Introduction au calcul parallèle avec labibliothèque MPI (Message Passing Interface)

Stephanie Delage Santacreu

Introduction au calcul parallèle avec la bibliothèque MPI (Message Passing Interface)

Stéphanie DELAGE SANTACREU,

Pôle Calcul Scientifique de l'UPPA ($http://sinum.univ-pau.fr$),

CRI rue Jules Ferry, PAU.

Novembre 2008

Table des matières

1 Introduction

Ce cours est issu de la formation faite par l'IDRIS ($http://www.idris.fr$).

Un code de calcul ou programme peut être écrit suivant deux modèles de programmation : séquentiel ou parallèle.
Dans le modèle séquentiel, un programme est exécuté par un unique processus. Ce processus tourne sur un processeur d'une machine et a accès à la mémoire du processeur.
Il arrive que pour certains codes de calcul, la mémoire d'un seul processeur ne suffise plus (manipulation de gros tableaux) et/ou le temps de calcul soit trop important...

Pour palier à ces problèmes, on peut avoir recours à différentes méthodes comme le raffinement de maillage adaptatif, le grid-meshing ou la programmation parallèle.
On s'intéresse ici plus particulièrement à la programmation parallèle. Celle-ci permet de répartir les charges de calcul sur plusieurs processus. Il existe deux types de programmation parallèle : le MPI (Message Passing Interface) et le $openMP$ (Multithreading). On se limitera au MPI.
Dans un modèle de programmation parallèle par échange de messages (MPI), le programme est dupliqué sur plusieurs processus. Chaque processus exécute un exemplaire du programme et a accès à sa mémoire propre. De ce fait, les variables du programme deviennent des variables locales au niveau de chaque processus. De plus un processus ne peut pas accéder à la mémoire des processus voisins. Il peut toutefois envoyer des informations à d'autres processus à condition que ces derniers (processus récepteurs) soient au courant qu'ils devaient recevoir ces informations du processus émetteur.
La communication entre processus se fait uniquement par passage de messages entre processus (c'est à dire : envoi et reception de messages). Techniquement, cette communication se fait via des fonctions de la bibliothèque MPI appelées dans le programme. L'environnement MPI permet de gérer et interpréter ces messages.

2 Environnement MPI

2.1 description

Pour utiliser la bibliothèque MPI, le programme source doit impérativement contenir :

1. l'appel au module MPI : **include mpif.h** en fortran77, **use MPI** en fortran90, **include mpi.h** en $C/C++$.
2. l'initialisation de l'environnement via l'appel à la subroutine **MPI_INIT(code)**. Cette fonction retourne une valeur dans la variable *code*. Si l'initialisation s'est bien passée, la valeur de *code* est égale à celle dans **MPI_SUCCESS**.
3. la désactivation de l'environnement via l'appel à la subroutine **MPI_FINALIZE (code)**. L'oublie de cette subroutine provoque une erreur.

Une fois l'environnement MPI initialisé, on dispose d'un ensemble de processus actifs et d'un espace de communication au sein duquel on va pouvoir effectuer des opérations MPI. Ce couple (processus actifs,espace de communication) est appelé communicateur.
Le communicateur par défaut est **MPI_COMM_WORLD** et comprend tous les processus actifs. Il est initialisé lors de l'appel à la fonction **MPI_INIT()** et désactivé par l'appel à la fonction **MPI_FINALIZE()**.
On peut connaître le nombre de processus actifs gérés par un communicateur avec la fonction **MPI_COMM_SIZE**(*comm*, *nb_procs*, *code*) ainsi que le rang (ou numéro) d'un processus avec la fonction **MPI_COMM_RANK**(*comm*, *rang*, *code*).

. **comm** ($< in >$) est en entier désignant le communicateur,
. **nb_procs** ($< out >$) est en entier donnant le nombre de processus,
. **rang** ($< out >$) est un entier indiquant le rang du processus. Il est important dans le sens où il sert lui d'identifificateur dans un communicateur.
. **code** ($< out >$) est un entier retournant un code d'erreur.

Remarque : Il faut toujours essayer de développer un code parallèle pour un nombre quelconque de processus.

2.2 Du programme source à l'exécution :

Trois étapes sont nécessaires : l'écriture du programme, sa compilation puis son exécution.

1. **L'écriture** : La figure 1 montre un exemple de programme (exercice Idris) en $fortran90$.
2. **La compilation** du programme peut se faire par l'intermédiaire d'un Makefile (figure 2). Les options de compilations dépendent du compilateur. La figure montre un exemple de Makefile pour compiler le programme $pairimpair.f90$ (figure 1) avec le compilateur du constructeur IBM (XL)

3. **L'exécution** du programme en interactif sur 4 processus se fait via la commande :
   mpiexec -n 4 /users/uppa/delage/test/pairimpair.
   Réponse du programme :
   processus de rang pair : 0
   processus de rang impair : 1
   processus de rang pair : 2
   processus de rang impair : 3

```
program pairimpair

!utilisation de la bibliotheque mpi
use mpi

!declarations variables
implicit none
integer::nb_procs !nombre de processus
integer::rang     !rang d'un processus
integer:: code    !retour d'erreur


!initialisation environnement MPI
call MPI_INIT(code)

!Gestion des erreurs d'initialisation MPI
if(code/=MPI_SUCCESS)then
   print*,'erreur MPI',code
   stop
end if


!fonctions MPI : nombre de processus actif du
!commutateur MPI_COMM_WORLD
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

!fonctions MPI : rang processus
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!processus de rang pair et impair
if (mod(rang,2)==0) then
   print*,'processus de rang pair : ', rang
else
   print*,'processus de rang impair : ', rang
end if


!desactivation environnement MPI
call MPI_FINALIZE(code)

end program
```

FIG. 1: Programme (en Fortran90) processus pair et impair.

```
## indique le suffixe des fichiers sources
.SUFFIXES:.f90

## choix du compilateur en séquentiel
G90=mpxlf90

## choix des options de compilation de base
OPTC= -qfree=f90 -qsuffix=f=f90
OPTBIT=-q64
## choix des options de Débogage
ODEBUG=-qnooptimize -qsource
DEBUG= -g -qcheck -qflttrap=:ov:zero:inv

## choix des options d'optimisation
OPT= -O2

## options de profilage
PROF= -pg  -qfullpath -qdbg

## Options de compilation si phase de débogage (décommenter)
FFLAGS=$(OPTC)$(OPTBIT) $(PROF) $(ODEBUG) $(DEBUG)
##Options de compilation avec optimisation (décommenter)
##FFLAGS= $(OPTC) $(OPT) $(OPTBIT) $(PROF)

## fichiers objets compilés (attention mettre des tabulations
## et non des espaces)
OBJ= pairimpair.o

## fichiers sources à compiler  (attention mettre des
## tabulations  et non des espaces)
SRC= pairimpair.f90


## choix bibliothèques : LIBPATH indique chemin et LIBS le nom bibliothèque liblapack
LIBPATH= /usr/local/lapack
LIBS= -llapack

##options editions de liens
LFLAGS= $(OPT) $(OPTBIT) $(PROF)

## nom de l'executable
EXEC=pairimpair

## construction des fichiers objets à partir des fichiers sources
.f90.o :
        $(G90) -c $(FFLAGS) $<

## construction de l'executable
all : $(OBJ)
        $(G90) $(LFLAGS) $(OBJ) -o $(EXEC)

## nettoyage
clean :
        rm -f $(OBJ) $(EXEC)

## dépendances :
pairimpair.o: pairimpair.f90
        $(G90) -c $(FFLAGS) pairimpair.f90
```

FIG. 2: Exemple de Makefile pour le programme *pairimpair.f*90.

3 Communications

3.1 Communications point à point

La communication point à point est une communication entre deux processus. L'un d'eux envoie un message (c'est l'**émetteur**), l'autre le reçoit (c'est le **récepteur**).
Ce message doit contenir un certain nombre d'informations pour assurer une bonne réception et interprétation par le récepteur, à savoir :

. le rang du processus émetteur (*rang_proc_source*),
. le rang du processus récepteur (*rang_proc_dest*),
. l'étiquette du message (*tag_emis* pour message émis, *tag_recu* pour message reçu),
. le nom du communicateur (*comm*),
. le type des données échangées (*type_emis* pour message émis, *type_recu* pour message reçu), voir tableau 1 pour le fortran et tableau 2 pour le C,
. le nom données échangées (*val_emis* pour message émis, *val_recu* pour message reçu).
. la taille des données échangées (*taille_emis* pour message émis, *taille_recu* pour message reçu) (scalaire, vecteur, matrice, ...).

Plusieurs modes de transfert sont possibles pour échanger des messages. On décrit ici des fonctions MPI de communication en mode bloquant, qui laisse la main une fois que le message est bien reçu. C'est le mode à utiliser quand on commence à paralléliser un code. On peut ensuite passer à un autre mode de tranfert pour optimiser le temps de communication (détail plus tard).

1. **MPI_SEND**(*val_emis*, *taille_emis*, *type_emis*, *rang_proc_dest*, *tag_emis*, *comm*, *code*) pour l'envoi du message suivi de **MPI_RECV**(*val_recu*, *taille_recu*, *type_recu*, *rang_proc_dest*, *tag_recu*, *comm*, *statut*, *code*) pour la réception. Quand un message est envoyé, il faut être sûr qu'il a été bien reçu.
   . **val_emis** ($< in >$) : élément envoyé,
   . **taille_emis** ($< in >$) : entier indiquant la taille de l'élément envoyé (scalaire, vecteur, ...),
   . **type_emis** ($< in >$) : type de l'élément envoyé,
   . **rang_proc_dest** ($< in >$) : entier indiquant le rang du processus qui reçoit le message,
   . **tag_emis** ($< in >$) : entier désignant l'étiquette du message,
   . **comm** ($< in >$) : entier désignant le communicateur (MPI_COMM_WORLD par défaut),
   . **code** ($< out >$) : entier donnant un code d'erreur,

. **val_recu** ($< in >$) : élément reçu,
. **taille_recu** ($< in >$) : entier indiquant la taille de l'élément reçu (scalaire, vecteur, ...). Il doit correspondre à celui indiqué dans $taille\_emis$.
. **type_recu** ($< in >$) : type de l'élément reçu. Il doit aussi correspondre à celui indiqué dans $type\_emis$.
. **tag_recu** ($< in >$) :entier désignant l'étiquette du message,
. **statut** ($< out >$) : tableau d'entiers de taille MPI_STATUS_SIZE contenant de nombreuses informations sur le message.

