Historique .......................................................... 13
  1.4 – Évolutions en cours et à venir ............................... 14
  1.5 – Bibliographie ...................................................... 16

2 – Environnement .......................................................... 18
  2.1 – Description ......................................................... 18
  2.2 – Exemple .............................................................. 21

3 – Communications point à point .................................... 22
  3.1 – Notions générales .................................................. 22
  3.2 – Types de données de base ....................................... 25
  3.3 – Autres possibilités ................................................ 27
  3.4 – Exemple : anneau de communication ......................... 30

4 – Communications collectives ....................................... 35
  4.1 – Notions générales ................................................ 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 - Synchronisation globale : MPI_BARRIER()</td>
<td>37</td>
</tr>
<tr>
<td>4.3 - Diffusion générale : MPI_BCAST()</td>
<td>38</td>
</tr>
<tr>
<td>4.4 - Diffusion sélective : MPI_SCATTER()</td>
<td>40</td>
</tr>
<tr>
<td>4.5 - Collecte : MPI_GATHER()</td>
<td>43</td>
</tr>
<tr>
<td>4.6 - Collecte générale : MPI_ALLGATHER()</td>
<td>45</td>
</tr>
<tr>
<td>4.7 - Échanges croisés : MPI_ALLTOALL()</td>
<td>48</td>
</tr>
<tr>
<td>4.8 - Réductions réparties</td>
<td>51</td>
</tr>
<tr>
<td>4.9 - Compléments</td>
<td>60</td>
</tr>
<tr>
<td>5 - Optimisations</td>
<td>61</td>
</tr>
<tr>
<td>5.1 - Introduction</td>
<td>61</td>
</tr>
<tr>
<td>5.2 - Programme modèle</td>
<td>62</td>
</tr>
<tr>
<td>5.3 - Temps de communication</td>
<td>65</td>
</tr>
<tr>
<td>5.4 - Quelques définitions</td>
<td>66</td>
</tr>
<tr>
<td>5.5 - Que fournit MPI ?</td>
<td>70</td>
</tr>
<tr>
<td>5.6 - Envoi synchrone bloquant</td>
<td>72</td>
</tr>
<tr>
<td>5.7 - Envoi synchrone non-bloquant</td>
<td>74</td>
</tr>
</tbody>
</table>
## MPI-1 — Plan

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>Conseils 1</td>
<td>78</td>
</tr>
<tr>
<td>5.9</td>
<td>Communications persistantes</td>
<td>82</td>
</tr>
<tr>
<td>5.10</td>
<td>Conseils 2</td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td>Types de données dérivés</td>
<td>90</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Types contigus</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Types avec un pas constant</td>
<td>93</td>
</tr>
<tr>
<td>6.4</td>
<td>Autres sous-programmes</td>
<td>95</td>
</tr>
<tr>
<td>6.5</td>
<td>Exemples</td>
<td>96</td>
</tr>
<tr>
<td>6.6</td>
<td>Types homogènes à pas variable</td>
<td>102</td>
</tr>
<tr>
<td>6.7</td>
<td>Types hétérogènes</td>
<td>109</td>
</tr>
<tr>
<td>6.8</td>
<td>Sous-programmes annexes</td>
<td>113</td>
</tr>
<tr>
<td>6.9</td>
<td>Conclusion</td>
<td>114</td>
</tr>
<tr>
<td>7</td>
<td>Topologies</td>
<td>115</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>115</td>
</tr>
<tr>
<td>7.2</td>
<td>Topologies de processus</td>
<td>116</td>
</tr>
<tr>
<td>7.3</td>
<td>Topologies cartésiennes</td>
<td>117</td>
</tr>
</tbody>
</table>
7.4 – Graphe de processus ............................................................. 132
8 – Communicateurs ................................................................. 139
  8.1 – Introduction ................................................................. 139
  8.2 – Communicateur par défaut ............................................... 141
  8.3 – Groupes et communicateurs ............................................. 145
  8.4 – Communicateur issu d’un autre ........................................ 147
  8.5 – Subdiviser une topologie cartésienne ................................ 152
  8.6 – Intra et intercommunicateurs .......................................... 158
  8.7 – Exemple récapitulatif ..................................................... 159
  8.8 – Conclusion ................................................................. 166
9 – Évolution de MPI : MPI-2 .................................................... 167
1 – Introduction : définitions

1.1 – Définitions

1. Le 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 centrale allouée au processus ;
   - un processus s’exécute sur un processeur physique de la machine.

*Figure 1 – Modèle de programmation séquentiel*
Dans le modèle de programmation par échange de messages :

- le programme est écrit dans un langage classique (*Fortran*, *C*, *C++*, etc.) ;
- chaque processus exécute éventuellement des parties différentes d’un programme ;
- toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus ;
- 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
Le modèle d’exécution **SPMD** :

- *Single Program, Multiple Data* ;
- le même programme est exécuté par tous les processus ;
- toutes les machines supportent ce modèle de programmation et certaines ne supportent que celui-là ;
- c’est un cas particulier du modèle plus général **MPMD** (*Multiple Program, Multiple Data*), qu’il peut d’ailleurs émuler.

**Figure 3 – Single Program, Multiple Data**
Exemple en *Fortran* d’émulation *MPMD* en *SPMD*

```fortran
program spmd
    if (ProcessusMaître) then
        call LeMaitre(Arguments)
    else
        call LesEsclaves(Arguments)
    endif
end program spmd
```
1.2 – Concepts de l’échange de messages

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

![Diagram](image)

**Figure 4 – Échange d’un message**
1 – Introduction : concepts de l’échange de messages

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 5** – Constitution d’un message

![Diagramme de message](image)
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**

Ces sous-programmes peuvent être classés dans les grandes catégories suivantes :

1. environnement ;
2. communications point à point ;
3. communications collectives ;
4. types de données dérivés ;
5. topologies ;
6. groupes et communicateurs.
1.3 – Historique

- Novembre 92 (*Supercomputing ’92*) : « formalisation » d’un groupe de travail créé en avril 92 et décision d’adopter les structures et les méthodes du groupe HPF (*High Performance Fortran*)
- Participants, américains (essentiellement) et européens, aussi bien constructeurs que représentants du monde académique
- « Brouillon » de MPI-1 présenté en novembre 93 (*Supercomputing ’93*), finalisé en mars 1994
- Vise à la fois la *portabilité* et la garantie de *bonnes performances*
- MPI-2 publié en juillet 97, suite à des travaux ayant commencé au printemps 95
- « Standard » non élaboré par les organismes officiels de normalisation (ISO, ANSI, etc.)
- MPI 1.1 publié en 1995, 1.2 en 1997 et 1.3 en 2008, avec seulement des clarifications et des changements mineurs (principalement dans quelques dénominations)
- MPI 2.1 publié en juin 2008
Nouveaux groupes de travail constitués en novembre 2007 (à *Supercomputing ’07*) pour travailler sur l’évolution de MPI

Trois nouvelles versions prévues

MPI 2.1

- uniquement pour des clarifications mais aucun changement dans le standard 2.0,
- publié en juin 2008,
- voir [http://www.mpi-forum.org/mpi2_1](http://www.mpi-forum.org/mpi2_1)
  [http://meetings.mpi-forum.org/MPI_2.1_main_page.php](http://meetings.mpi-forum.org/MPI_2.1_main_page.php)

MPI 2.2

- corrections jugées nécessaires au standard 2.0 et « petites » additions,
- attendu en 2009,
- voir [http://meetings.mpi-forum.org/MPI_2.2_main_page.php](http://meetings.mpi-forum.org/MPI_2.2_main_page.php)
MPI 3.0

⇒ changements et ajouts importants par rapport à la version 2.2,
⇒ pour un meilleur support des applications actuelles et futures, notamment sur les machines massivement parallèles et many cores,
⇒ principaux changements actuellement envisagés :
  ⇒ communications collectives non bloquantes,
  ⇒ nouvelle implémentation des copies mémoire à mémoire,
  ⇒ tolérance aux pannes,
  ⇒ Fortran (2003-2008) bindings,
  ⇒ interfaçage d’outils externes (pour le débogage et les mesures de performance),
  ⇒ etc.
⇒ attendu en 2010.
⇒ voir http://meetings.mpi-forum.org/MPI_3.0_main_page.php
  https://svn.mpi-forum.org/trac/mpi-forum-web/wiki
1 – Introduction : bibliographie

1.5 – Bibliographie

Les spécifications de la norme actuelle 1.3 (mai 2008) :

Les spécifications de la norme actuelle 2.1 (juin 2008) :

Les principaux ouvrages :
http://www.hlrs.de/organization/par/services/models/mpi/mpi21


Des documentations complémentaires :
http://www.mpi-forum.org/docs/
http://www.mcs.anl.gov/mpi/
Implémentations *MPI* du domaine public : elles peuvent être installées sur un grand nombre d’architectures mais leurs performances sont en général en dessous de celles des implémentations constructeurs

1. **MPICH2** (la version 1.1 intègre la norme MPI 2.1) :
   http://www-unix.mcs.anl.gov/mpi/mpich2/
2. **Open MPI** (la version 1.3 intègre la norme MPI 2.1) :
   http://www.open-mpi.org/

Quelques bibliothèques scientifiques parallèles du domaine public :

1. **ScaLAPACK** : résolution de problèmes d’algèbre linéaire par des méthodes directes. Les sources sont téléchargeables sur le site
   http://www.netlib.org/scalapack/
2. **PLAPACK** : résolution de problèmes d’algèbre linéaire par des méthodes directes. Les sources sont téléchargeables sur le site
   http://www.cs.utexas.edu/~plapack/
3. **PETSc** : résolution de problèmes d’algèbre linéaire et non-linéaire par des méthodes itératives. Les sources sont téléchargeables sur le site
   http://www-fp.mcs.anl.gov/petsc/
4. **FFTW** : transformées de Fourier rapides. Les sources sont téléchargeables sur le site http://www.fftw.org
5. Documentation IDRIS sur les bibliothèques scientifiques :
   http://www.idris.fr/data/cours/parallel/para_b/choix_doc.html
2 – Environnement : description

2.1 – Description

Tout unité de programme appelant des sous-programmes MPI doit inclure un fichier d’en-têtes. En Fortran, il faut maintenant 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 :

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

call MPI_INIT(code)
```

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

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

call MPI_FINALIZE(code)
```
Toutes les opérations effectuées par MPI portent sur des **communicateurs**. Le communicateur par défaut est **MPI_COMM_WORLD** qui comprend tous les processus actifs.