2. **MPI_SENDRECV**(val_emis, $taille\_emis$, $type\_emis$, $rang\_proc\_dest$, tag_emis, val_recu, $taille\_recu$, $type\_recu$, $rang\_proc\_source$, tag_recu, $comm$, $statut$, $code$) pour l'envoi et la reception de messages. Attention, si on utilise la même variable pour l'envoi et la réception ($val\_emis = val\_recu$), il y a écrasement. **rang_proc_source** ($< in >$) est un entier indiquant le rang du processus qui a émis le message
3. **MPI_SENDRECV_REPLACE**(val_emis_recu, $taille\_emis\_recu$, $type\_emis\_recu$, $rang\_proc\_dest$, tag_emis, $rang\_proc\_source$, tag_recu, $comm$, $statut$, $code$) pour l'envoi et la reception de messages en utilisant le même variable val_emis_recu pour l'envoi et la réception. Cette fois-ci il n'y a pas d'écrasement.

La figure 3 propose un exemple de programme en $Fortran90$ d'échange de messages entre deux processus (exercice de l'Idris) :

type MPI	**type Fortran**
$MPI\_INTEGER$	INTEGER
$MPI\_INTEGER8$	INTEGER, kind=8
MPI_REAL	REAL
$MPI\_DOUBLE\_PRECISION$	DOUBLE
$MPI\_COMPLEX$	COMPLEX
$MPI\_LOGICAL$	LOGICAL
$MPI\_CHARACTER$	CHARACTER
MPI_PACKED	Types hétérogènes

TAB. 1: Principaux type de données pour le fortran

Remarques :
– On peut utiliser des "$jokers$" pour le rang du processus ($rang\_proc\_dest = MPI\_ANY\_SOURCE$, ie on reçoit de n'importe

```
program pingpong
!utilisation de la bibliotheque mpi
use mpi

!declarations variables
implicit none
integer::nb_procs,rang !nombre de processus,rang d'un processus
integer:: code    !retour d'erreur
integer,parameter:: tag=100, nb_val=5!taille message, etiquette
integer,dimension(MPI_STATUS_SIZE)::statut
real*8::temps_debut,temps_fin
real*8,dimension(nb_val)::val

!initialisation environnement MPI
call MPI_INIT(code)

!Gestion des erreurs d'initialisation MPI
if(code/=MPI_SUCCESS)then
   print*,'erreur MPI',code
   stop
end if

!fonctions MPI : nombre de processus actif du commutateur
!MPI_COMM_WORLD
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

!fonctions MPI : rang processus
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!pingpong : echange de donnees entre 2 processeurs (0 et 1)
if (rang==0) then
   call random_number(val)
   temps_debut=MPI_WTIME()
   call MPI_SEND(val,nb_val,MPI_DOUBLE_PRECISION,&
        nb_procs-rang-1,tag,MPI_COMM_WORLD,code)
   call MPI_RECV(val,nb_val,MPI_DOUBLE_PRECISION,&
        nb_procs-rang-1,tag,MPI_COMM_WORLD,statut,code)
   temps_fin=MPI_WTIME()
   print*, 'temps de communication : ', temps_fin-temps_debut
else if (rang==1)then
   call MPI_RECV(val,nb_val,MPI_DOUBLE_PRECISION,&
        nb_procs-rang-1,tag,MPI_COMM_WORLD,statut,code)
   call MPI_SEND(val,nb_val,MPI_DOUBLE_PRECISION,&
        nb_procs-rang-1,tag,MPI_COMM_WORLD,code)
end if

!desactivation environnement MPI
call MPI_FINALIZE(code)

end program
```

FIG. 3: Programme (en Fortran90) d'échange de messages entre deux processus.

qui) et l'étiquette ($tag\_recu = MPI\_ANY\_TAG$) lors de la réception d'un message.

– On peut communiquer avec un processus fictif de rang **MPI_PROC_NULL**. C'est très utile lors d'une action générique et qu'un des processus n'existe pas. Par exemple, si on décide d'envoyer un message à son processus voisin droit, au niveau des bords d'un domaine de calcul, il n'y a plus de processus droit, d'où l'utilité d'envoyer le message à un processus fictif. Si on envoie un message à un processus qui n'existe pas, cela provoque une erreur.

type MPI	type C
MPI_INT	signed int
$MPI\_UNSIGNED\_INT$	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
$MPI\_UNSIGNED\_CHAR$	unsigned char
MPI_PACKED	Types hétérogènes

TAB. 2: Principaux type de données pour le C

3.2 Communications collectives

Elles permettent de communiquer en un seul appel avec tous les processus d'un communicateur. Ce sont des fonctions bloquantes, c'est à dire que le système ne rend la main à un processus qu'une fois qu'il a terminé la tache collective.
Pour ce type de communication, les étiquettes sont automatiquement gérées par le système.
On détaille quelques fonctions de communication collective.

3.2.1 Diffusion générale : MPI_BCAST()

Cette fonction permet à un processus de diffuser ($BCAST$ pour broadcast) un message à tous les processus du communicateur indiqué, y compris à lui-même (figure 4). **MPI_BCAST**(val_emis, $taille\_emis$, $type\_emis$, $rang\_proc\_source$, $comm$, $code$).

FIG. 4: Fonction $MPI\_BCAST()$. Le processus $Proc1$ diffuse la donnée A (stockée dans sa mémoire) à tous les processus $Proc0$ à $Proc3$

3.2.2 Diffusion sélective de données réparties : MPI_SCATTER

Cette fonction permet à un processus de diffuser des données aux processus du communicateur indiqué de façon sélective. En fait le processus émetteur dispose de données qu'il répartit. Chaque processus (émetteur même compris) reçoit un paquet de données différent (figures 5 et 7).
MPI_SCATTER(*val_emis*, *taille_emis*, *type_emis*, *val_recu*, *taille_recu*, *type_recu*,*rang_proc_source*, *comm*, *code*).

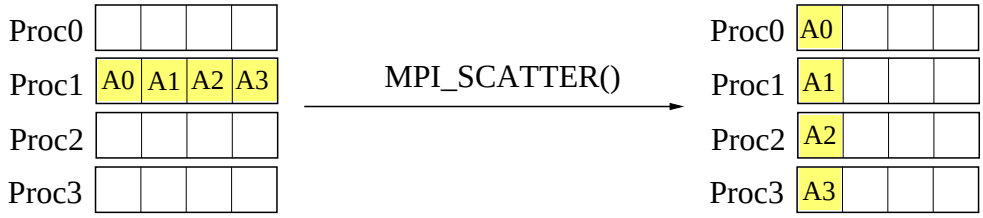


FIG. 5: Fonction $MPI\_SCATTER()$. Le processus $Proc1$ diffuse ses données $A0$,..., $A3$ (stockées dans sa mémoire) à tous les processus $Proc0$ à $Proc3$ de façon répartie.

3.2.3 Collecte de données réparties : MPI_GATHER()

Cette fonction permet au processus récepteur de collecter les données provenant de tous les processus (lui-même compris). Attention, le résultat n'est connu que par le processus récepteur (figure 6).
MPI_GATHER(*val_emis*, *taille_emis*, *type_emis*, *val_recu*, *taille_recu*, *type_recu*, *comm*, *code*).

FIG. 6: Fonction $MPI\_GATHER()$. Le processus $Proc1$ collecte les données $A0$,..., $A3$ provenant des processus $Proc0$ à $Proc3$.

```
program scatter

!utilisation de la bibliotheque mpi
use mpi

!declarations variables
implicit none
integer::nb_procs,N !nbre de processus,dimension valproc
integer,parameter::dim=120 !dimension valeur
integer::rang,code,i       !rang d'un processus, erreur
integer,allocatable,dimension(:):: valeur,valproc   !valeur

!initialisation environnement MPI
call MPI_INIT(code)

!Gestion des erreurs d'initialisation MPI
if(code/=MPI_SUCCESS)then
   print*,'erreur MPI',code
   stop
end if


!nombre de processus actifs de MPI_COMM_WORLD
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
!fonctions MPI : rang processus
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!allocation et initialisation de valeur
if (rang==0)  then
   allocate (valeur(dim))
   valeur(:)=(/(10*i,i=1,dim)/)
end if

!diffusion selective du processus 0 vers les autres processus
N=dim/nb_procs
allocate(valproc(N))
call MPI_SCATTER(valeur,N,MPI_INTEGER,&
     valproc,N,MPI_INTEGER,0,MPI_COMM_WORLD,code)
print*,'processeur ',rang,'reception valeurs ', valproc

!desallocation
if (rang==0) deallocate(valeur)
deallocate(valproc)
!desactivation environnement MPI
call MPI_FINALIZE(code)

end program scatter
```

FIG. 7: Exemple de programme utilisant la fonction $MPI\_SCATTER()$.

3.2.4 Collecte générale : MPI_ALLGATHER()

Cette fonction effectue la même chose que la fonction MPI_GATHER, excepté que le résultat de la collecte est connue de tous les processus du communicateur (figure 8). Ce serait l'équivalent d'un MPI_GATHER suivi d'un MPI_BCAST. **MPI_ALLGATHER**(val_emis, $taille\_emis$, $type\_emis$, val_recu, $taille\_recu$, $type\_recu$,$rang\_proc\_dest$, $comm$, $code$).

FIG. 8: Fonction $MPI\_ALLGATHER()$. Le processus $Proc1$ collecte les données $A0$,..., $A3$ provenant des processus $Proc0$ à $Proc3$ et le diffuse à tous les processus.

3.2.5 Synchronisation globale : MPI_BARRIER

Cette fonction bloque les processus à l'endroit où elle est appelée dans le programme. Les processus restent en attente au niveau de cette barrière ($BARRIER$) jusqu'à ce qu'ils y soient tous parvenus. Ensuite, ils sont libérés. **MPI_BARRIER**($comm$, $code$).

3.3 Opérations de réduction et communications collectives

Une réduction consiste à appliquer une opération à un ensemble d'éléments pour obtenir un scalaire. Cela peut être la somme des éléments d'un vecteur ou la valeur maximale d'un vecteur (voir tableau 3).

Nom	Opération
MPI_SUM	somme des éléments
MPI_PROD	produit des éléments
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum

TAB. 3: Quelques opérations de réduction prédéfinies

Certaines fonctions MPI permettent de faire des opérations de réductions en plus des communications collectives. c'est le cas des fonctions :

. **MPI_REDUCE()** qui permet de faire des opérations de réduction sur des données réparties. Le résultat de l'opération de réduction est récupéré sur un seul processus. **MPI_REDUCE**(val_emis, val_recu, $taille\_emis$, $type\_emis$, $operation$, $rang\_proc\_dest$, $comm$, $code$).

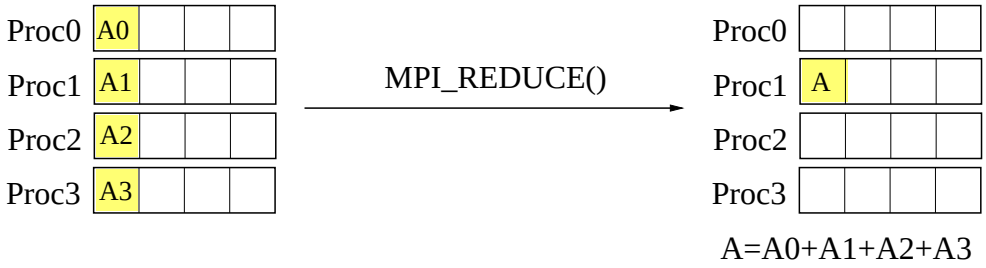


FIG. 9: Fonction $MPI\_REDUCE()$. Le processus $Proc1$ collecte les données $A0$,..., $A3$ provenant des processus $Proc0$ à $Proc3$ et fait une opération de réduction.