![Diagram of MPI_COMM_WORLD](image.png)

**Figure 6 – Communicateur MPI_COMM_WORLD**
À tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme `MPI_COMM_SIZE()` :

```fortran
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) :

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

call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
```
2.2 – Exemple

```fortran
program qui_je_suis
  use mpi
  implicit none
  integer :: nb_procs, rang, code

  call MPI_INIT(code)

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

  print *, 'Je suis le processus ', rang, ' parmi ', nb_procs

  call MPI_FINALIZE(code)
end program qui_je_suis
```

```plaintext
> 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
```
Une communication dite **point à point** a lieu entre deux processus, l’un appelé processus **émetteur** et l’autre processus **récepteur** (ou **destinataire**).
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 :

① du rang du processus émetteur ;
② du rang du processus récepteur ;
③ de l’étiquette (tag) du message ;
④ du nom du communicateur qui définira le contexte de communication de l’opération.

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 qui seront vus au chapitre 5.
program point_a_point
  use mpi
  implicit none

  integer, dimension(MPI_STATUS_SIZE) :: statut
  integer, parameter :: etiquette=100
  integer :: rang, valeur, code

  call MPI_INIT(code)

  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)

  if (rang == 2) then
    valeur=1000
    call MPI_SEND(valeur, 1, MPI_INTEGER, 5, etiquette, MPI_COMM_WORLD, code)
  elseif (rang == 5) then
    call MPI_RECV(valeur, 1, MPI_INTEGER, 2, etiquette, MPI_COMM_WORLD, statut, code)
    print *, 'Moi, processus 5, j’ai reçu ', valeur, ’ du processus 2.’
  end if

  call MPI_FINALIZE(code)

end program point_a_point

> mpiexec -n 7 point_a_point

Moi, processus 5, j’ai reçu 1000 du processus 2
### 3.2 – Types de données de base

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

<table>
<thead>
<tr>
<th>Type MPI</th>
<th>Type Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Types hétérogènes</td>
</tr>
</tbody>
</table>
Tableau 2 – Principaux types de données de base (C)

<table>
<thead>
<tr>
<th>Type MPI</th>
<th>Type C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Types hétérogènes</td>
</tr>
</tbody>
</table>
3 – Communications point à point : autres possibilités

3.3 – 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 enchaînent 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 à l’aide de sous-programmes tels que MPI_TYPE_CONTIGUOUS(), MPI_TYPE_VECTOR(), MPI_TYPE_INDEXED() et MPI_TYPE_STRUCT() (voir le chapitre 6).
### Communications point à point : autres possibilités

**Figure 8** – Communication `sendrecv` entre les processus 0 et 1

```fortran
program sendrecv
  use mpi
  implicit none
  integer, dimension(MPI_STATUS_SIZE) :: statut
  integer, parameter :: etiquette=100
  integer :: rang,valeur,num_proc,code

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

  ! On suppose avoir exactement 2 processus
  num_proc=mod(rang+1,2)

  call MPI_SENDRECV(rang+1000,1,MPI_INTEGER,num_proc,etiquette,valeur,1,MPI_INTEGER, &
                   num_proc,etiquette,MPI_COMM_WORLD,statut,code)

  print *, 'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus ',num_proc

  call MPI_FINALIZE(code)
end program sendrecv
```

**INSTITUT DU DÉVELOPPEMENT ET DES RÉSSOURCES EN INFORMATIQUE SCIENTIFIQUE**

J. Chergui, I. Dupays, D. Girou, S. Requena, P. Wautelet
Attention ! Il convient de noter que si le sous-programme `MPI_SEND()` est implémenté de façon synchrone (voir le chapitre 5) dans la version de la bibliothèque `MPI` mise en œuvre, le code précédent serait en situation de verrouillage si à la place de l’ordre `MPI_SENDRECV()` on utilisait un ordre `MPI_SEND()` suivi d’un ordre `MPI_RECV()`. En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens. Pour des raisons de portabilité, il faut donc absolument éviter ces cas-là.

```
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)
```
3.4 – Exemple : anneau de communication

Figure 9 – Anneau de communication
Si tous les processus font un envoi puis une réception, toutes les communications pourront potentiellement démarrer simultanément et n’auront donc pas lieu en anneau (outre le problème déjà mentionné de portabilité, au cas où l’implémentation du \texttt{MPI\_SEND()} est faite de façon synchrone dans la version de la bibliothèque \texttt{MPI} mise en œuvre):

```fortran
... valeur=rang+1000
call \texttt{MPI\_SEND}(valeur,1,\texttt{MPI\_INTEGER},num\_proc\_suivant,etiquette,\texttt{MPI\_COMM\_WORLD},code)
call \texttt{MPI\_RECV}(valeur,1,\texttt{MPI\_INTEGER},num\_proc\_precedent,etiquette,\texttt{MPI\_COMM\_WORLD}, &
                        statut,code)
...```

\texttt{MPI-1 – Version 2.24 – Mars 2009}

J. Chergui, I. Dupays, D. Girou, S. Requena, P. Wautelet
Pour que les communications se fassent réellement en **anneau**, à l’image d’un passage de **jeton** entre processus, il faut procéder différemment et faire en sorte qu’un processus initie la chaîne :

![Figure 10 – Anneau de communication](image-url)
program anneau
  use mpi
  implicit none
  integer, dimension(MPI_STATUS_SIZE) :: statut
  integer, parameter :: etiquette=100
  integer :: nb_procs,rang,valeur, &
    num_proc_precedent,num_proc_suivant,code
  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  num_proc_suivant=mod(rang+1,nb_procs)
  num_proc_precedent=mod(nb_procs+rang-1,nb_procs)

  if (rang == 0) then
    call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
      MPI_COMM_WORLD,code)
    call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
      MPI_COMM_WORLD,statut,code)
  else
    call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
      MPI_COMM_WORLD,statut,code)
    call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
      MPI_COMM_WORLD,code)
  end if

  print *,Mois, proc. ’,rang,’ j’ai reçu ’,valeur,’ du proc. ’,num_proc_precedent
  call MPI_FINALIZE(code)
end program anneau
3 – Communications point à point : exemple anneau

`mpiexec -n 7 anneau`

Moi, proc. 1, j’ai reçu 1000 du proc. 0
Moi, proc. 2, j’ai reçu 1001 du proc. 1
Moi, proc. 3, j’ai reçu 1002 du proc. 2
Moi, proc. 4, j’ai reçu 1003 du proc. 3
Moi, proc. 5, j’ai reçu 1004 du proc. 4
Moi, proc. 6, j’ai reçu 1005 du proc. 5
Moi, proc. 0, j’ai reçu 1006 du proc. 6
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.
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, qu’elles soient d’un type prédéfini (somme, produit, maximum, minimum, etc.) ou d’un type personnel : `MPI_REDUCE()`;
   - opérations de réduction avec diffusion du résultat (il s’agit en fait d’un `MPI_REDUCE()` suivi d’un `MPI_BCAST()`) : `MPI_ALLREDUCE()`.
4.2 – Synchronisation globale : MPI_BARRIER() 

Figure 11 – Synchronisation globale : MPI_BARRIER() 

integer, intent(out) :: code 

call MPI_BARRIER(MPI_COMM_WORLD, code)
4.3 – Diffusion générale : MPI_BCAST()

Figure 12 – Diffusion générale : MPI_BCAST()
program bcast
    use mpi
    implicit none

    integer :: rang, valeur, code

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

    if (rang == 2) valeur = rang + 1000

    call MPI_BCAST(valeur, 1, MPI_INTEGER, 2, MPI_COMM_WORLD, code)

    print *, 'Moi, processus ', rang, ', j''ai reçu ', valeur, ' du processus 2'

    call MPI_FINALIZE(code)

end program bcast
4.4 – Diffusion sélective : MPI_SCATTER()
program scatter
  use mpi
  implicit none

  integer, parameter :: nb_valeurs=128
  integer :: nb_procs, rang, longueur_tranche, i, code
  real, allocatable, dimension(:) :: valeurs, donnees

  call MPI_INIT(code)

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

  longueur_tranche=nb_valeurs/nb_procs
  allocate(donnees(longueur_tranche))

  if (rang == 2) then
    allocate(valeurs(nb_valeurs))
    valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
  end if

  call MPI_SCATTER(valeurs, longueur_tranche, MPI_REAL, donnees, longueur_tranche, 
                   MPI_REAL, 2, MPI_COMM_WORLD, code)

  print *, 'Moi, processus ', rang, ', j''ai reçu ', donnees(1), ' à ', 
           donnees(longueur_tranche), ' du processus 2'

  call MPI_FINALIZE(code)
end program scatter
4 – Communications collectives : diffusion sélective

```
> mpiexec -n 4 scatter

Moi, processus 0, j’ai reçu 1001. à 1032. du processus 2
Moi, processus 1, j’ai reçu 1033. à 1064. du processus 2
Moi, processus 3, j’ai reçu 1097. à 1128. du processus 2
Moi, processus 2, j’ai reçu 1065. à 1096. du processus 2
```
4.5 – Collecte : MPI_GATHER()
program gather
  use mpi
  implicit none
  integer, parameter :: nb_valeurs=128
  integer :: nb_procs,rang,longueur_tranche,i,code
  real, dimension(nb_valeurs) :: donnees
  real, allocatable, dimension(:) :: valeurs

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

  longueur_tranche=nb_valeurs/nb_procs
  allocate(valeurs(longueur_tranche))

  valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)

  call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
                  MPI_REAL,2,MPI_COMM_WORLD,code)

  if (rang == 2) print *,’Moi, processus 2’, j’ai reçu ’,donnees(1),’ ... ’, &
      donnees(longueur_tranche+1),’ ... ’,donnees(nb_valeurs)’
  call MPI_FINALIZE(code)
end program gather

> mpiexec -n 4 gather

Moi, processus 2, j’ai reçu 1001. ... 1033. ... 1128.
4.6 – Collecte générale : MPI_ALLGATHER()
program allgather
  use mpi
  implicit none

  integer, parameter :: nb_valeurs=128
  integer :: nb_procs, rang, longueur_tranche, i, code
  real, dimension(nb_valeurs) :: donnees
  real, allocatable, dimension(:) :: valeurs

  call MPI_INIT(code)

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

  longueur_tranche=nb_valeurs/nb_procs
  allocate(valeurs(longueur_tranche))

  valeurs(:)=/(1000.+rang*longueur_tranche+i, i=1,longueur_tranche)/

  call MPI_ALLGATHER(valeurs, longueur_tranche, MPI_REAL, donnees, longueur_tranche, &
                     MPI_REAL, MPI_COMM_WORLD, code)

  print *, 'Moi, processus ', rang, ', j’ai reçu ', donnees(1), ', ... ', &
          donnees(longueur_tranche+1), ', ... ', donnees(nb_valeurs)

  call MPI_FINALIZE(code)
end program allgather
4 – Communications collectives : collecte générale

```bash
> mpiexec -n 4 allgather

Moi, processus 0, j’ai reçu 1001. ... 1033. ... 1128.
Moi, processus 1, j’ai reçu 1001. ... 1033. ... 1128.
Moi, processus 2, j’ai reçu 1001. ... 1033. ... 1128.
Moi, processus 3, j’ai reçu 1001. ... 1033. ... 1128.
```
4.7 – Échanges croisés : MPI_ALLTOALL()
program alltoall
    use mpi
    implicit none

    integer, parameter :: nb_valeurs=128
    integer :: nb_procs,rang,longueur_tranche,i,code
    real, dimension(nb_valeurs) :: valeurs,donnees

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

    valeurs(:)=/(1000.+rang*nb_valeurs+i,i=1,nb_valeurs)/
    longueur_tranche=nb_valeurs/nb_procs

    call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
                      MPI_REAL,MPI_COMM_WORLD,code)

    print *,’Moi, processus ’,rang,’ , j’ai reçu ’,donnees(1),’ ... ’, &
     donnees(longueur_tranche+1),’ ... ’,donnees(nb_valeurs)’

    call MPI_FINALIZE(code)
end program alltoall
> mpiexec -n 4 alltoall

<table>
<thead>
<tr>
<th>Moi, processus</th>
<th>j’ai reçu</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1001</td>
<td>...</td>
<td>1129</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>1065</td>
<td>...</td>
<td>1193</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1033</td>
<td>...</td>
<td>1161</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>1097</td>
<td>...</td>
<td>1225</td>
<td>...</td>
</tr>
</tbody>
</table>
4 – Communications collectives : réductions réparties

4.8 – 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 \( \text{SUM}(A(:)) \) ou la recherche de l’élément de valeur maximum dans un vecteur \( \text{MAX}(V(:)) \).

\textit{MPI} propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus, avec récupération du résultat sur un seul processus (\texttt{MPI_REDUCE()}) ou bien sur tous (\texttt{MPI_ALLREDUCE()}), qui est en fait seulement un \texttt{MPI_REDUCE()} suivi d’un \texttt{MPI_BCAST()}).

Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d’entre eux.

Le sous-programme \texttt{MPI_SCAN()} permet en plus d’effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du groupe.

Les sous-programmes \texttt{MPI_OP_CREATE()} et \texttt{MPI_OP_FREE()} permettent de définir des opérations de réduction personnelles.
Table 3 – Principales opérations de réduction prédéfinies (il existe aussi d’autres opérations logiques)

<table>
<thead>
<tr>
<th>Nom</th>
<th>Opération</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM</td>
<td>Somme des éléments</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Produit des éléments</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>Recherche du maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Recherche du minimum</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Recherche de l’indice du maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Recherche de l’indice du minimum</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>ET logique</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>OU logique</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>OU exclusif logique</td>
</tr>
</tbody>
</table>
Figure 17 – Réduction répartie (somme)
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 18 – Réduction répartie (produit) avec diffusion du résultat
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
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
Définition du produit terme à terme de deux vecteurs de nombres complexes

integer function mon_produit(vecteur1,vecteur2,longueur,type_donnee) result(inutilise)
  implicit none

  complex,dimension(longueur) :: vecteur1,vecteur2
  integer :: longueur,type_donnee,i

  do i=1,longueur
    vecteur2(i) = cmplx(real(vecteur1(i))*real(vecteur2(i)) - &
      aimag(vecteur1(i))*aimag(vecteur2(i)), &
      real(vecteur1(i))*aimag(vecteur2(i)) + &
      aimag(vecteur1(i))*real(vecteur2(i)))
  end do

  inutilise=0

end function mon_produit

> mpiexec -n 5 ma_reduction

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

5.1 – Introduction

L’optimisation doit être un souci essentiel lorsque la part des communications par rapport aux calculs devient assez importante.

L’optimisation des communications peut s’accomplir à différents niveaux dont les principaux sont :

1. recouvrir les communications par des calculs ;
2. éviter si possible la recopie du message dans un espace mémoire temporaire (buffering) ;
3. minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication.
5.2 – Programme modèle

program AOptimiser
  use mpi
  implicit none

  integer, parameter :: na=256, nb=200
  integer, parameter :: m=2048, etiquette=1111
  real, dimension(na,na) :: a
  real, dimension(nb,nb) :: b
  real, dimension(na) :: pivota
  real, dimension(nb) :: pivotb
  real, dimension(m,m) :: c
  integer :: rang, code
  real(kind=8) :: temps_debut, temps_fin, temps_fin_max
  integer, dimension(MPI_STATUS_SIZE) :: statut

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

  ! Initialisation des tableaux
  call random_number(a)
  call random_number(b)
  call random_number(c)
temps_debut = MPI_WTIME()
if (rang == 0) then
  ! Envoi d'un gros message
  call MPI_SEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
  ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
  call sgetrf(na, na, a, na, pivota, code)
  ! Calcul modifiant le contenu du tableau C
  c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
elseif (rang == 1) then
  ! Calcul (factorisation LU avec LAPACK)
  call sgetrf(na, na, a, na, pivota, code)
  ! Reception d'un gros message
  call MPI_RECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,code)
  ! Calcul dependant de la reception du message precedent
  a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
  ! Calcul independant de la reception du message precedent
  call sgetrf(nb, nb, b, nb, pivotb, code)
end if
temps_fin = (MPI_WTIME() - temps_debut)
!
! Obtention du temps de restitution maximum
!call MPI_REDUCE(temps_fin,temps_fin_max,1,MPI_DOUBLE_PRECISION, MPI_MAX,0, &
!  MPI_COMM_WORLD,code)
if (rang == 0) print('("Temps : ",f6.3," secondes")'),temps_fin_max
!
call MPI_FINALIZE(code)
end program AOptimiser
Hors communications, le maximum des temps de calcul par processus est de **0,15 secondes**. Ce qui veut dire que les communications prennent environ 78% du temps global!
5.3 – Temps de communication

Que comprend le temps que l’on mesure avec MPI_WTIME() ?

- Calcul
- Préparation
- Transfert

- Calcul
- Latence
- Surcoût
- Transfert

☞ Latence : temps d’initialisation des paramètres réseaux.
☞ Surcoût : temps de préparation du message ; caractéristique liée à l’implémentation MPI et au mode de transfert.
5 – Optimisations : quelques définitions

5.4 – Quelques définitions

1 Recopie temporaire d’un message. C’est la copie du message dans une mémoire tampon locale (buffer) avant son envoi. Cette opération est prise en charge par le système MPI dans certains cas.

Figure 19 – Recopie temporaire d’un message
2 **Envoi non bloquant avec recopie temporaire, non couplé avec la réception.** L’appel à un sous-programme de ce type retourne au programme appelant même quand la réception n’a pas été postée. La recopie temporaire des messages est l’un des moyens d’implémenter un envoi non bloquant afin de découpler l’envoi de la réception.

![Diagram](image)

**Figure 20** – Envoi non bloquant avec recopie temporaire du message
3 *Envoi bloquant sans recopie temporaire, couplé avec la réception.* Le message ne quitte le processus émetteur que lorsque le processus récepteur est prêt à le recevoir.

**Figure 21** – Envoi bloquant couplé avec la réception
**5 – Optimisations : quelques définitions**

4 *Envoi non bloquant sans recopie temporaire, couplé avec la réception.* L’appel à ce type de sous-programmes retourne immédiatement au programme appelant bien que l’envoi effectif du message reste couplé avec la réception. Il est donc à la charge du programmeur de s’assurer que le message est bien arrivé à sa destination finale avant de pouvoir modifier les données envoyées.

![Diagram](attachment:image.png)

**Figure 22 – Envoi non bloquant couplé avec la réception**
5.5 – Que fournit MPI ?

Avec MPI l’envoi d’un message peut se faire suivant différents modes :

1. **standard** : il est à la charge de MPI d’effectuer ou non une recopie temporaire du message. Si c’est le cas, l’envoi se termine lorsque la recopie temporaire est achevée (l’envoi est ainsi découpé de la réception). Dans le cas contraire, l’envoi se termine quand la réception du message est achevée.

2. **synchronous** : l’envoi du message ne se termine que si la réception a été postée et la lecture du message terminée. C’est un envoi couplé avec la réception.

3. **buffered** : il est à la charge du programmeur d’effectuer une recopie temporaire du message. L’envoi du message se termine lorsque la recopie temporaire est achevée. L’envoi est ainsi découpé de la réception.

4. **ready** : l’envoi du message ne peut commencer que si la réception a été postée auparavant (ce mode est intéressant pour les applications clients-serveurs).
À titre indicatif voici les différents cas envisagés par la norme sachant que les implémentations peuvent être différentes :

<table>
<thead>
<tr>
<th>modes</th>
<th>bloquant</th>
<th>non-bloquant</th>
</tr>
</thead>
<tbody>
<tr>
<td>envoi <em>standard</em></td>
<td><code>MPI_SEND()</code>&lt;sup&gt;a&lt;/sup&gt;</td>
<td><code>MPI_ISEND()</code></td>
</tr>
<tr>
<td>envoi <em>synchronous</em></td>
<td><code>MPI_SSEND()</code></td>
<td><code>MPI_ISSEND()</code></td>
</tr>
<tr>
<td>envoi <em>buffered</em></td>
<td><code>MPI_BSEND()</code></td>
<td><code>MPI_IBSEND()</code></td>
</tr>
<tr>
<td>réception</td>
<td><code>MPI_RECV()</code></td>
<td><code>MPI_I_RECV()</code></td>
</tr>
</tbody>
</table>

Remarque : pour une implémentation MPI donnée, un envoi standard peut-être bloquant avec recopie temporaire ou bien synchrone avec la réception, ou bien l’un ou l’autre suivant la taille du message à envoyer.

---

<sup>a</sup> Voir la remarque.
5.6 – Envoi synchrone bloquant

Ce mode d’envoi (MPI_SSEND()) de messages permet d’éviter la recopie temporaire des messages et, par conséquent, les surcoûts que cela peut engendrer.

Dans le programme modèle, il suffit de remplacer MPI_SEND() par MPI_SSEND() pour gagner un facteur 2!
temps_debut = MPI_WTIME()
if (rang == 0) then
    ! Envoi d'un gros message
    call MPI_SSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
    ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
    call sgetrf(na, na, a, na, pivota, code)
    ! Calcul modifiant le contenu du tableau C
    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
elseif (rang == 1) then
    ! Calcul (factorisation LU avec LAPACK)
    call sgetrf(na, na, a, na, pivota, code)
    ! Reception d'un gros message
    call MPI_RECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,code)
    ! Calcul dependant de la reception du message precedent
    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
    ! Calcul independant de la reception du message precedent
    call sgetrf(nb, nb, b, nb, pivotb, code)
end if
temps_fin = (MPI_WTIME() - temps_debut)

> mpiexec -n 2 AOptimiser
Temps : 0.36 secondes
5 – Optimisations : synchrone non-bloquant

5.7 – Envoi synchrone non-bloquant

L’utilisation des sous-programmes `MPI_ISSEND()` et `MPI_IRecv()` conjointement aux sous-programmes de synchronisation précédents permet principalement de recouvrir les communications par des calculs.

Le programme modèle, une fois modifié, donne des gains en performances atteignant environ un facteur 3 par rapport à la version initiale!
program AOptimiser
  use mpi
  implicit none

  integer, parameter :: na=256, nb=200
  integer, parameter :: m=2048, etiquette=1111
  real, dimension(na,na) :: a
  real, dimension(nb,nb) :: b
  real, dimension(na) :: pivota
  real, dimension(nb) :: pivotb
  real, dimension(m,m) :: c
  integer :: rang, code, info, requete0, requete1
  real(kind=8) :: temps_debut, temps_fin, temps_fin_max
  integer, dimension(MPI_STATUS_SIZE) :: statut

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

  ! Initialisation des tableaux
  call random_number(a)
  call random_number(b)
  call random_number(c)
temps_debut = MPI_WTIME()
if (rang == 0) then
    ! Requête d’envoi d’un gros message
    call MPI_IsSend(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
    ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
    call sgetrf(na, na, a, na, pivota, code)
    call MPI_Wait(requete0,statut,code)
    ! Calcul modifiant le contenu du tableau C
    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
elseif (rang == 1) then
    ! Calcul (factorisation LU avec LAPACK)
    call sgetrf(na, na, a, na, pivota, code)
    ! Requête de réception d’un gros message
    call MPI_Recv(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
    ! Calcul indépendant de la réception du message précédent, recouvrant celle-ci
    call sgetrf(nb, nb, b, nb, pivotb, code)
    call MPI_Wait(requete1,statut,code)
    ! Calcul dépendant de la réception du message précédent
    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
end if
temps_fin = (MPI_WTIME() - temps_debut)

> mpiexec -n 2 AOtimiser
Temps :  0.23 secondes

calcul  communication
En général, dans le cas d’un envoi (MPI_IxSEND()) ou d’une réception (MPI_IRECV()) non bloquant, il existe toute une palette de sous-programmes qui permettent :

☞ de synchroniser un processus (ex. MPI_WAIT()) jusqu’à terminaison de la requête ;
☞ ou de vérifier (ex. MPI_TEST()) si une requête est bien terminée ;
☞ ou encore de contrôler avant réception (ex. MPI_PROBE() ou MPI_IPROBE()) si un message particulier est bien arrivé.
5.8 – Conseils 1

☞ Éviter si possible la recopie temporaire des messages en utilisant le sous-programme `MPI_SSEND()`.

☞ Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages en utilisant les sous-programmes non bloquants `MPI_ISSEND()` et `MPI_Irecv()`.
Figure 23 – Latences des communications point à point sur IBM SP4 et NEC SX-8 (valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures)
Débits des communications point à point intra-noeud sur la machine IBM SP4 (MP_EAGER_LIMIT=4Ko, valeur par défaut)

Débits des communications point à point intra-noeud sur la machine IBM SP4 (MP_EAGER_LIMIT=64Ko)

(a) MP_EAGER_LIMIT = 4 Ko (valeur par défaut)
(b) MP_EAGER_LIMIT = 64 Ko

**Figure 24** – Débits intra-nœud sur IBM SP4 (*valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures*)
Débits des communications point à point intra-noeud sur la machine NEC SX-8 (messages courts < 512 Ko)

(a) Messages courts

Débits des communications point à point intra-noeud sur la machine NEC SX-8 (512 Ko < messages longs < 64 Mo)

(b) Messages longs

Figure 25 – Débits intra-noeud sur NEC SX-8 (*valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures*)
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.

<table>
<thead>
<tr>
<th>Type d’envoi</th>
<th>Fonction</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>MPI_SEND_INIT()</td>
</tr>
<tr>
<td>synchronoue</td>
<td>MPI_SSEND_INIT()</td>
</tr>
<tr>
<td>buffered</td>
<td>MPI_BSEND_INIT()</td>
</tr>
<tr>
<td>standard de réception</td>
<td>MPI_RECV_INIT()</td>
</tr>
</tbody>
</table>

Reprenons le programme modèle...
if (rang == 0) then
  do k = 1, 1000
    ! Requête d’envoi d’un gros message
    call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
    ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
    call sgetrf(na, na, a, na, pivota, code)
    call MPI_WAIT(requete0,statut,code)
    ! Calcul modifiant le contenu du tableau C
    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
  end do
elseif (rang == 1) then
  do k = 1, 1000
    ! Calcul (factorisation LU avec LAPACK)
    call sgetrf(na, na, a, na, pivota, code)
    ! Requête de réception d’un gros message
    call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
    ! Calcul indépendant de la réception du message précédent, recouvrant celle-ci
    call sgetrf(nb, nb, b, nb, pivotb, code)
    call MPI_WAIT(requete1,statut,code)
    ! Calcul dépendant de la réception du message précédent
    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
  end do
end if
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 considérable lorsque ce mode de communication est réellement implémenté.
if (rang == 0) then
  call MPI_SSEND_INIT(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
do k = 1, 1000
    ! Requête d'envoi d'un gros message
    call MPI_START(requete0,code)
    ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
    call sgetrf(na, na, a, na, pivota, code)
    call MPI_WAIT(requete0,statut,code)
    ! Calcul modifiant le contenu du tableau C
    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
  end do
elseif (rang == 1) then
  call MPI_RECV_INIT(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
do k = 1, 1000
    ! Calcul (factorisation LU avec LAPACK)
    call sgetrf(na, na, a, na, pivota, code)
    ! Requête de réception d'un gros message
    call MPI_START(requete1,code)
    ! Calcul indépendant de la réception du message précédent, recouvrant celle-ci
    call sgetrf(nb, nb, b, nb, pivotb, code)
    call MPI_WAIT(requete1,statut,code)
    ! Calcul dépendant de la réception du message précédent
    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
  end do
end if
Ici, l’implémentation MPI-1 et/ou l’infrastructure matérielle de la machine ne permettent 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_fffff_INIT()` est équivalente à une communication non bloquante `MPI_Ifffff()`.

☞ 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(requeste,code)`.`

☞ Ce sous-programme ne libèrera la requête `requete` qu’une fois que la communication associée sera réellement terminée.
5.10 – Conseils 2

Minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication en utilisant une fois pour toutes un schéma persistant de communication et activer celui-ci autant de fois qu’il est nécessaire à l’aide du sous-programme `MPI_START()`.

Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages car un schéma persistant (ex. `MPI_SSEND_INIT()`) est forcément activé d’une façon non bloquante à l’appel du sous-programme `MPI_START()`.
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_CREATE_HVECTOR()`.

À chaque fois que l’on crée un type de données, il faut le valider à l’aide du sous-programme `MPI_TYPE_COMMIT()`.

Si on souhaite réutiliser le même type, on doit le libérer avec le sous-programme `MPI_TYPE_FREE()`.
Figure 26 – Hiérarchie des constructeurs de type MPI
6 – Types de données dérivés : contigus

6.2 – Types contigus

**MPI_TYPE_CONTIGUOUS()** crée une structure de données à partir d’un ensemble homogène de type prédéfini de données contiguës en mémoire.