. **MPI_ALL_REDUCE()** qui permet de faire les mêmes opérations de réduction que **MPI_REDUCE()**. La différence est que le résultat de l'opération de réduction est connu de tous les processus d'un même communicateur. **MPI_ALL_REDUCE**(val_emis, val_recu, $taille\_emis$, $type\_emis$, $operation$, $comm$, $code$).

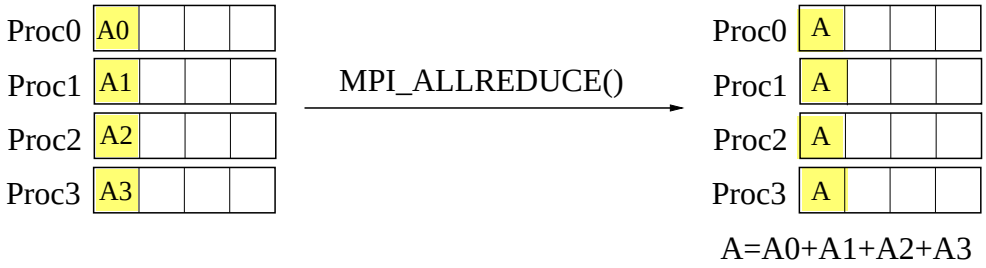


FIG. 10: Fonction $MPI\_ALLREDUCE()$. Opération de réduction et diffusion du résultat à l'ensemble des processus.

4 Optimisation d'un programme parallèle

L'optimisation d'un code séquentiel concerne la minimisation du temps de calcul. Lorsqu'on parallélise un code, un autre temps s'ajoute au temps de calcul, c'est le temps de communication entre les processus (**temps total = temps calcul + temps communication**). L'optimisation d'un code parallèle consiste donc à minimiser le temps de communication entre les processus. Celui-ci peut être mesuré via la fonction **MPI_WTIME()**.
Avant de se lancer dans l'optimisation d'un programme parallèle, il faut d'abord comparer le temps de calcul et le temps de communication au temps total de simulation. Si le temps de communication est prépondérant devant le temps de calcul, alors on peut passer à la phase d'optimisation. Cette étape consiste à réduire le temps de communication. Celui-ci contient un temps de préparation du message et un temps de transfert. Le temps de préparation contient un temps de latence pendant lequel les paramètres réseaux sont initialisés. Le reste du temps de préparation des messages (appelé aussi temps de surcoût) est lié à l'implémentation MPI et au mode de transfert utilisé (voir figure 11 pour plus de détails).
Il existe plusieurs possibilités pour optimiser le temps de communication, parmi elles :
– recouvrir les communications par des calculs.
– limiter les modes de transfert qui utilisent la recopie du message dans un espace mémoire temporaire ($buffering$).
– limiter les appels répétitifs aux fonctions de communication MPI (qui coûtent cher en temps).

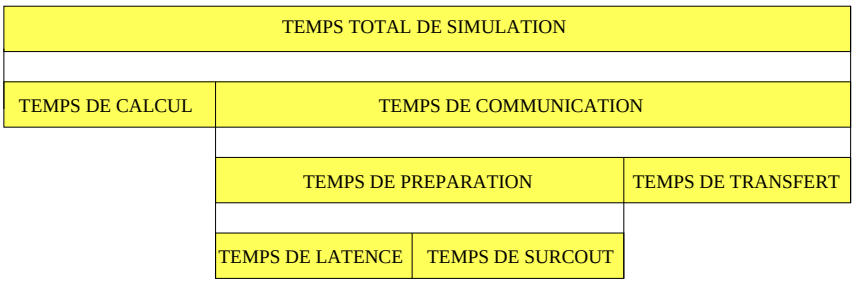


FIG. 11: Composition du total de simulation d'un programme parallèle.

4.1 Modes d'envoi des messages avec MPI

1. **Standard** : MPI **choisit** ou non de recopier le message à envoyer dans une zone mémoire tampon du processus émetteur. S'il y a recopie, l'action d'envoi se termine lorsque la recopie est terminée, donc avant que la réception du message ait commencée. Ceci permet de découpler l'envoi de la réception (asynchrone). S'il n'y a pas recopie du message, l'envoi s'achève une fois que le processus destinataire a bien reçu le message. L'envoi et la réception sont alors couplés (synchrone) (figures 12 et 13). MPI bascule automatiquement du mode asynchrone au mode synchrone suivant la taille des messages à transférer. Pour les petits messages, il y a recopie dans une zone tampon et pour les messages de grande taille, il n'y a pas de recopie.
2. **Synchroneous** (synchrone) : l'utilisateur impose un couplage entre l'envoi et la réception. L'envoi peut commencer avant même que l'opération de réception ait été initialisée. L'opération d'envoi s'achève une fois que l'opération de réception a été postée (par le processus récepteur) et le message bien reçu (figure 13).
3. **Buffered** : L'envoi du message s'achève une fois que la recopie du message dans une zone tampon est terminée. L'envoi et la réception sont découplés. Attention, l'utilisateur doit effectuer lui-même la recopie. Ce type d'envoi est déconseillé.
4. **Ready** : L'envoi du message ne peut commencer que si la réception correspondante a DÉJÀ été postée, Sinon il y a erreur. Ce type d'envoi est intéressant pour les applications clients-serveurs.

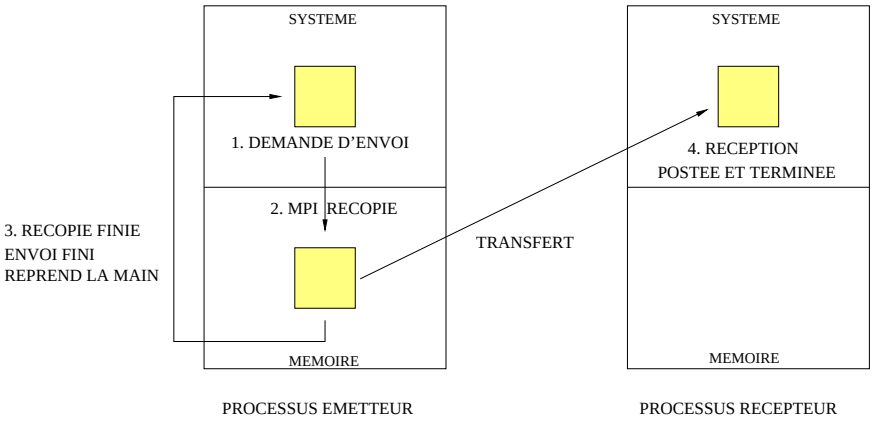


FIG. 12: Envoi standard avec recopie.

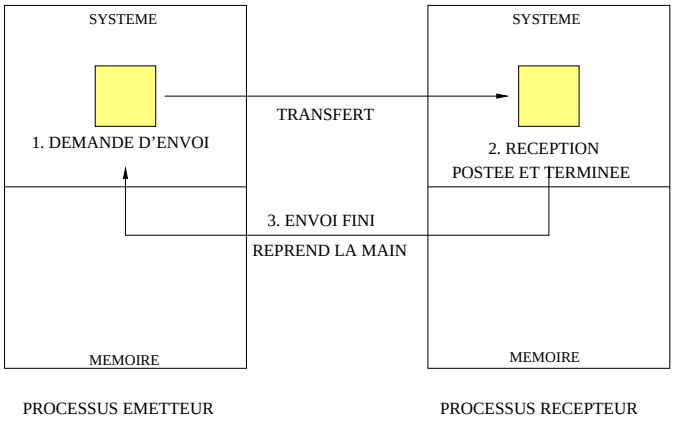


FIG. 13: Envoi standard sans recopie ou envoi synchrone.

4.2 Communications bloquantes et non-bloquantes

4.2.1 Communications bloquantes

Lors de communications bloquantes, le processus qui effectue une action (de communication) ne rend la main qu'une fois l'action terminée.
Envoi bloquant MPI_SEND() : Le processus émetteur termine son action une fois que le message a été envoyé et reçu par le processus récepteur.
Réception bloquante MPI_RECV() : Le processus récepteur ne rend la main que lorsqu'il a bien reçu les données.

4.2.2 Communications non-bloquantes

Les communications non-bloquantes permettent d'optimiser le temps de communication dans le sens où elles les recouvrent par des calculs. En effet, le processus qui effectue une opération de communication rend la main avant que l'opération ait été terminée.

. **Envoi non-bloquant MPI_ISEND()** : le processus émetteur initialise l'opération d'envoi mais ne la réalise pas. L'appel à la fonction MPI sera terminée avant que le message ne soit parti. Ainsi, son transfert à partir de la mémoire du processeur émetteur peut être fait simultanément avec des calculs après l'initialisation et avant l'envoi effectif. Attention, l'utilisateur doit lui-même s'assurer que le message a bien été envoyé avec des fonctions MPI adaptées (**MPI_TEST()** et **MPI_WAIT()**).
La fonction **MPI_ISEND**(val_emis, $taille\_emis$, $type\_emis$, $rang\_proc\_dest$, tag_emis, $comm$, $requete$, $code$) possède un argument

de plus que la fonction $MPI\_SEND()$, *requete*. L'argument *requete* ($< out >$) :
  . identifie l'opération de communication,
  . contient des informations sur l'opération de communication, à savoir le mode d'envoi, la destination du message, ...
  . fait correspondre l'opération d'initialisation de la communication et celle de réalisation de la communication.

. **Réception non-bloquante MPI_IRECV()** : Le processus récepteur initialise la réception mais ne la réalise pas. L'appel à la fonction MPI sera terminée avant que le message ne soit reçu. Ainsi, la réception du message par le processus récepteur peut être faite simultanément avec des calculs après l'initialisation et avant la réception effective. Attention, l'utilisateur doit lui-même s'assurer que le message a bien été reçu avec des fonctions MPI adaptées (**MPI_TEST()** et **MPI_WAIT()**).

Pour s'assurer que l'opération de communication a bien été effectuée, l'utilisateur peut utiliser deux fonctions MPI :
- **MPI_TEST**(*requete*, $flag$, *statut*) : cette fonction permet de tester si l'opération de communication, identifiée par *requete* ($< in >$), est terminée. MPI_TEST retourne 2 arguments de sortie ($< out >$) $flag$ et *statut*. Si l'opération de communication est terminée, alors $flag = .true.$. Sinon $flag = .false.$.
- **MPI_WAIT** (*requete*, *statut*) : cette fonction oblige l'opération de communication, identifiée par *requete* ($< in >$), à s'achever.