```
1.  6.  11.  16.  21.  26.
2.  7.  12.  17.  22.  27.
3.  8.  13.  18.  23.  28.
5. 10.  15.  20.  25.  30.
```

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

**Figure 27 – Sous-programme MPI_TYPE_CONTIGUOUS**

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

call **MPI_TYPE_CONTIGUOUS**(nombre, ancien_type, nouveau_type, code)
```
6.3 – Types avec un pas constant

MPI_TYPE_VECTOR() crée une structure de données à partir d’un ensemble homogène de type prédéfini de données distantes d’un pas constant en mémoire. Le pas est donné en nombre d’éléments.

<table>
<thead>
<tr>
<th></th>
<th>1.</th>
<th>6.</th>
<th>11.</th>
<th>16.</th>
<th>21.</th>
<th>26.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>7.</td>
<td>12.</td>
<td>17.</td>
<td>22.</td>
<td>27.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>8.</td>
<td>13.</td>
<td>18.</td>
<td>23.</td>
<td>28.</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>10.</td>
<td>15.</td>
<td>20.</td>
<td>25.</td>
<td>30.</td>
<td></td>
</tr>
</tbody>
</table>

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

Figure 28 – 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)
MPI_TYPE_CREATE_HVECTOR() crée une structure de données à partir d’un ensemble homogène de type prédéfini 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 vus précédemment, parce qu’alors le pas ne peut plus alors être exprimé en nombre d’éléments du type générique.

```fortran
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)
```
Il est nécessaire de valider tout nouveau type de données dérivé à l’aide du sous-programme **MPI_TYPE_COMMIT()**.