4.3 Synthèse

mode	bloquant	non-bloquant
envoi standard	$MPI\_SEND()$	$MPI\_ISEND()$
envoi synchrone	$MPI\_SSEND()$	$MPI\_ISSEND()$
réception	$MPI\_RECV()$	$MPI\_IRECV()$

Tab. 4: Principaux modes de transfert

Arguments des différentes fonctions MPI du tableau 4 :
- $MPI\_SEND(val\_emis, taille\_emis, type\_emis, rang\_proc\_dest, tag\_emis, comm, code)$
- $MPI\_ISEND(val\_emis, taille\_emis, type\_emis, rang\_proc\_dest, tag\_emis, requete, code)$
- $MPI\_SSEND(val\_emis, taille\_emis, type\_emis, rang\_proc\_dest, tag\_emis, requete, code)$

- $MPI\_ISSEND(val\_emis, taille\_emis, type\_emis, rang\_proc\_dest, tag\_emis, requete, code)$
- $MPI\_RECV(val\_recu, taille\_recu, type\_recu, rang\_proc\_dest, tag\_recu, comm, statut, code)$
- $MPI\_IRECV(val\_recu, taille\_recu, type\_recu, rang\_proc\_dest, tag\_recu, comm, requete, code)$

5 Types de données dérivés

Dans les communications MPI, les données transférées sont typées. MPI dispose de types prédéfinis comme $MPI\_INTEGER$, MPI_REAL, ...
On peut créer des structures de données plus complexes, soit homogènes (constitués de données de même type) soit hétérogènes (constitués de données de types différents).
La création d'un **type dérivé** doit être suivi de sa validation via la fonction **MPI_TYPE_COMMIT()**. Pour réutiliser le même type dérivé, il faut d'abord le libérer avec la fonction **MPI_TYPE_FREE()**.
Attention, il faut éviter de passer directement des sections de tableaux. Mieux vaut passer par les types dérivés !

5.1 Types de données dérivés homogènes

5.1.1 Données contigües

On peut construire une structure de données homogènes à partir d'un ensemble de données, de type prédéfini, contigües en mémoire, via la fonction **MPI_TYPE_CONTIGUOUS**(*nb_element*, *old_type*, *new_type*, *code*) :

. **nb_element** (< *in* >) est le nombre d'éléments de type prédéfini à mettre dans le type dérivé,
. **old_type** (< *in* >) est le type des éléments qui constitue le nouveau type dérivé,
. **new_type** (< *out* >) est le nom du nouveau type dérivé et *code* le retour d'erreur,
. **code** (< *out* >) est le code d'erreur.

A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34

FIG. 14: création d'un type dérivé *colonne* à partir d'une matrice de réels $A_{i,j}$, $i \in [1,3]$, $j \in [1,4]$. $nb\_element = 3$, $old\_type = MPI\_REAL$

Les données contigües peuvent représenter une colonne d'une matrice de réels $A_{i,j}$, $i \in [1,3]$, $j \in [1,4]$, par exemple (partie colorée en jaune de la figure 14).

5.1.2 Données non contigües avec un pas constant

On peut construire une structure de données homogènes à partir d'un ensemble de données, de type prédéfini, distantes d'un pas constant en mémoire.

Le pas peut être donné en nombre d'éléments : c'est le cas par exemple pour passer une ligne d'une matrice de réels $A_{i,j}$, $i \in [1,3]$, $j \in [1,4]$,par exemple (partie colorée en jaune de la figure 15). Dans ce cas, on utilise la fonction **MPI_TYPE_VECTOR**(*nb_blocs*, *longueur_bloc*, *pas*, *old_type*, *new_type*, *code*).

- **nb_blocs** (< *in* >) est le nombre de blocs,
- **longueur_bloc** (< *in* >) est le nombre d'éléments (de type *old_type*) dans chaque bloc,
- **pas** (< *in* >) est le nombre d'éléments (de type *old_type*) entre chaque début de bloc,
- **old_type** (< *in* >) est le type de l'élément à partir duquel on souhaite créer un type dérivé,
- **new_type** (< *out* >) est le type dérivé.

Le pas peut être donné en nombre d'octets : c'est le cas lorsque le type dérivé est construit à partir de types plus complexes que les types prédéfinis. Dans ce cas, on utilise la fonction **MPI_TYPE_CREATE_HVECTOR**(*nb_blocs*, *longueur_bloc*, *pas*, *old_type*, *new_type*, *code*) :

- **nb_blocs** (< *in* >) est le nombre de blocs,
- **longueur_bloc** (< *in* >) est le nombre d'éléments (de type *old_type*) dans chaque bloc,
- **pas** (< *in* >) est le nombre d'octets (de type *old_type*) entre chaque début de bloc. Il doit être déclaré comme : **integer(kind=MPI_ADDRESS_KIND) : :pas**.
- **old_type** (< *in* >) est le type de l'élément à partir duquel on souhaite créer un type dérivé.
- **new_type** (< *out* >) est le type dérivé.

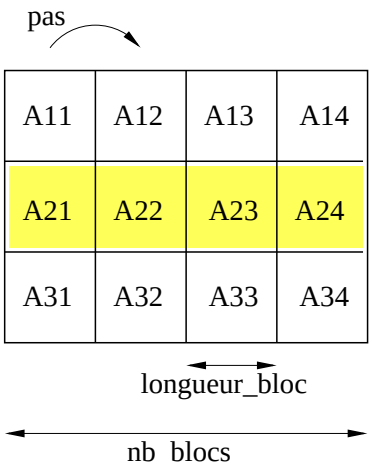


FIG. 15: création d'un type dérivé *ligne* à partir d'une matrice de réels $A_{i,j}$, $i \in [1,3]$, $j \in [1,4]$. $nb\_blocs = 4$, $longueur\_bloc = 1$, $pas = 3$, $old\_type = MPI\_REAL$.

5.1.3 Données non contigües avec un pas variable

On peut construire une structure de données homogènes composée d'une séquence de blocs contenant un nombre variable d'éléments de type prédéfini et dont les blocs sont distants d'un pas variable en mémoire.

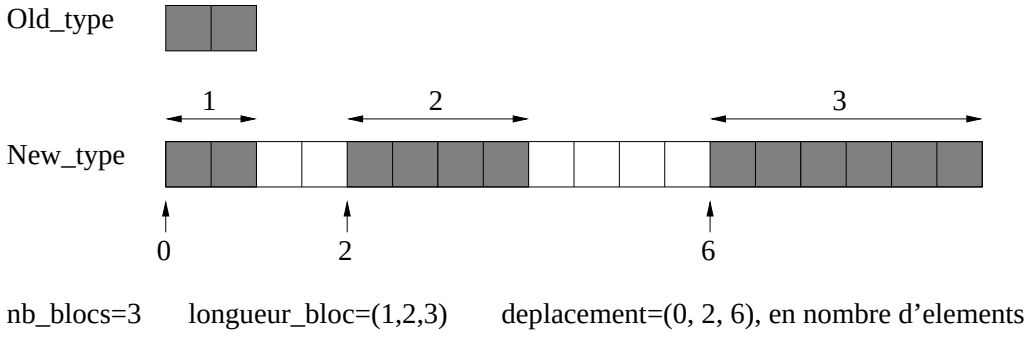


FIG. 16: Construction d'une structure de données avec $MPI\_TYPE\_INDEXED$.

Le pas peut être donné en nombre d'éléments : Dans ce cas on utilise la fonction **MPI_TYPE_INDEXED**(nb_blocs, $longueur\_bloc$, $deplacement$, old_type, new_type, $code$).

– **nb_blocs** ($< in >$) est le nombre de blocs,
– **longueur_bloc** ($< in >$) est un tableau de dimension nb_blocs contenant le nombre d'éléments de type old_type par bloc,
– **deplacement** ($< in >$) est un tableau de dimension nb_blocs contenant l'espacement entre chaque bloc. Cet espacement est donné en multiple de old_type (voir figure 16).

Le pas peut être donné en nombre d'octets quand la structure est construite à partir d'éléments de type plus complexes que ceux prédéfinis : Dans ce cas on utilise la fonction : **MPI_TYPE_CREATE_HINDEXED** (nb_blocs, $longueur\_bloc$, $deplacement$, old_type,new_type,$code$).
– **nb_blocs** ($< in >$) est le nombre de blocs,
– **longueur_bloc** ($< in >$) est un tableau de dimension nb_blocs contenant le nombre d'éléments de type old_type par bloc,
– **deplacement** ($< in >$) est un tableau de dimension nb_blocs contenant l'espacement entre chaque bloc. Cet espacement est donné en octets (voir figure 17) et doit être déclarée comme $integer(kind = MPI\_ADDRESS\_KIND), dimension(nb\_blocs) :: pas$.

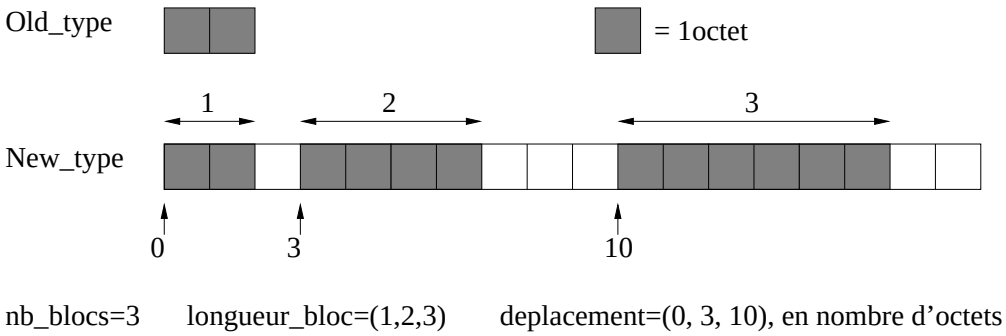


FIG. 17: Construction d'une structure de données avec $MPI\_TYPE\_CREATE\_HINDEXED$.

5.2 Types dérivés hétérogènes

C'est le constructeur le plus général. Il permet de créer des types de données hétérogènes équivalent à des structures en C ou à des types dérivés en $Fortran90$. Pour cela, il faut d'abord créer une structure constituée d'éléments de types différents.

On utilise ensuite la fonction
MPI_TYPE_CREATE_STRUCT(*nb_elements*, *longueur_bloc*, *deplacement*, *tableau_types*,*new_type*,*code*) (voir figure 18).

- **nb_elements** (< *in* >) est le nombre d'éléments de la structure, chaque élément constitue un bloc.
- **longueur_bloc** (< *in* >) est un tableau de dimension *nb_elements* contenant le nombre d'éléments de chaque bloc de la structure,
- **deplacement** (< *in* >) est un tableau de dimension *nb_elements* contenant l'adresse de chaque début de bloc par rapport à celle du premier bloc, en octets. Il doit être déclaré comme : $integer(kind = MPI\_ADDRESS\_KIND), dimension(nb\_blocs) :: deplacement$. On détermine cette adresse via la fonction **MPI_GET_ADDRESS**(*element_struct*,*adress_element*, *code*).
- **tableau_types** (< *in* >) est un tableau de dimension *nb_elements* contenant le type de chaque bloc de la structure.
- **new_type** (< *out* >) est le nouveau type de données.