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

call MPI_TYPE_COMMIT(nouveau_type,code)
```

La libération d’un type de données dérivé se fait par le sous-programme **MPI_TYPE_FREE()**.

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

call MPI_TYPE_FREE(nouveau_type,code)
```
program colonne
  use mpi
  implicit none

  integer, parameter :: nb_lignes=5, nb_colonnes=6
  integer, parameter :: etiquette=100
  real, dimension(nb_lignes,nb_colonnes) :: a
  integer, dimension(MPI_STATUS_SIZE) :: statut
  integer :: rang, code, type_colonne

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

  ! Initialisation de la matrice sur chaque processus
  a(:,:) = real(rang)

  ! Définition du type type_colonne
  call MPI_TYPE_CONTIGUOUS(nb_lignes, MPI_REAL, type_colonne, code)

  ! Validation du type type_colonne
  call MPI_TYPE_COMMIT(type_colonne, code)
! Envoi de la première colonne
if ( rang == 0 ) then
   call MPI_SEND(a(1,1),1,type_colonne,1,etiquette,MPI_COMM_WORLD,code)

! Réception dans la dernière colonne
elseif ( rang == 1 ) then
   call MPI_RECV(a(1,nb_colonnes),1,type_colonne,0,etiquette,&
    MPI_COMM_WORLD,statut,code)
end if

! Libère le type
call MPI_TYPE_FREE(type_colonne,code)

call MPI_FINALIZE(code)
end program colonne
Le type « ligne d’une matrice »

program ligne
use mpi
implicit none

integer, parameter :: nb_lignes=5,nb_colonnes=6
integer, parameter :: etiquette=100
real, dimension(nb_lignes,nb_colonnes):: a
integer, dimension(MPI_STATUS_SIZE) :: statut
integer :: rang,code,type_ligne

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

! Initialisation de la matrice sur chaque processus
a(:,:) = real(rang)

! Définition du type type_ligne
call MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)

! Validation du type type_ligne
call MPI_TYPE_COMMIT(type_ligne,code)
! Envoi de la deuxième ligne
if ( rang == 0 ) then
  call MPI_SEND(a(2,1),1,type_ligne,1,étiquette,MPI_COMM_WORLD,code)

! Réception dans l'avant-dernière ligne
elseif ( rang == 1 ) then
  call MPI_RECV(a(nb_lignes-1,1),1,type_ligne,0,étiquette,&
                MPI_COMM_WORLD,statut,code)
end if

! Libère le type type_ligne
call MPI_TYPE_FREE(type_ligne,code)

call MPI_FINALIZE(code)
end program ligne
program bloc
    use mpi
    implicit none

    integer, parameter :: nb_lignes=5,nb_colonnes=6
    integer, parameter :: etiquette=100
    integer, parameter :: nb_lignes_bloc=2,nb_colonnes_bloc=3
    real, dimension(nb_lignes,nb_colonnes):: a
    integer, dimension(MPI_STATUS_SIZE) :: statut
    integer :: rang,code,type_bloc

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

    ! Initialisation de la matrice sur chaque processus
    a(:,:,)= real(rang)

    ! Création du type type_bloc
    call MPI_TYPE_VECTOR(nb_colonnes_bloc,nb_lignes_bloc,nb_lignes,&
                         MPI_REAL,type_bloc,code)

    ! Validation du type type_bloc
    call MPI_TYPE_COMMIT(type_bloc,code)
! Envoi d’un bloc
if ( rang == 0 ) then
  call MPI_SEND(a(1,1),1,type_bloc,1,etiquette,MPI_COMM_WORLD,code)

! Réception du bloc
elseif ( rang == 1 ) then
  call MPI_RECV(a(nb_lignes-1,nb_colonnes-2),1,type_bloc,0,etiquette,&
                MPI_COMM_WORLD,statut,code)
end if

! Libération du type type_bloc
call MPI_TYPE_FREE(type_bloc,code)

call MPI_FINALIZE(code)

end program bloc
6.6 – 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)

ancien_type

nouveau_type

**Figure 29 – Le constructeur MPI_TYPE_INDEXED**

```plaintext
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)

ancien_type

nouveau_type

Figure 30 – 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) :: déplacements
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)
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*.
### Figure 31 – Échanges entre les 2 processus

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

<table>
<thead>
<tr>
<th>Processus 0</th>
<th>Processus 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 9 -17 -25 -33 -41 -49 -57</td>
<td>-1 9 -17 -25 -33 -41 -49 -57</td>
</tr>
<tr>
<td>-4 12 -20 -28 -36 -44 -52 -60</td>
<td>18 35 44 -28 -36 -44 -52 -60</td>
</tr>
<tr>
<td>-6 14 -22 -30 -38 -46 -54 -62</td>
<td>26 41 49 53 58 -46 -54 -62</td>
</tr>
<tr>
<td>-8 16 -24 -32 -40 -48 -56 -64</td>
<td>33 43 51 57 60 62 63 -64</td>
</tr>
</tbody>
</table>
program triangle
  use mpi
  implicit none

  integer,parameter :: n=8,etiquette=100
  real,dimension(n,n) :: a
  integer,dimension(MPI_STATUS_SIZE) :: statut
  integer :: i,code
  integer :: rang,type_triangle
  integer,dimension(n) :: longueurs_blocs,deplacements

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

  ! Initialisation de la matrice sur chaque processus
  a(:,:,)= reshape( (/ (sign(i,-rang),i=1,n*n) /), (/n,n/))

  ! Création du type matrice triangulaire sup pour le processus 0
  ! et du type matrice triangulaire inférieure pour le processus1
  if (rang == 0) then
    longueurs_blocs(:) = (/ (i-1,i=1,n) /)
    deplacements(:) = (/ (n*(i-1),i=1,n) /)
  else
    longueurs_blocs(:) = (/ (n-i,i=1,n) /)
    deplacements(:) = (/ (n*(i-1)+i,i=1,n) /)
  endif
6 – Types de données dérivés : homogènes à pas var.

```fortran
! Permutation des matrices triangulaires supérieure et inférieure
call MPI_SENDRECV_REPLACE(a, 1, type_triangle, mod(rang+1, 2), etiquette, mod(rang+1, 2), &
                        etiquette, MPI_COMM_WORLD, statut, code)

! Libération du type triangle
call MPI_TYPE_FREE(type_triangle, code)

end program triangle
```
6.7 – Types hétérogènes

Le sous-programme `MPI_TYPE_CREATE_STRUCT()` est le constructeur de types le plus général.

Il a les mêmes fonctionnalités que `MPI_TYPE_INDEXED()` mais permet en plus la réplication de blocs de données de types différents.

Les paramètres de `MPI_TYPE_CREATE_STRUCT()` sont les mêmes que ceux de `MPI_TYPE_INDEXED()` avec en plus :

- le champ `anciens_types` est maintenant un vecteur de types de données `MPI` ;
- compte tenu de l’hétérogénéité des données et de leur alignement en mémoire, le calcul du déplacement entre deux éléments repose sur la différence de leurs adresses ;

`MPI`, via `MPI_GET_ADDRESS()`, fournit un sous-programme portable qui permet de retourner l’adresse d’une variable.
nb=5, longueurs_blocs=(3,1,5,1,1), déplacements=(0,7,11,21,26),
anciens_types=(type1,type2,type3,type1,type3)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure32.png}
\caption{Le constructeur MPI\_TYPE\_CREATE\_STRUCT}
\end{figure}

\begin{Verbatim}
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
integer(kind=\texttt{MPI\_ADDRESS\_KIND}),intent(in),dimension(nb) :: déplacements
integer,intent(in),dimension(nb) :: anciens_types

integer, intent(out) :: nouveau_type,code

call \texttt{MPI\_TYPE\_CREATE\_STRUCT}(nb,longueurs_blocs,deplacements,
anciens_types,nouveau_type,code)
\end{Verbatim}

\begin{Verbatim}
<\texttt{type}>,intent(in) :: variable
integer(kind=\texttt{MPI\_ADDRESS\_KIND}),intent(out) :: adresse_variable
integer,intent(out) :: code

call \texttt{MPI\_GET\_ADDRESS}(variable,adresse_variable,code)
\end{Verbatim}
program Interaction_Particules
    use mpi
    implicit none

    integer, parameter :: n=1000,etiquette=100
    integer, dimension(MPI_STATUS_SIZE) :: statut
    integer :: rang,code,type_particule,i
    integer, dimension(4) :: types,longueurs_blocs
    integer(kind=MPI_ADDRESS_KIND), dimension(4) :: deplacements,adresses

    type Particule
        character(len=5) :: categorie
        integer :: masse
        real, dimension(3) :: coords
        logical :: classe
    end type Particule

    type(Particule), dimension(n) :: p,temp_p

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

    ! Construction du type de données
    types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
    longueurs_blocs = (/5,1,3,1/)
6 – Types de données dérivés : hétérogènes

```fortran
! Calcul des déplacements relatifs à l'adresse de départ
do i=1,4
  deplacements(i)=adresses(i) - adresses(1)
end do

! Validation du type structuré
call MPI_TYPE_CREATE_STRUCT(4,longueurs_blocs,deplacements,types,type_particule, &
  code)

! Initialisation des particules pour chaque processus
....

! Envoi des particules de 0 vers 1
if (rang == 0) then
  call MPI_SEND(p(1)%categorie,n,type_particule,1,etiquette,MPI_COMM_WORLD,code)
else
  call MPI_RECV(temp_p(1)%categorie,n,type_particule,0,etiquette,MPI_COMM_WORLD, &
  statut,code)
endif

! Libération du type
call MPI_TYPE_FREE(type_particule,code)
call MPI_FINALIZE(code)
end program Interaction_Particules
```
6.8 – Sous-programmes annexes

La taille totale d’un type de données : \texttt{MPI\_TYPE\_SIZE()}

\begin{verbatim}
integer, intent(in) :: type_donnee
integer, intent(out) :: taille, code

call MPI\_TYPE\_SIZE(type_donnee,taille,code)
\end{verbatim}

La taille ainsi que la borne inférieure d’un type dérivé, \textit{en tenant compte des éventuels alignements mémoire} : \texttt{MPI\_TYPE\_GET\_EXTENT()}

\begin{verbatim}
integer, intent(in) :: type_derive
integer(kind=\texttt{MPI\_ADDRESS\_KIND}),intent(out):: borne_inf_alignee,taille_alignee
integer, intent(out) :: code

call MPI\_TYPE\_GET\_EXTENT(type_derive,borne_inf_alignee,taille_alignee,code)
\end{verbatim}

On peut modifier la borne inférieure d’un type dérivé (et par voie de conséquence sa taille totale) pour créer un nouveau type adapté du précédent

\begin{verbatim}
integer, intent(in) :: ancien_type
integer(kind=\texttt{MPI\_ADDRESS\_KIND}),intent(in) :: nouvelle_borne_inf,nouvelle_taille
integer, intent(out) :: nouveau_type,code

call MPI\_TYPE\_CREATE\_RESIZED(ancien_type,nouvelle_borne_inf,nouvelle_taille,
nouveau_type,code)
\end{verbatim}
6.9 – 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 au chapitre suivant) fait de MPI l’outil idéal pour tous les problèmes de décomposition de domaines avec des maillages réguliers ou irréguliers.
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.
7.2 – Topologies de processus

Topologies de type cartésien :

➣ chaque processus est défini dans une grille de processus ;
➣ 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 – Topologies : cartésiennes

7.3 – Topologies cartésiennes

Une topologie cartésienne est définie lorsqu’un ensemble de processus appartenant à un communicateur donné comm_ancien appellent le sous-programme MPI_CART_CREATE().

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

```fortran
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 33 – Topologie cartésienne 2D périodique en y
Exemple sur une grille 3D comportant 4 domaines suivant x, 2 suivant y et 2 suivant z, non périodique.

```fortran
use mpi
dimension ndims = 3
dimension dims(3)
dimension periods(3)
dimension 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 34 – Topologie cartésienne 3D non périodique
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.

```fortran
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.

<table>
<thead>
<tr>
<th>dims en entrée</th>
<th>call MPI_DIMS_CREATE</th>
<th>dims en sortie</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(8,2,dims,code)</td>
<td>(4,2)</td>
</tr>
<tr>
<td>(0,0,0)</td>
<td>(16,3,dims,code)</td>
<td>(4,2,2)</td>
</tr>
<tr>
<td>(0,4,0)</td>
<td>(16,3,dims,code)</td>
<td>(2,4,2)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>(16,3,dims,code)</td>
<td>error</td>
</tr>
</tbody>
</table>
Dans une topologie cartésienne, le sous-programme `MPI_CART_RANK()` retourne le rang du processus associé aux coordonnées dans la grille.

```fortran
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 35 – Topologie cartésienne 2D périodique en y

```fortran
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
```

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

```fortran
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)
```
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).
Dans une topologie cartésienne, un processus appelant le sous-programme \texttt{MPI\_CART\_SHIFT()} se voit retourner le rang de ses processus voisins dans une direction donnée.

\begin{verbatim}
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)
\end{verbatim}

Le paramètre \texttt{direction} correspond à l’axe du déplacement (xyz).

Le paramètre \texttt{pas} correspond au pas du déplacement.
**Figure 37 – Appel du sous-programme MPI_CART SHIFT()**

```fortran
call MPI_CART_SHIFT(comm_2D, 0, 1, rang_gauche, rang_droit, code)
```

Pour le processus 2, rang_gauche=0, rang_droit=4

```fortran
call MPI_CART_SHIFT(comm_2D, 1, 1, rang_bas, rang_haut, code)
```

Pour le processus 2, rang_bas=3, rang_haut=3
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
Exemple de programme :

```fortran
program decomposition
  use mpi
  implicit none

  integer :: rang_ds_topo,nb_procs
  integer :: code,comm_2D
  integer, dimension(4) :: voisin
  integer, parameter :: N=1,E=2,S=3,W=4
  integer, parameter :: ndims = 2
  integer, dimension (ndims) :: dims,coords
  logical, dimension (ndims) :: periods
  logical :: reorganisation

  call MPI_INIT(code)

  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

  ! Connaître le nombre de processus suivant x et y
  dims(:) = 0

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

! Création grille 2D périodique en y
periods(1) = .false.
periods(2) = .true.
reorganisation = .false.

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

! Connaître mes coordonnées dans la topologie
call MPI_COMM_RANK(comm_2D, rang_ds_topo, code)
call MPI_CART_COORDS(comm_2D, rang_ds_topo, ndims, coords, code)

! Initialisation du tableau voisin à la valeur MPI_PROC_NULL
voisin(:) = MPI_PROC_NULL

! Recherche de mes voisins Ouest et Est
call MPI_CART_SHIFT(comm_2D, 0, 1, voisin(W), voisin(E), code)

! Recherche de mes voisins Sud et Nord
call MPI_CART_SHIFT(comm_2D, 1, 1, voisin(S), voisin(N), code)

call MPI_FINALIZE(code)

end program decomposition
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 \texttt{MPI\_GRAPH\_CREATE()} permet alors de définir une topologie de type graphe en indiquant les voisins de chaque sous-domaine.

```fortran
integer, intent(in) :: comm_ancien, nb_procs
integer, dimension(nb_procs),intent(in) :: index
integer, dimension(nb_voisins_max),intent(in) :: liste_voisins
logical, intent(in) :: reorganisation
integer, intent(out) :: comm_nouveau, code

call MPI\_GRAPH\_CREATE(comm_ancien, nb_procs, index, liste_voisins, reorganisation, &
comm_nouveau, code)
```
Les tableaux d’entiers `index` et `liste_voisins` permettent de définir la liste des voisins pour chacun des nœuds.

**Figure 39 – Graphe de processus**

```plaintext
index = (/ 1, 5, 8, 11, 14, 16 /)
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 :

```fortran
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 :

```fortran
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)
```
program graphe

use mpi
implicit none

integer :: rang, code, comm_graphe, nb_voisins, i, iteration=0
integer, parameter :: etiquette=100
integer, dimension(6) :: index
integer, dimension(16) :: liste_voisins
integer, allocatable, dimension(:) :: voisins
integer, dimension(MPI_STATUS_SIZE) :: statut

real :: propagation, & ! Propagation du feu
        & ! depuis les voisins
        feu=0., & ! Valeur du feu
        bois=1., & ! Rien n’a encore brûlé
        arret=1. & ! Tout a brûlé si arret <= 0.01

call MPI_INIT(code)

! On définit les voisins de chacune des parcelles
index = (/1, 5, 8, 11, 14, 16 /)
liste_voisins = (/1, 0,5,2,3, 1,3,4, 1,2,4, 3,2,5, 1,4 /)

call MPI_GRAPH_CREATE(MPI_COMM_WORLD, 6, index, liste_voisins, .false., comm_graphe, code)
call MPI_COMM_RANK(comm_graphe, rang, code)
if (rang == 2) feu=1. ! Le feu se déclare arbitrairement sur la parcelle 2

call MPI_GRAPH_NEIGHBORS_COUNT(comm_graphe,rang,nb_voisins,code)
allocate(voisins(nb_voisins)) ! Allocation du tableau voisins
call MPI_GRAPH_NEIGHBORS(comm_graphe,rang,nb_voisins,voisins,code)

do while (arret > 0.01) ! On arrête dès qu’il n’y a plus rien à brûler
    do i=1,nb_voisins
        call MPI_SENDRECV(minval((/1.,feu/)),1,MPI_REAL,voisins(i),etiquette, &
                       propagation, 1,MPI_REAL,voisins(i),etiquette, &
                       comm_graphe,statut,code)
        ! Le feu se développe en local sous l’influence des voisins
        feu=1.2*feu + 0.2*propagation*bois
        bois=bois/(1.+feu) ! On calcule ce qui reste de bois sur la parcelle
    end do

call MPI_ALLREDUCE(bois,arret,1,MPI_REAL,MPI_SUM,comm_graphe,code)

iteration=iteration+1
print '("Itération ",i2," parcelle ",i2," bois=",f5.3"),iteration,rang,bois
call MPI_BARRIER(comm_graphe,code)
if (rang == 0) print '("--")'
end do
deallocate(voisins)
call MPI_FINALIZE(code)
end program graphe
7 – Topologies : graphe de processus

```bash
> 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
--
```
Figure 40 – Définition d’une topologie quelconque via un graphe — Exemple de la propagation d’un feu de forêt
8 – Communicateurs

8.1 – 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 41 – Partitionnement d’un communicateur**
Dans l’exemple qui suit, nous allons :
☞ regrouper d’une part les processus de rang pair et d’autre part les 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 42 – Création/destruction d’un communicateur

```
call MPI_INIT(...)

mpiexec -n 8 CommPairImpair
```

```
call MPI_COMM_CREATE(...)call MPI_BCAST(...)call MPI_COMM_FREE(...)```

```
call MPI_INIT(...)```
C’est l’histoire de la poule et de l’œuf...

☞ On ne peut créer un communicateur qu’à partir d’un autre communicateur
☞ Fort heureusement, cela a été résolu en postulant que la poule existait déjà. En effet, un communicateur est fourni par défaut, dont l’identificateur **MPI_COMM_WORLD** est un entier défini dans les fichiers d’en-tête.
☞ Ce communicateur initial **MPI_COMM_WORLD** est créé pour toute la durée d’exécution du programme à l’appel du sous-programme **MPI_INIT()**
☞ Ce communicateur 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
Dans cet exemple, le processus 2 diffuse un message contenant un vecteur “\(a\)” à tous les processus du communicateur **MPI_COMM_WORLD** (donc de l’application) :

```fortran
program monde
  use mpi
  implicit none

  integer, parameter :: m=16
  integer :: rang_dans_monde, code
  real, dimension(m) :: a

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

  a(:)=0.
  if (rang_dans_monde == 2) a(:)= 2.

  call MPI_BCAST(a,m,MPI_REAL,2,MPI_COMM_WORLD,code)

  call MPI_FINALIZE(code)

end program monde
```
MPICOMM_WORLD

Figure 43 – Le communicateur par défaut

```bash
> mpiexec -n 8 monde
```
Que faire pour que le processus 2 diffuse ce message au sous-ensemble de processus de rang pair, par exemple ?

☞ 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 2 doit envoyer le message est pair ou impair.

☞ La solution est de créer un communicateur regroupant ces processus de sorte que le processus 2 diffuse le message à eux seuls

![Diagram](image)

**Figure 44 – Un nouveau communicateur**
8.3 – Groupes et communicateurs

Un communicateur est constitué :
① d’un groupe, qui est un ensemble ordonné de processus ;
② 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 »)

En pratique, pour construire un communicateur, il existe deux façons de procéder :
① par l’intermédiaire d’un groupe de processus ;
② directement à partir d’un autre communicateur.
Dans la bibliothèque *MPI*, divers sous-programmes existent pour construire des communicateurs : `MPI_CART_CREATE()`*, MPI_CART_SUB()*, `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()`. 
8 – Communicateurs : issu d’un autre

8.4 – Communicateur issu d’un autre

L’utilisation directe des groupes présente dans ce cas 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 le soin à MPI d’ordonner le rang des processus dans ces deux communicateurs;
☞ faire des tests conditionnels lors de l’appel au sous-programme MPI_BCAST() :

```fortran
if (mod(rang_dans_monde,2) == 0) then
    ! Diffusion du message seulement aux processus de rangs pairs
    call MPI_BCAST(a,m,MPI_REAL,rang_ds_pair,comm_pair,code)
else
    ! Diffusion du message seulement aux processus de rangs impairs
    call MPI_BCAST(a,m,MPI_REAL,rang_ds_impair,comm_impair,code)
end if
```

MPI-1 – Version 2.24 – Mars 2009

J. Chergui, I. Dupays, D. Girou, S. Requena, P. Wautelet
Le sous-programme **MPI_COMM_SPLIT()** permet de partitionner un communicateur donné en autant de communicateurs que l’on veut...