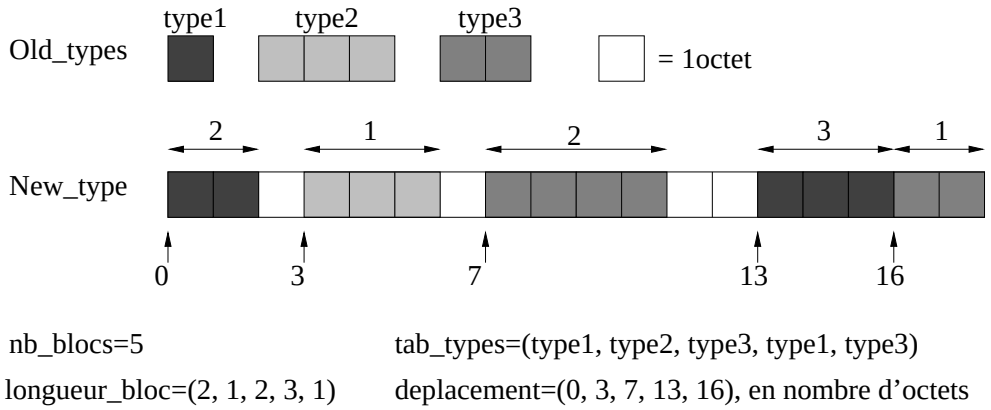


FIG. 18: Construction d'une structure de données hétérogène avec $MPI\_TYPE\_CREATE\_STRUCT$.

5.3 Exemples

5.3.1 Exemples sur des types de données dérivés homogènes

Exemple1 : Construction d'un type sous matrice avec la fonction $MPI\_TYPE\_VECTOR()$: On construit un type dérivé sous-matrice à partir d'éléments d'une matrice de réels (voir programme figure 19).

```
program sousmat

USE MPI

implicit none

integer,dimension(MPI_STATUS_SIZE)::statut
integer:: nb_procs,rang,tag=102,code
integer:: type_sousmat !types derives
integer,parameter:: nb_lignes=5,nb_col=6 !dimensions matrices
integer,parameter:: nb_lignes2=2,nb_col2=3 !dimensions sous matrices
real,dimension(nb_lignes,nb_col)::A0,A1
integer::i,j

!initialisation MPI
call MPI_INIT(code)

!fonctions MPI : nombre et rang processus
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!creation d'un type derive sousmatrice
call MPI_TYPE_VECTOR(nb_col2,nb_lignes2,nb_lignes,MPI_REAL,type_sousmat,code)
call MPI_TYPE_COMMIT(type_sousmat,code)

!initialisation matrices
do i=1,nb_lignes
   do j=1,nb_col
      A0(i,j)=(i+2*j)*1.00
   end do
end do
A1=0.0

!communications MPI
if (rang==0) then
   do i=1,nb_lignes
      print*,'A0',A0(i,:)
   end do
   !envoi de la matrice transpose, on donne le 1er element mat(1,1)
   call MPI_SEND(A0(2,1),1,type_sousmat,1,tag,MPI_COMM_WORLD,code)
elseif (rang==1) then
   !reception matrice transpose matt
   call MPI_RECV(A1(1,1),1,type_sousmat,0,tag,MPI_COMM_WORLD,statut,code)
   do i=1,nb_lignes
      print*,'A1',A1(i,:)
   end do
end if

!liberation type_sousmat
call MPI_TYPE_FREE(type_sousmat,code)

!desallocation MPI
call MPI_FINALIZE(code)

end program sousmat
```

```
-bash-2.05b$ mpiexec -n 2 /users/uppa/delage/test/sousmat
 A0 3.000000000 5.000000000 7.000000000 9.000000000 11.00000000 13.00000000
 A0 4.000000000 6.000000000 8.000000000 10.00000000 12.00000000 14.00000000
 A0 5.000000000 7.000000000 9.000000000 11.00000000 13.00000000 15.00000000
 A0 6.000000000 8.000000000 10.00000000 12.00000000 14.00000000 16.00000000
 A0 7.000000000 9.000000000 11.00000000 13.00000000 15.00000000 17.00000000

 A1 4.000000000 6.000000000 8.000000000 1.000000000 1.000000000 1.000000000
 A1 5.000000000 7.000000000 9.000000000 1.000000000 1.000000000 1.000000000
 A1 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000
 A1 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000
 A1 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000 1.000000000
```

FIG. 19: Construction de la transposée d'une matrice à l'aide de types dérivés : programme en *Fortran*90.

Exemple2 : construction de la transposée d'une matrice à l'aide de type dérivés (figures 20 et 21) (fonctions MPI_TYPE_VECTOR et $MPI\_TYPE\_CREATE\_HVECTOR$) : on construit un type ligne à partir d'éléments d'une matrice de réels, puis un type dérivé transpose (correspondant à la transposée d'une matrice) à partir du type dérivé ligne.

```
program transpose

  USE MPI

  implicit none
  integer,dimension(MPI_STATUS_SIZE)::statut
  integer:: code,nb_procs,rang,taille_reel,i,j,nn
  integer(kind=MPI_ADDRESS_KIND):: pas
  integer:: type_ligne,type_transpose !types derives
  integer,parameter:: nb_lignes=5,nb_col=4,tag=102 !dim matrices, etiquette
  real,dimension(nb_lignes,nb_col)::mat
  real,dimension(nb_col,nb_lignes)::matt

  !initialisation MPI
  call MPI_INIT(code)

  !fonctions MPI : nombre et rang processus
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  !connaitre la taille en octets du type MPI_REAL
  call MPI_TYPE_SIZE(MPI_REAL,taille_reel,code)

  !creation et validation d'un type derive ligne
  call MPI_TYPE_VECTOR(nb_col,1,nb_lignes,MPI_REAL,type_ligne,code)
  call MPI_TYPE_COMMIT(type_ligne,code)

  !creation et validation d'un type transpose (pr calculer la transpose de mat)
  pas=taille_reel
  call MPI_TYPE_CREATE_HVECTOR(nb_lignes,1,pas,type_ligne,type_transpose,code)
  call MPI_TYPE_COMMIT(type_transpose,code)

  nn=nb_lignes*nb_col
  if (rang==0) then
     mat(:,:)=reshape((/(i,i=1,nn)/),(/nb_lignes,nb_col/)) !initialisation mat
     do i=1,nb_lignes
        print*,'mat',mat(i,:)
     end do
     !envoi de la matrice transpose, on donne le 1er element mat(1,1)
     call MPI_SEND(mat(1,1),1,type_transpose,1,tag,MPI_COMM_WORLD,code)
  elseif (rang==1) then
     !reception matrice transpose matt
     call MPI_RECV(matt,nn,MPI_REAL,0,tag,MPI_COMM_WORLD,statut,code)
     do i=1,nb_col
        print*,'matt',matt(i,:)
     end do
  end if

  !desallocation MPI
  call MPI_FINALIZE(code)

end program transpose
```

FIG. 20: Construction de la transposée d'une matrice à l'aide de types dérivés : programme.

Exemple3 : construction de matrices à partir de types dérivés homogènes à pas variables (fonctions $MPI\_TYPE\_INDEXED()$) (figure 23 et son résultat sur la figure 22).

```
bash-2.05b$ mpiexec -n 2 /users/uppa/delage/test/transpose
mat 1.000000000 6.000000000 11.00000000 16.00000000
mat 2.000000000 7.000000000 12.00000000 17.00000000
mat 3.000000000 8.000000000 13.00000000 18.00000000
mat 4.000000000 9.000000000 14.00000000 19.00000000
mat 5.000000000 10.00000000 15.00000000 20.00000000

matt 1.000000000 2.000000000 3.000000000 4.000000000 5.000000000
matt 6.000000000 7.000000000 8.000000000 9.000000000 10.00000000
matt 11.00000000 12.00000000 13.00000000 14.00000000 15.00000000
matt 16.00000000 17.00000000 18.00000000 19.00000000 20.00000000
```

FIG. 21: Construction de la transposée d'une matrice à l'aide de types dérivés : résultat.

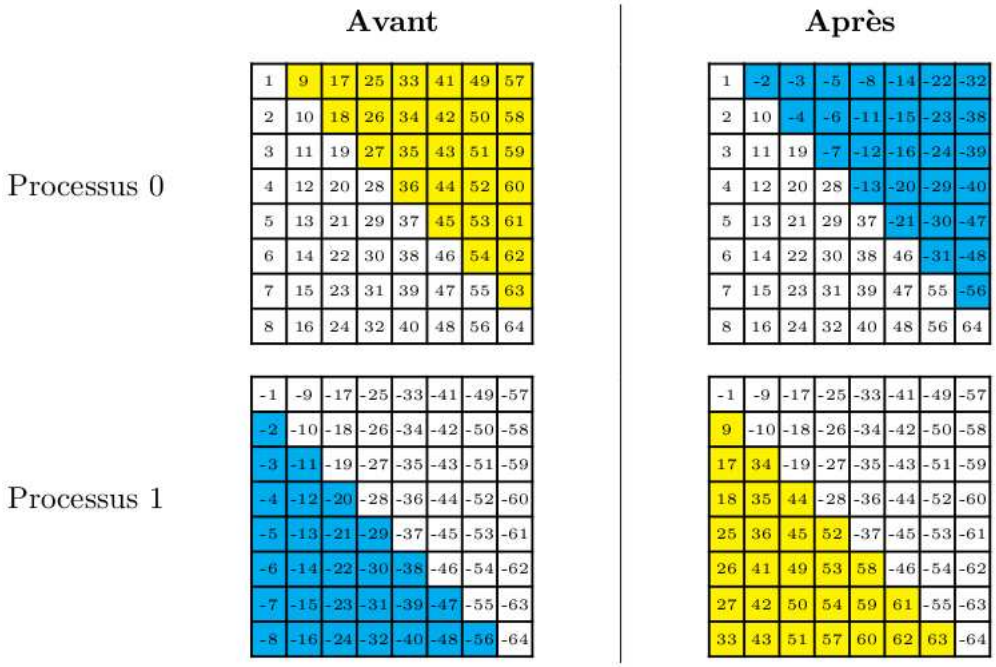


FIG. 22: résultat du programme *triangle.f*90 (figure du cours de l'Idris).

5.3.2 Exemples sur des types dérivés hétérogènes

La figure montre un exemple de programme en *Fortran*90 utilisant une structure de données hétérogène :

```
program triangle

USE MPI

implicit none

integer,parameter:: nb_lignes=8,nb_col=8,etiquette=1000
real,dimension(nb_lignes,nb_col)::A0
integer,dimension(MPI_STATUS_SIZE)::statut
integer:: nb_procs,rang,code
integer:: type_triangle !types derives
integer,dimension(nb_col)::longueurs_blocs, deplacement

integer::i,j

!initialisation MPI
call MPI_INIT(code)

!fonctions MPI : nombre et rang processus
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

 !initialisation matrice
do i=1,nb_lignes
   do j=1,nb_col
      A0(i,j)=sign((j-1)*nb_lignes+i,-rang)
   end do
end do


!creation d'un type derive triangle puis validation
if (rang==0) then
   longueurs_blocs(:)=(/(i-1,i=1,nb_col)/)
   deplacement(:)=(/((i-1)*nb_lignes,i=1,nb_col)/)
else if (rang==1) then
   longueurs_blocs(:)=(/((nb_lignes-i),i=1,nb_col)/)
   deplacement(:)=(/((i-1)*nb_lignes+i,i=1,nb_col)/)
end if
call MPI_TYPE_INDEXED(nb_col,longueurs_blocs,deplacement,MPI_REAL,type_triangle,code)
call MPI_TYPE_COMMIT(type_triangle,code)

!communications MPI
call MPI_SENDRECV_REPLACE(A0,1,type_triangle,nb_procs-rang-1,etiquette,&
     nb_procs-rang-1,etiquette,MPI_COMM_WORLD,statut,code)

!liberation type_sousmat
call MPI_TYPE_FREE(type_triangle,code)

!desallocation MPI
call MPI_FINALIZE(code)

end program triangle
```

FIG. 23: Programme montrant l'échange de sous matrices triangulaires entre deux processus.

```
program interaction_particules
use MPI

implicit none
integer,parameter                                  ::etiquette=100,n=1, nb_blocs=3
integer,dimension(MPI_STATUS_SIZE)                 ::statut
integer                                            ::rang,code,type_particule,i
integer,dimension(nb_blocs)                        ::longueurs_blocs,types
integer(kind=MPI_ADDRESS_KIND),dimension(nb_blocs)::deplacement,adresses

type Particule
   character(len=5)                                ::categorie
   integer                                         ::masse
   real,dimension(3)                               ::coords
end type Particule
type(Particule)                                    ::p0,p1

!initialisation MPI et rang processus
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!creation et validation type de donnees heterogene :type_particule
types=(/MPI_CHARACTER,MPI_INTEGER,MPI_REAL/)
longueurs_blocs=(/5,1,3/)
if(rang==0)then
   CALL MPI_GET_ADDRESS(p0%categorie,adresses(1),code)
   CALL MPI_GET_ADDRESS(p0%masse,adresses(2),code)
   CALL MPI_GET_ADDRESS(p0%coords,adresses(3),code)
else if(rang==1)then
   CALL MPI_GET_ADDRESS(p1%categorie,adresses(1),code)
   CALL MPI_GET_ADDRESS(p1%masse,adresses(2),code)
   CALL MPI_GET_ADDRESS(p1%coords,adresses(3),code)
end if
do i=1,nb_blocs
   deplacement(i)=adresses(i)-adresses(1)
end do
CALL MPI_TYPE_CREATE_STRUCT(nb_blocs,longueurs_blocs,deplacement,types,type_particule,code)

CALL MPI_TYPE_COMMIT(type_particule,code)!validation type_particule

!communications
if (rang==0)then
   p0%categorie='azote'; p0%masse=6;p0%coords=(/1,2,5/)
   print*,'p0',p0
   CALL MPI_SEND(p0%categorie,1,type_particule,1,etiquette,MPI_COMM_WORLD,code)
else if (rang==1) then
   CALL MPI_RECV(p1%categorie,1,type_particule,0,etiquette,MPI_COMM_WORLD,statut,code)
   print*,'p1',p1
end if

 !liberation type_particule et desallocation MPI
CALL MPI_TYPE_FREE(type_particule,code)
CALL MPI_FINALIZE(code)

end program interaction_particules
```

FIG. 24: Interaction de deux particules avec *MPI_TYPE_CREATE_STRUCT*.

6 Topologies

MPI permet de définir des topologies virtuelles de type cartésien ou graphe. Ces topologies font correspondre le domaine de calcul à une grille de processus et permettent donc de répartir les processus de manière régulière sur le domaine de calcul. Ceci se révèle très utile pour les problèmes de type décomposition de domaine.

6.1 Topologies de type cartésien

Dans cette topologie :
- chaque processus est défini dans une grille de processus,
- la grille peut être périodique ou non,
- les processus de cette topologie sont identifiés par leurs coordonnées dans la grille.

6.1.1 Création d'une topologie cartésienne

Pour créer une topologie cartésienne contenant un ensemble de processus qui appartienne à un communicateur donné ($comm\_ancien$), on utilise la fonction : **MPI_CART_CREATE**($comm\_ancien$, $ndims$, $dims$, $periods$, $reorganisation$, $comm\_nouveau$, $code$).

. **comm_ancien** ($< in >$) : entier indiquant le nom de l'ancien communicateur (par défaut MPI_COMM_WORLD),
. **ndims** ($< in >$) : entier indiquant la dimension d'espace (2 pour $2D$),
. **dims** ($< in >$) : tableau d'entiers de dimension $ndims$ indiquant le nombre d'éléments suivant chaque direction,
. **periods** ($< in >$) : tableau de logicals de dimension $ndims$ indiquant la periodicité ($true$ pour oui),
. **reorganisation** ($< in >$) : logical, $reorganisation = true$ si renumérotation des processus dans $comm\_nouveau$ (conseillé mais faire très attention), $reorganisation = false$ sinon. Si $reorganisation = true$, MPI renumérote les processus suivant l'axe y puis x en $2D$, suivant l'axe z, y puis x en $3D$.
. **comm_nouveau** ($< out >$) : nom du nouveau communicateur (de type entier).
. **code** ($< out >$) : code de retour (de type entier).

Attention, cette communication est collective, elle concerne donc l'ensemble des processus appartenant à $comm\_ancien$.

6.1.2 Quelques fonctions utiles

. La fonction **MPI_DIMS_CREATE**(*nb_procs*, *ndims*, *dims*, *code*) détermine le nombre de processus suivant chaque dimension de la grille en fonction du nombre total de processus.
  . **nb_procs** (< *in* >) est un entier indiquant le nombre de processus,
  . **ndims** (< *in* >) est un entier indiquant la dimension d'espace,
  . **dims** (< *inout* >) est un tableau d'entiers de dimension *ndims* indiquant le nombre d'éléments suivant chaque direction. Si en entrée, *dims* est un tableau d'entiers nuls, c'est *MPI* qui répartit les processus dans chaque direction en fonction du nombre total de processus. Le tableau 5 montre quelques répartitions de processus en fonction de la valeur initiale de *dims* (voir aussi la figure 25).
  . **code** (< *out* >) est le code d'erreur.

Entrées < *in* >			Sortie < *out* >
nb_procs	*ndims*	*dims*	*dims*
8	2	(0, 0)	(4, 2)
16	3	(0, 0, 0)	(4, 2, 2)
16	3	(0, 4, 0)	(2, 4, 2)
16	3	(0, 3, 0)	*error*

TAB. 5: Répartition des processus avec la fonction *MPI_DIMS_CREATE* dans une topographie cartésienne par *MPI*.

. La fonction **MPI_CART_RANK**(*comm_nouveau*, *coords*, *rang*, *code*) donne le rang *rang* du processus associé au coordonnées *coords* dans la grille (voir aussi la figure 25).
  . **comm_nouveau** (< *in* >) est un entier indiquant le nouveau communicateur,
  . **coords** (< *in* >) est un tableau d'entiers de dimension *ndims* contenant les coordonnées du processus de rang *rang* dans la grille,
  . **rang** (< *out* >) est un entier indiquant le rang du processus ayant pour coordonnées *coords*
  . **code** (< *out* >) est le code d'erreur (type entier).
. La fonction **MPI_CART_COORDS**(*comm_nouveau*, *rang*, *ndims*, *coords*, *code*) est la fonction inverse de *MPI_CART_RANK* dans le sens où elle donne les coordonnées *coords* d'un processus de rang *rang* dans la grille.
  . **comm_nouveau** (< *in* >) est un entier indiquant le nouveau communicateur,
  . **rang** (< *in* >) est un entier indiquant le rang du processus,

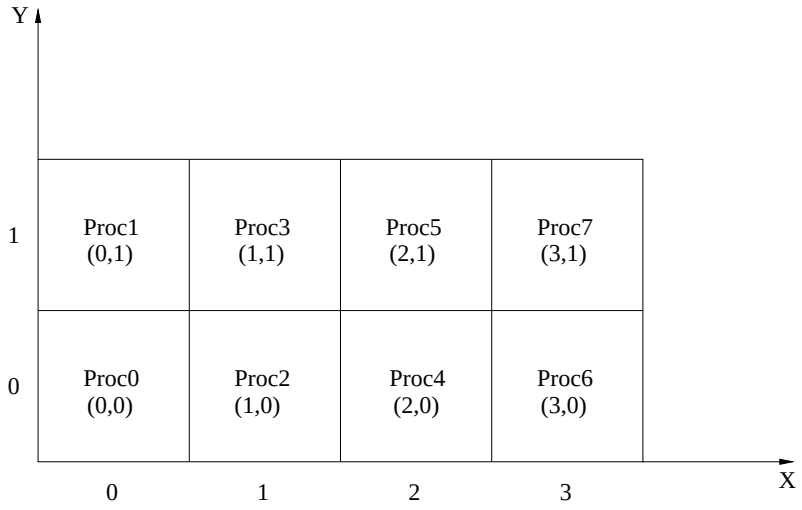


FIG. 25: Répartition de 8 processus ($Proc0$ à $Proc7$) choisi par MPI sur la grille. Chaque processus a des coordonnées.

. **ndims** ($< in >$) est un entier indiquant le nombre de dimensions,
. **coords** ($< out >$) est un tableau d'entiers de dimension *ndims* contenant les coordonnées du processus de rang *rang* dans la grille. (voir aussi la figure 25).

. La fonction **MPI_CART_SHIFT**(*comm_nouveau*, *direction*, *pas*, *rang_precedent*, *rang_suivant*, *code*) donne le rang des processus voisins (*rang_precedent* et *rang_suivant*) dans la direction choisie, *direction*.
. **comm_nouveau** ($< in >$) est un entier indiquant le nouveau communicateur.
. **direction** ($< in >$) est un entier indiquant la direction dans laquelle on va chercher les rangs des processus voisins. $direction = 0$ suivant x et $direction = 1$ suivant y (figure 25).
. **pas** ($< in >$) correspond au déplacement, $pas = 1$ pour les voisins directs (type entier).
. **ndims** ($< in >$) est un entier indiquant le nombre de dimensions.
. **rang_precedent** ($< out >$) est le rang du processus voisin Ouest si $direction = 0$, Sud si $direction = 1$.
. **rang_suivant** ($< out >$) est un entier indiquant le rang du processus voisin Est si $direction = 0$, Nord si $direction = 1$.
. **code** ($< out >$) est un entier indiquant le code d'erreur.

6.1.3 Exemple

La figure 26 montre comment on peut utiliser les différentes fonctions MPI concernant la topologie cartésienne.