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

call MPI_COMM_SPLIT(comm, couleur, clef, nouveau_comm, code)
```

<table>
<thead>
<tr>
<th>processus</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>rang_monde</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>couleur</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>clef</td>
<td>2</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>rang_nv_com</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 45** – Construction de communicateurs avec **MPI_COMM_SPLIT()**

Un processus qui se voit attribuer une couleur égale à la valeur **MPI_UNDEFINED** n’appartiendra qu’à son communicateur initial.
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()`.

### Figure 46 – Construction du communicateur `CommPairsImpairs` avec `MPI_COMM_SPLIT()`

<table>
<thead>
<tr>
<th>processus</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>rang_monde</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>couleur</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>clef</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>rang_pairs_imp</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

En pratique, ceci se met en place très simplement...
program PairsImpairs
    use mpi
    implicit none

    integer, parameter :: m=16
    integer :: clef,couleur,CommPairsImpairs
    integer :: rang_dans_monde,code
    real, dimension(m) :: a

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

    ! Initialisation du vecteur A
    a(:)=0.
    if(rang_dans_monde == 2) a(:)=2.
    if(rang_dans_monde == 5) a(:)=5.
if (mod(rang_dans_monde, 2) == 0) then
    ! Couleur et clefs des processus pairs
    couleur = 0
    if (rang_dans_monde == 2) then
        clef = 0
    else
        clef = rang_dans_monde + 1
    end if
else
    ! Couleur et clefs des processus impairs
    couleur = 1
    if (rang_dans_monde == 5) then
        clef = 0
    else
        clef = rang_dans_monde
    end if
end if

! Création des communicateurs pair et impair en leur donnant une même dénomination
call MPI_COMM_SPLIT(MPI_COMM_WORLD, couleur, clef, CommPairsImpairs, code)

! Diffusion du message par le processus 0 de chaque communicateur aux processus
! de son groupe
call MPI_BCAST(a, m, MPI_REAL, 0, CommPairsImpairs, code)

! Destruction des communicateurs
call MPI_COMM_FREE(CommPairsImpairs, code)
call MPI_FINALIZE(code)
end program PairsImpairs
8.5 – 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 47 – Deux exemples de distribution de données dans une topologie 2D dégénérée
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) :: Subdivision
integer,intent(in) :: CommCart
integer,intent(out) :: CommCartD, code
call MPI_CART_SUB(CommCart,Subdivision,CommCartD,code)
```

**Figure 48** – Représentation initiale d’un tableau $V$ dans la grille 2D et représentation finale après la distribution de celui-ci sur la grille 2D dégénérée.
program CommCartSub
  use mpi
  implicit none

  integer         :: Comm2D,Comm1D,rang,code
  integer,parameter :: NDim2D=2
  integer,dimension(NDim2D) :: Dim2D,Coord2D
  logical,dimension(NDim2D) :: Periode,Subdivision
  logical          :: Reordonne
  integer,parameter :: m=4
  real, dimension(m) :: V(:)=0.
  real              :: W=0.
call MPI_INIT(code)

! Création de la grille 2D initiale
Dim2D(1) = 4
Dim2D(2) = 3
Periode(:) = .false.
ReOrdonne = .false.
call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Periode,ReOrdonne,Comm2D,code)
call MPI_COMM_RANK(Comm2D,rang,code)
call MPI_CART_COORDS(Comm2D,rang,NDim2D,Coord2D,code)

! Initialisation du vecteur $V$
if (Coord2D(1) == 1) $V(:)=\text{real}(\text{rang})$

! Chaque ligne de la grille doit être une topologie cartésienne 1D
Subdivision(1) = .true.
Subdivision(2) = .false.

! Subdivision de la grille cartésienne 2D
call MPI_CART_SUB(Comm2D,Subdivision,Comm1D,code)

! Les processus de la colonne 2 distribuent le vecteur $V$ aux processus de leur ligne
call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,code)

print '("Rang : ",I2," ; Coordonnees : ",I1,"","I1,"); W = ",F2.0")', &
rang,Coord2D(1),Coord2D(2),W
call MPI_FINALIZE(code)
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  =  3.
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.
```
8.6 – Intra et intercommunicateurs

Les communicateurs que nous avons construits jusqu’à présent sont des intracommunicateurs (ex. comm_pair et comm_impair) 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.

Une fois ce lien établi, seules sont possibles les communications point à point, les communications collectives au sein d’un intercommunicateur n’étant pas permises dans MPI-1 (cette limitation a disparu dans MPI-2).

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...
8.7 – Exemple récapitulatif

MPI_COMM_WORLD

IntraComm (ocean)  InterComm  IntraComm (atmosphere)  InterComm  IntraComm (visualisation)

\[
\begin{align*}
a_0^0 & \quad b_1^0 & \quad c_2^0 \\
d_3^1 & \quad e_4^1 & \quad f_5^1 \\
g_6^2 & \quad h_7^2 & \quad \text{OceanAtmos...} & \quad \text{VisuAtmos...} \\
\end{align*}
\]

Figure 49 – Couplage océan/atmosphère
program OceanAtmosphere
  use mpi
  implicit none

  integer, parameter :: tag1 = 1111, tag2 = 2222
  integer :: RangMonde, NombreIntraComm, couleur, code, &
             IntraComm, CommOceanAtmosphere, CommVisuAtmosphere

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

  ! Construction des 3 IntraCommunicateurs
  NombreIntraComm = 3
  couleur = mod(RangMonde, NombreIntraComm) ! = 0, 1, 2
  call MPI_COMM_SPLIT(MPI_COMM_WORLD, couleur, RangMonde, IntraComm, code)
Construction des deux InterCommunicateurs et appel des sous-programmes de calcul

```fortran
select case(couleur)

  case(0)
    ! InterCommunicateur OceanAtmosphere pour que le groupe 0 communique
    ! avec le groupe 1
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,1,tag1,CommOceanAtmosphere, &
                             code)
    call ocean(IntraComm,CommOceanAtmosphere)

  case(1)
    ! InterCommunicateur OceanAtmosphere pour que le groupe 1 communique
    ! avec le groupe 0
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,0,tag1,CommOceanAtmosphere, &
                             code)

    ! InterCommunicateur CommVisuAtmosphere pour que le groupe 1 communique
    ! avec le groupe 2
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,2,tag2,CommVisuAtmosphere,code)
    call atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)

  case(2)
    ! InterCommunicateur CommVisuAtmosphere pour que le groupe 2 communique
    ! avec le groupe 1
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,1,tag2,CommVisuAtmosphere,code)
    call visualisation(IntraComm,CommVisuAtmosphere)

end select
```
subroutine ocean(IntraComm,CommOceanAtmosphere)
    use mpi
    implicit none
    integer,parameter :: n=1024,tag1=3333
    real,dimension(n) :: a,b,c
    integer :: rang,code,germe(1),IntraComm,CommOceanAtmosphere
    integer,dimension(MPI_STATUS_SIZE) :: statut
    integer,intrinsic :: irtc

    ! Les processus 0, 3, 6 dédiés au modèle océanographique effectuent un calcul
    germe(1)=irtc()
    call random_seed(put=germe)
    call random_number(a)
    call random_number(b)
    call random_number(c)
    a(:) = b(:) * c(:)

    ! Les processus impliqués dans le modèle océan effectuent une opération collective
    call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_SUM,IntraComm,code)

    ! Rang du processus dans IntraComm
    call MPI_COMM_RANK(IntraComm,rang,code)

    ! Échange de messages avec les processus associés au modèle atmosphérique
    call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
                               CommOceanAtmosphere,statut,code)

    ! Le modèle océanographique tient compte des valeurs atmosphériques
    a(:) = b(:) * c(:)
end subroutine ocean
subroutine atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)
  use mpi
  implicit none

  integer,parameter :: n=1024,tag1=3333,tag2=4444
  real,dimension(n) :: a,b,c
  integer :: rang,code,germe(1),IntraComm, &
                     CommOceanAtmosphere,CommVisuAtmosphere

  integer,dimension(MPI_STATUS_SIZE) :: statut
  integer,intrinsic :: irtc

! Les processus 1, 4, 7 dédiés au modèle atmosphérique effectuent un calcul
  germe(1)=irtc()
  call random_seed(put=germe)

  call random_number(a)
call random_number(b)
call random_number(c)

  a(:) = b(:) + c(:)
Les processus dédiés au modèle atmosphère effectuent une opération collective

```fortran
! Les processus dédiés au modèle atmosphère effectuent une opération collective
call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_MAX,IntraComm,code)

! Rang du processus dans IntraComm
call MPI_COMM_RANK(IntraComm,rang,code)

! Échange de messages avec les processus dédiés au modèle océanographique
call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
CommOceanAtmosphere,statut,code)

! Le modèle atmosphère tient compte des valeurs océanographiques
a(:, ) = b(:, ) * c(:, )

! Envoi des résultats aux processus dédiés à la visualisation
if (rang == 0 .or. rang == 1) then
    call MPI_SSEND(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,code)
end if

end subroutine atmosphere
```
subroutine visualisation(IntraComm,CommVisuAtmosphere)
  use mpi
  implicit none

  integer,parameter :: n=1024,tag2=4444
  real,dimension(n) :: a,b,c
  integer :: rang,code,IntraComm,CommVisuAtmosphere
  integer,dimension(MPI_STATUS_SIZE) :: statut

  ! Les processus 2 et 5 sont chargés de la visualisation
  call MPI_COMM_RANK(IntraComm,rang,code)

  ! Réception des valeurs du champ à tracer
  call MPI_RECV(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,statut,code)

  print*,'Moi, processus ','rang,' je trace mon champ A : ',a(:)

end subroutine visualisation
8.8 – Conclusion

☞ **Groupes et contextes** définissent un objet appelé *communicateur*.
☞ Les communicateurs définissent la portée des communications.
☞ Ils sont utilisés pour dissocier les espaces de communication.
☞ Un communicateur doit être spécifié à l’appel de toute fonction d’échange de messages.
☞ Ils permettent d’éviter les confusions lors de la sélection des messages, par exemple au moment de l’appel à un sous-programme d’une bibliothèque scientifique qui elle-même effectue des échanges de messages.
☞ Les communicateurs offrent une programmation modulaire du point de vue de l’espace de communication. Dans le cadre de projets importants, chaque équipe développe son module sans se soucier du choix du communicateur des autres équipes (exemple : modèle couplé océan/atmosphère).
Historique :

- début des travaux en mars 1995 ;
- brouillon présenté pour SuperComputing 96 ;
- version « officielle » disponible en juillet 1997 ;
- voir http://www.erc.msstate.edu/mpi/mpi2.html

Principaux domaines nouveaux :

- gestion dynamique des processus :
  - possibilité de développer des codes MPMD ;
  - support multi plates-formes ;
  - démarrage et arrêt dynamique de sous-tâches ;
  - gestion de signaux système .
- communications de mémoire à mémoire ;
- entrées/sorties parallèles.
Autres domaines où apparaissent des améliorations :

- extensions concernant les intracommunicateurs ;
- extensions concernant les intercommunicateurs ;
- divers autres apports :
  - inter-opérabilité entre C et Fortran ;
  - interfaçage avec C++ et Fortran 90 (avec des limitations dans ce dernier cas).

Extensions proposées en dehors de MPI-2 : IMPI (*Interoperable MPI*), MPI-RT (*real time extensions*)
<table>
<thead>
<tr>
<th>Concept</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrière</td>
<td>37</td>
</tr>
<tr>
<td>bloquantes (communications)</td>
<td>23, 29, 31, 67, 69, 74, 77, 78, 89</td>
</tr>
<tr>
<td>non-bloquantes (communications)</td>
<td>88</td>
</tr>
<tr>
<td>collectives (communications)</td>
<td>12</td>
</tr>
<tr>
<td>communicateur</td>
<td>12, 19, 20, 23, 35, 117, 139, 141–144, 146–149, 152, 166</td>
</tr>
<tr>
<td>intercommunicateur</td>
<td>158</td>
</tr>
<tr>
<td>intracommunicateur</td>
<td>158</td>
</tr>
<tr>
<td>communication</td>
<td>12, 61, 64, 65, 67–69, 72, 74, 77–79, 81–83, 88, 89, 139, 141, 146, 158, 166</td>
</tr>
<tr>
<td>contexte de communication</td>
<td>23, 146, 166</td>
</tr>
<tr>
<td>envoi</td>
<td>67, 68, 70–72, 77, 82, 83</td>
</tr>
<tr>
<td>étiquette</td>
<td>23, 35</td>
</tr>
<tr>
<td>groupe</td>
<td>51, 146, 147, 166</td>
</tr>
<tr>
<td>intercommunicateur</td>
<td>158</td>
</tr>
<tr>
<td>intracommunicateur</td>
<td>158</td>
</tr>
<tr>
<td>message</td>
<td>7, 10–12, 66, 69, 70, 72, 78, 82, 142, 144, 166</td>
</tr>
<tr>
<td>MPMD</td>
<td>8, 9</td>
</tr>
</tbody>
</table>
optimisation ................................................................. 61
performances ................................................................. 147
persistantes (communications) ........................................... 82, 83, 88, 89
portabilité ................................................................. 16, 102, 114
processeur ................................................................. 6
processus ... 7, 8, 10–12, 19, 20, 22, 23, 27, 31, 32, 35, 51, 60, 114–117, 122, 123, 125, 127, 139, 142, 144, 146–148, 158
rang ................................................................. 20, 23, 27, 123, 125, 127, 147, 148
réception ................................................................. 67, 68, 70, 71, 77, 82, 83
requête ................................................................. 83, 88
SPMD ................................................................. 8, 9
surcoût ................................................................. 72, 89
topologie ................................................................. 12, 114–117, 122, 123, 125, 127, 152, 154
types derivés ................................................................. 114
### Index des constantes MPI

<table>
<thead>
<tr>
<th>Constante</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi</td>
<td>18</td>
</tr>
<tr>
<td>mpi.h</td>
<td>18</td>
</tr>
<tr>
<td>MPI_ADDRESS_KIND</td>
<td>94, 104, 110, 111, 113</td>
</tr>
<tr>
<td>MPI_ANY_SOURCE</td>
<td>27</td>
</tr>
<tr>
<td>MPI_ANY_TAG</td>
<td>27</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>111</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>58, 90</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>63</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>24, 28, 29, 31, 33, 39, 54, 56, 90, 94, 102, 111</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>111</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>63, 164</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>27, 131</td>
</tr>
<tr>
<td>Constant</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>56</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>41, 44, 46, 49, 63, 73, 76, 84, 86, 90, 92–94, 96, 98, 100, 102, 108, 111, 136, 142, 147, 151, 156, 162, 164, 165</td>
</tr>
<tr>
<td>MPI_STATUS_IGNORE</td>
<td>27</td>
</tr>
<tr>
<td>MPI_STATUS_SIZE</td>
<td>24, 28, 33, 62, 75, 96, 98, 100, 107, 111, 135, 162, 163, 165</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>54, 136, 162</td>
</tr>
<tr>
<td>MPI_UNDEFINED</td>
<td>148</td>
</tr>
<tr>
<td>mpif.h</td>
<td>18</td>
</tr>
<tr>
<td>Sous-programme</td>
<td>Pages</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>MPI_ALLGATHER</td>
<td>36, 46, 60</td>
</tr>
<tr>
<td>MPI_ALLGATHERV</td>
<td>60</td>
</tr>
<tr>
<td>MPI_ALLREDUCE</td>
<td>36, 51, 56, 136, 162, 164</td>
</tr>
<tr>
<td>MPI_ALLTOALL</td>
<td>36, 49, 60</td>
</tr>
<tr>
<td>MPI_ALLTOALLV</td>
<td>60</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>36, 37, 136</td>
</tr>
<tr>
<td>MPI_BCAST</td>
<td>36, 39, 51, 142, 147, 151</td>
</tr>
<tr>
<td>MPI_CART_COORDS</td>
<td>125, 126, 131, 156</td>
</tr>
<tr>
<td>MPI_CART_CREATE</td>
<td>117, 118, 120, 131, 146, 156</td>
</tr>
<tr>
<td>MPI_CART_RANK</td>
<td>123, 124</td>
</tr>
<tr>
<td>MPI_CART_SHIFT</td>
<td>127–129, 131</td>
</tr>
<tr>
<td>MPI_CART_SUB</td>
<td>146, 154, 156</td>
</tr>
<tr>
<td>MPI_COMM_CREATE</td>
<td>146</td>
</tr>
<tr>
<td>MPI_COMM_DUP</td>
<td>146</td>
</tr>
<tr>
<td>MPI_COMM_FREE</td>
<td>146, 151</td>
</tr>
<tr>
<td>MPI_COMM_RANK</td>
<td>20, 21, 24, 28, 33, 39, 41, 44, 46, 49, 54, 56, 58, 62, 75, 96, 98, 100, 107, 111, 131, 135, 142, 150, 156, 160, 162, 164, 165</td>
</tr>
<tr>
<td>MPI_COMM_SIZE</td>
<td>20, 21, 33, 41, 44, 46, 49, 54, 56, 130</td>
</tr>
<tr>
<td>MPI_COMM_SPLIT</td>
<td>146, 148, 149, 151, 154, 160</td>
</tr>
<tr>
<td>Sous-programme</td>
<td>Pages</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>MPI_DIMS_CREATE</td>
<td>122, 130</td>
</tr>
<tr>
<td>MPI_FINALIZE</td>
<td>18, 21, 24, 28, 33, 39, 41, 44, 46, 49, 54, 56, 58, 63, 97, 99, 101, 108, 112, 131, 136, 141, 142, 151, 156</td>
</tr>
<tr>
<td>MPI_GATHER</td>
<td>36, 44, 60</td>
</tr>
<tr>
<td>MPI_GATHERV</td>
<td>60</td>
</tr>
<tr>
<td>MPI_GET_ADDRESS</td>
<td>109, 110, 112</td>
</tr>
<tr>
<td>MPI_GRAPH_CREATE</td>
<td>132, 135</td>
</tr>
<tr>
<td>MPI_GRAPH_NEIGHBORS</td>
<td>134, 136</td>
</tr>
<tr>
<td>MPI_GRAPH_NEIGHBORS_COUNT</td>
<td>134, 136</td>
</tr>
<tr>
<td>MPI_INIT</td>
<td>18, 21, 24, 28, 33, 39, 41, 44, 46, 49, 54, 56, 58, 62, 75, 96, 98, 100, 107, 111, 130, 135, 141, 142, 150, 156, 160</td>
</tr>
<tr>
<td>MPI_INTERCOMM_CREATE</td>
<td>158, 161</td>
</tr>
<tr>
<td>MPI_IPROBE</td>
<td>77</td>
</tr>
<tr>
<td>MPI_IRECV</td>
<td>74, 76–78, 84</td>
</tr>
<tr>
<td>MPI_ISSEND</td>
<td>74, 76, 78, 84</td>
</tr>
<tr>
<td>MPI_IxSEND</td>
<td>77</td>
</tr>
<tr>
<td>MPI_OP_CREATE</td>
<td>51, 58</td>
</tr>
<tr>
<td>MPI_OP_FREE</td>
<td>51</td>
</tr>
<tr>
<td>MPI_PROBE</td>
<td>77</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>24, 29, 31, 33, 63, 73, 97, 99, 101, 112, 165</td>
</tr>
</tbody>
</table>
## Index des sous-programmes MPI

<table>
<thead>
<tr>
<th>Sous-programme</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_RECV_INIT</td>
<td>86</td>
</tr>
<tr>
<td>MPI_REDUCE</td>
<td>36, 51, 54, 58, 63</td>
</tr>
<tr>
<td>MPI_REQUEST_FREE</td>
<td>88</td>
</tr>
<tr>
<td>MPI_SCAN</td>
<td>51</td>
</tr>
<tr>
<td>MPI_SCATTER</td>
<td>36, 41, 60, 156</td>
</tr>
<tr>
<td>MPI_SCATTERV</td>
<td>60</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>24, 29, 31, 33, 63, 72, 97, 99, 101, 112</td>
</tr>
<tr>
<td>MPI_SENDRECV</td>
<td>27–29, 114, 136</td>
</tr>
<tr>
<td>MPI_SENDRECV_REPLACE</td>
<td>27, 105, 108, 162, 164</td>
</tr>
<tr>
<td>MPI_SSEND</td>
<td>72, 73, 78, 164</td>
</tr>
<tr>
<td>MPI_SSEND_INIT</td>
<td>86, 89</td>
</tr>
<tr>
<td>MPI_START</td>
<td>86, 88, 89</td>
</tr>
<tr>
<td>MPI_TEST</td>
<td>77</td>
</tr>
<tr>
<td>MPI_TYPE_COMMIT</td>
<td>90, 95, 96, 98, 100, 108, 112</td>
</tr>
<tr>
<td>MPI_TYPE_CONTIGUOUS</td>
<td>27, 90, 92, 96</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_HINDEXED</td>
<td>102, 104</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_HVECTOR</td>
<td>90, 94, 102</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_RESIZED</td>
<td>113</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_STRUCT</td>
<td>109, 110, 112</td>
</tr>
<tr>
<td>Sous-programme</td>
<td>Pages</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>MPI_TYPE_FREE</td>
<td>90, 95, 97, 99, 101, 108, 112</td>
</tr>
<tr>
<td>MPI_TYPE_GET_EXTENT</td>
<td>102, 113</td>
</tr>
<tr>
<td>MPI_TYPE_INDEXED</td>
<td>27, 102, 103, 108, 109</td>
</tr>
<tr>
<td>MPI_TYPE_SIZE</td>
<td>102, 113</td>
</tr>
<tr>
<td>MPI_TYPE_STRUCT</td>
<td>27</td>
</tr>
<tr>
<td>MPI_TYPE_VECTOR</td>
<td>27, 90, 93, 98, 100</td>
</tr>
<tr>
<td>MPI_WAIT</td>
<td>76, 77, 84, 86</td>
</tr>
<tr>
<td>MPI_WTIME</td>
<td>63, 65, 73, 76</td>
</tr>
</tbody>
</table>