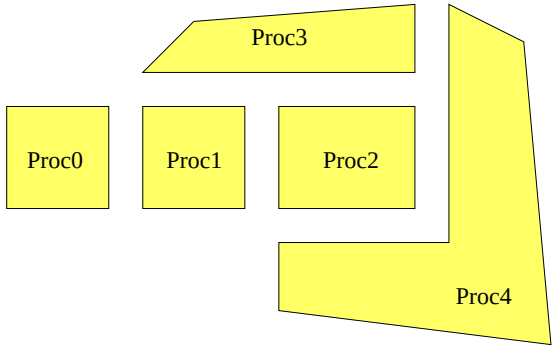
```
program topocart

use MPI

implicit none
integer                 :: code,nb_procs,rang,comm2D,direction,pas
integer,parameter       :: ndims=2,N=1,E=2,S=3,W=4
logical                 :: reorganisation
integer,dimension(4)    :: voisin
integer,dimension(ndims):: dims,coords
logical,dimension(ndims):: periods

!initialisation MPI
call MPI_INIT(code)

!fonctions MPI : nombre et rang processus
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

!initialisation des element de la topologie cartesienne
periods(1)=.false.;periods(2)=.true.;reorganisation=.false.
dims(:)=0
call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)

!creation d'une topologie cartesienne
call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm2D,code)

!connaitre les coordonnees dans la topologie
call MPI_COMM_RANK(comm2D,rang,code)
call MPI_CART_COORDS(comm2D,rang,ndims,coords,code)
print*, 'processus',rang,', mes coordonnees sont : ',coords(1),coords(2)

!initialisation des voisins au processus fictif MPI_PROC_NULL
voisin(:)=MPI_PROC_NULL

!recherche des voisins West et Est
direction=0;pas=1
call MPI_CART_SHIFT(comm2D,direction,pas,voisin(W),voisin(E),code)
!recherche des voisins Nord et Sud
direction=1;pas=1
call MPI_CART_SHIFT(comm2D,direction,pas,voisin(S),voisin(N),code)
print*, 'processus',rang,', mon voisin W est : ',voisin(W)
print*, 'processus',rang,', mon voisin E est : ',voisin(E)
print*, 'processus',rang,', mon voisin S est : ',voisin(S)
print*, 'processus',rang,', mon voisin N est : ',voisin(N)

!desallocation MPI
call MPI_FINALIZE(code)

end program topocart
```

FIG. 26: programme $F90$ montrant l'utilisation des différentes fonctions de topologie cartésienne

6.2 Topologies de type graphe

Lorsque la géométrie du domaine de calcul devient complexe, la topologie de graphe de processus est plus appropriée. On peut alors répartir les processus sur des sous domaines de géométries complexes elles aussi. Chaque processus peut avoir un nombre quelconque de voisins.

6.2.1 Création d'une topologie de type graphe



Numero Proc	Numero proc voisins
0	1
1	0, 2, 3
2	1, 3, 4
3	1, 2, 4
4	2, 3

nb_procs=5 index(1, 4, 7, 10, 12) liste_voisins=(1, 0,2,3, 1,3,4, 1,2,4, 2,3)

FIG. 27: Exemple de topologie de type graphe.

La fonction **MPI_GRAPH_CREATE**(*comm_ancien*, *nb_procs*, *index*, *liste_voisins*, *reorganisation*, *comm_nouveau*, *code*) permet de créer ce type de topologie (voir figure 27 pour exemple).

- **comm_ancien** ($< in >$) : nom de l'ancien communicateur (par défaut MPI_COMM_WORLD),
- **nb_procs** ($< in >$) : entier indiquant le nombre de processus.
- **index** ($< in >$) : tableau d'entiers de taille *nb_procs* indiquant pour chaque processus l'indice du tableau *liste_voisins* qui contient le dernier voisin.
- **liste_voisins** ($< in >$) : tableau d'entiers de taille le nombre total de voisins (sur tous les processus) indiquant successivement les rangs des processus voisins pour chaque processus.
- **reorganisation** ($< in >$) : logical, $reorganisation = true$ si renumérotation des processus dans *comm_nouveau* (conseillé mais faire très attention), $reorganisation = false$ sinon. Si $reorganisation = true$, MPI renumérote les processus suivant l'axe y puis x en $2D$, suivant l'axe z, y puis x en $3D$.

- **comm_nouveau** (< *out* >) : nom du nouveau communicateur (de type entier).
- **code**(<**out**>) : code de retour (de type entier).

6.2.2 Quelques fonctions utiles

. La fonction **MPI_GRAPH_NEIGHBORS_COUNT** (*comm_nouveau*, *rang*, *nb_voisins*, *code*) détermine le nombre de processus voisins pour un processus donné.
  . **comm_nouveau** (< *in* >) est un entier indiquant le nombre de processus,
  . **rang** (< *in* >) est un entier indiquant le rang du processus,
  . **nb_voisins**(< *out* >) est un entier indiquant le nombre de voisins,
  . **code** (< *out* >) est le code d'erreur.
. La fonction **MPI_GRAPH_NEIGHBORS**(*comm_nouveau*, *rang*, *nb_voisins*, *voisins*, *code*) donne la liste des processus voisins pour un processus donné.
  . **comm_nouveau** (< *in* >) est un entier indiquant le nombre de processus,
  . **rang** (< *in* >) est un entier indiquant le rang du processus,
  . **nb_voisins** (< *in* >) est un entier indiquant le nombre de voisins,
  . **voisins** (< *out* >) est tableau d'entiers de taille le nombre de voisins total, contenant la liste des voisins,
  . **code** (< *out* >) est le code d'erreur.

6.2.3 Exemple de l'Idris : propagation d'un feu de forêt

```
program feu_graph
use MPI

implicit none
integer                                 :: rang,code,comm_graph,nb_voisins,i,iteration=0
integer, parameter                      :: etiquette=100,nb_procs=6
integer, dimension(nb_procs)            :: index
integer, dimension(16)                  :: liste_voisins
integer, dimension(:),allocatable       :: voisins
integer, dimension(MPI_STATUS_SIZE)     :: statut
logical                                 :: reorganisation
real                                    :: propagation,feu=0., & !Propagation du feu,Valeur du feu
                                           bois=1.,       & ! Rien n'a encore brule
                                           arret=1.         ! Tout a brule si arret <= 0.01

!initialisation MPI
call MPI_INIT(code)

!on definit les voisins de chaque parcelle
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/)

!creation topologie de graphe
call MPI_GRAPH_CREATE(MPI_COMM_WORLD,nb_procs,index,liste_voisins,reorganisation,comm_graph,code)
call MPI_COMM_RANK(comm_graph,rang,code)!rang du processus dans nouveau communicateur

!declaration de feu dans la parcelle 2
if(rang==2) feu=1

!determination du nombre de voisins + liste
call MPI_GRAPH_NEIGHBORS_COUNT (comm_graph,rang,nb_voisins,code)
allocate(voisins(nb_voisins)) ! Allocation du tableau voisins
call MPI_GRAPH_NEIGHBORS (comm_graph,rang,nb_voisins,voisins,code)

do while (arret>0.01) ! critere d'arret : quand il n'y a plus rien a brule
   !propagation du feu aux voisins
   do i=1,nb_voisins
      call MPI_SENDRECV (min(1.,feu),1,MPI_REAL,voisins(i),etiquette,&
           propagation, 1, MPI_REAL ,voisins(i),etiquette,comm_graph,statut,code)
      feu=1.2*feu +0.2*propagation*bois !loi de propagation du feu
      bois=bois/(1.+feu) !quantite de bois restant par parcelle
   end do
   !determination de la quantite de bois restant sur l'ensemble  de la foret
   call MPI_ALLREDUCE(bois,arret,1,MPI_REAL,MPI_SUM,comm_graph,code)
   iteration=iteration+1
   print*,"Iteration ",iteration," parcelle ",rang," bois=",bois
   call MPI_BARRIER (comm_graph,code)
   if (rang == 0) print*,"----------"
end do
deallocate(voisins)

call MPI_FINALIZE(code)   !desallocation MPI

end program feu_graph
```

FIG. 28: Programme en $F90$ calculant la propagation d'un feu de forêt de parcelle en parcelle en utilisant une topologie de type graphe. (programme issu du cours de l'Idris)

```
-bash-2.05b$ mpiexec -n 6 /users/uppa/delage/test/feu_graph
 Iteration  1  parcelle  0  bois= 1.000000000
 Iteration  1  parcelle  1  bois= 0.6720429659
 Iteration  1  parcelle  2  bois= 0.6803275645E-01
 Iteration  1  parcelle  3  bois= 0.6017233729
 Iteration  1  parcelle  4  bois= 0.5886827707
 Iteration  1  parcelle  5  bois= 0.9526526928
 ----------
 Iteration  2  parcelle  0  bois= 0.9541984797
 Iteration  2  parcelle  1  bois= 0.1598708928
 Iteration  2  parcelle  2  bois= 0.1570278313E-02
 Iteration  2  parcelle  3  bois= 0.1597269773
 Iteration  2  parcelle  4  bois= 0.1509699374
 Iteration  2  parcelle  5  bois= 0.6900082827
 ----------

....

 Iteration  10  parcelle  0  bois= 0.7964182645E-02
 Iteration  10  parcelle  1  bois= 0.1894458834E-38
 Iteration  10  parcelle  2  bois= 0.2011066485E-38
 Iteration  10  parcelle  3  bois= 0.1619766026E-23
 Iteration  10  parcelle  4  bois= 0.1181390006E-23
 Iteration  10  parcelle  5  bois= 0.2635534679E-08
 ----------
```

FIG. 29: Affichage à l'écran des impressions du Programme de la figure 28. Le programme s'arrête au bout de 10 itérations lorsque le critère d'arrêt est vérifié.

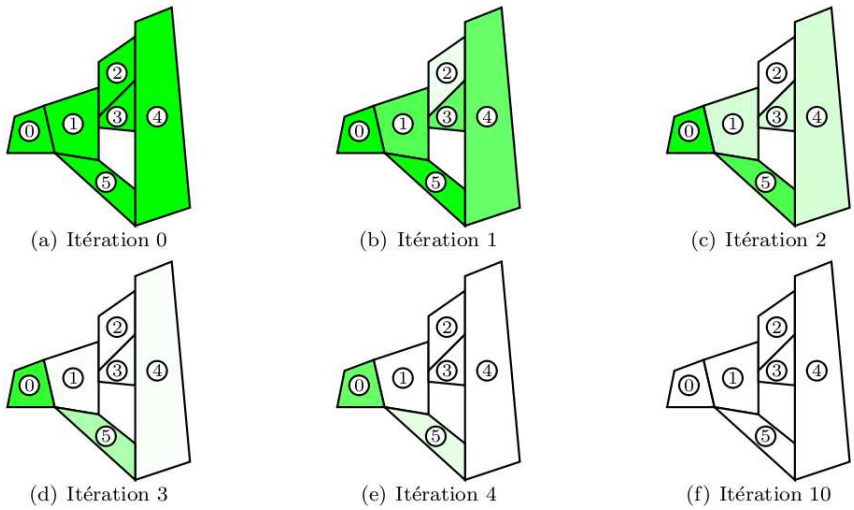


FIG. 30: Visualisation de la propagation du feu de parcelle en parcelle (figure cours Idris).

7 Communicateurs

7.1 Introduction

Un communicateur est constitué d'un ensemble de processus et d'un contexte de communication. Ce contexte, mis en place lors de la construction du communicateur, permet de délimiter l'espace de communication et est géré par MPI.
Par défaut, c'est le communicateur **MPI_COMM_WORLD** qui est créé lors de l'initialisation de l'environnement MPI. Celui-ci contient tous les processus actifs et c'est au sein de cet ensemble qu'on effectue des opérations de communication.

Cet ensemble de processus peut être partitionné en plusieurs sous-ensembles de processus au sein desquels on pourra effectuer des opérations MPI. Ainsi, chaque sous-ensemble dispose de son propre espace de communication qui correspond en fait à un communicateur.

Les communicateurs peuvent-être gérés de manière dynamique. Ils sont créés en utilisant les fonctions **MPI_CART_CREATE()**, **MPI_CART_SUB()**, **MPI_COMM_CREATE()**, **MPI_COMM_DUP()** ou **MPI_COMM_SPLIT()**. Ils sont détruits via la fonction **MPI_COMM_FREE()**.
On peut se demander quelle est l'utilité de créer un communicateur contenant un sous-ensemble de processus. Supposons qu'on dispose d'un ensemble de processus dans le communicateur MPI_COMM_WORLD. On souhaite effectuer des communications seulement entre les processus de rang pair. Une possibilité est de faire des tests (boucle if) sur le rang de chaque processus, mais cela devient vite coûteux..... Une autre possibilité consiste à regrouper les processus de rang pair dans un nouveau communicateur, ce qui supprime les tests (pourvu qu'on précise que les communications se font dans le nouveau communicateur).

7.2 Communicateur issu d'un autre

La fonction **MPI_COMM_SPLIT**($comm$, $couleur$, $clef$, $nouveau\_comm$, $code$) partitionne un communicateur donné en autant de communicateurs voulus :

- **comm** ($< in >$) est un entier désignant le communicateur de départ (qu'on veut partitionné),
- **couleur** ($< in >$) est un entier. Une *couleur* est affectée à chaque processus appartenant au même sous-espace de communication.

. **clef** (< *in* >) est un entier qui permet à MPI d'affecter un rang aux processus appartenant au même sous-espace de communication. Il s'agit d'une numérotation locale. MPI attribue les rangs suivant des valeurs de clefs croissantes. Il est conseillé de mettre la clef la plus petite sur le processus détenant l'information à distribuer (il aura ainsi le rang 0 dans la numérotation locale).
. **nouveau_comm** (< *out* >) est un entier désignant le nouveau communicateur.
. **code** (< *out* >) est un entier donnant le code d'erreur.

Un processus à qui on attribue une couleur **MPI_UNDEFINED** n'appartient qu'à son communicateur initial.

Exemple : construction d'un communicateur qui partage l'espace de communication initial entre processus de rangs pairs et impairs via le constructeur **MPI_COMM_SPLIT()** :

processus	$P0$	$P1$	$P2$	$P3$	$P4$	$P5$
rang_global	0	1	2	3	4	5
couleur	10	20	10	20	10	20
clef	6	1	0	2	5	0
rang_local	2	1	0	2	1	0

TAB. 6: Construction du communicateur $Comm\_pair\_impair$ avec MPI_COMM_SPLIT.

7.3 Subdivision de topologie

L'idée consiste à dégénérer une topologie cartésienne de processus de dimension d'espace dim en une topologie cartésienne de dimension $(dim - 1)$. Pour dégénérer une topologie cartésienne de dimension dim, il suffit de créer des communicateurs de dimension $(dim - 1)$. Pour une topologie cartésienne de dimension $dim = 2$, la dégénérescence se traduit par la création d'autant de communicateurs qu'il y a de lignes, par exemple (voir figure 32). Les transferts d'information se font alors au sein de chaque nouveau communicateur.
L'intérêt réside dans le fait qu'on peut effectuer des communications collectives restreintes à un sous ensemble de processus appartenant à la topologie dégénérée. On peut dégénérer une topologie cartésienne soit à partir de la fonction **MPI_COMM_SPLIT()** (détaillée précédement)

```
program PairsImpairs

  use mpi

  implicit none
  integer, parameter  :: m=3
  integer             :: clef,couleur,CommPairsImpairs
  integer             :: rang_global,code,rang_local
  real, dimension(m)  :: a


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

  ! Initialisation du vecteur A
  a(:)=0.
  if(rang_global == 2) a(:)=2.
  if(rang_global == 5) a(:)=5.

  if (mod(rang_global,2) == 0) then
     !Couleur et clefs des processus pairs
     couleur = 0
     if (rang_global == 2) then
        clef = 0
     else
        clef = rang_global + 1
     end if
  else
     !Couleur et clefs des processus impairs
     couleur = 1
     if (rang_global == 5) then
        clef = 0
     else
        clef = rang_global
     end if
  end if

  !Creation communicateurs pair et impair en leur donnant 1 meme denomination
  call MPI_COMM_SPLIT ( MPI_COMM_WORLD ,couleur,clef,CommPairsImpairs,code)
  call MPI_COMM_RANK ( CommPairsImpairs,rang_local,code)
  print*,'processus : rang global ',rang_global,', rang local',rang_local

  !Diffusion message par le processus 0 de chaque communicateur aux processus
  ! de son groupe
  call MPI_BCAST (a,m, MPI_REAL ,0,CommPairsImpairs,code)
  print*,'dans le comm local mon rang est :',rang_local,', a=',a

  ! Destruction des communicateurs
  call MPI_COMM_FREE (CommPairsImpairs,code)
  call MPI_FINALIZE (code)

end program PairsImpairs
```

FIG. 31: Programme en *F*90 illustrant le partage de l'espace de communication initial entre processus de rangs pairs et impairs.

soit à partir de la fonction **MPI_CART_SUB**(*CommCart*, *Subdivision*, *CommCartD*, *code*) où :

. **CommCart** (< *in* >) est un entier désignant une topologie cartésienne,
. **Subdivision** (< *in* >) est un tableau d'entiers de dimension le nombre de communicateurs qu'on veut créer,
. **CommCartD** (< *out* >) est un entier désignant la topologie cartésienne dégénérée,
. **code** (< *out* >) est un entier désignant le code d'erreur.

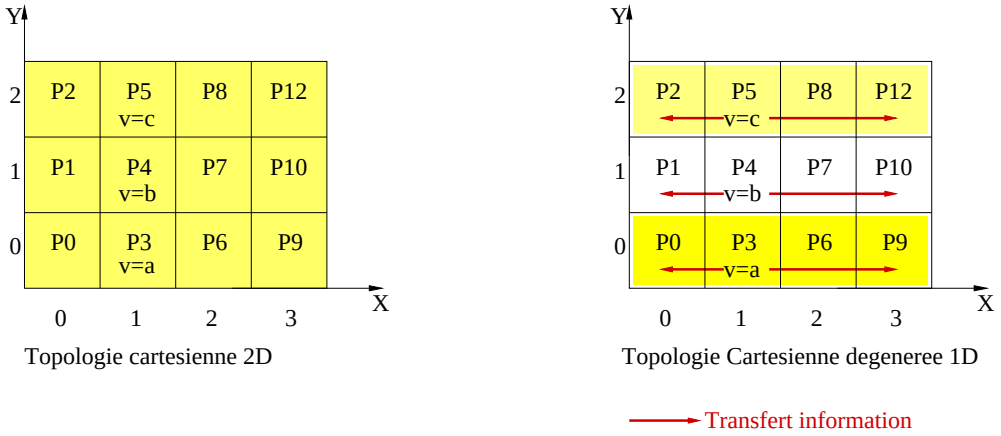


FIG. 32: Topologie cartésienne $2D$ dégénérée en topologie cartésienne $1D$ (subdivision en ligne).

7.4 Intra et intercommunicateur

Les intracommunicateurs sont des communicateurs qui effectuent des opérations de communication entre processus d'un même communicateur. Mais les communications entre communicateurs sont impossibles. Jusqu'à présent, c'est ce type de communicateur qu'on a construit.
Un intercommunicateur est un communicateur qui permet d'établir des communications entre intracommunicateurs. Attention, dans le $MPI-1$ seules les communications point à point sont permises. Pour créer un intercommunicateur, on utilise la fonction
MPI_INTERCOMM_CREATE().

```
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
  real                        :: W=0.

  call MPI_INIT (code)

  ! Creation de la grille 2D initiale
  Dim2D(1) = 4
  Dim2D(2) = 3
  Periode(:) = .false.
  ReOrdonne = .false.
  V(:)=0.
  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(:)=real(rang)

  ! Chaque ligne de la grille doit etre une topologie cartesienne 1D
  Subdivision(1) = .true.
  Subdivision(2) = .false.
  ! Subdivision de la grille cartesienne 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 : ",rang," ; Coordonnees : (",Coord2D(1),",",Coord2D(2),") ; W = ",W

  call MPI_FINALIZE (code)

end program CommCartSub
```

FIG. 33: Programme dégénérant une topologie cartésienne $2D$ en une topologie cartésienne $1D$ avec la fonction $MPI\_CART\_SUB()$.

```
program CommCartSplit

  use mpi

  implicit none

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

  call MPI_INIT (code)

  ! Creation de la grille 2D initiale
  Dim2D(1) = 4
  Dim2D(2) = 3
  Periode(:) = .false.
  ReOrdonne = .false.
  V(:)=0.
  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(:)=real(rang)

  ! Chaque ligne de la grille doit etre une topologie cartesienne 1D
  couleur=Coord2D(2)
  clef=Coord2D(1)
  call MPI_COMM_SPLIT(Comm2D,couleur,clef,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 : ",rang," ; Coordonnees : (",Coord2D(1),",",Coord2D(2),") ; W = ",W

  call MPI_FINALIZE (code)

end program CommCartSplit
```

FIG. 34: Programme dégénérant une topologie cartésienne $2D$ en une topologie cartésienne $1D$ avec la fonction $MPI\_COMM\_SPLIT()$.

8 Conclusion

La bibliothèque MPI permet de faire du calcul parallèle en se basant sur le principe d'échanges de messages (communications) entre processus, chacun ayant sa propre mémoire. Ces opérations de communications peuvent être collectives ou point à point.
Dans un code de calcul parallèle, si le temps de communication devient prépondérant devant celui de calcul, on peut optimiser le temps de communication grâce aux différents modes de transfert de messages proposés par MPI ainsi qu'en recouvrant les communications par des calculs.
MPI permet aussi de créer des topologies cartésiennes ou de type graphes (pour des géométries de domaines plus complexes), ce qui est très pratique lorsqu'on travaille sur des domaines de calcul ou qu'on souhaite faire de la décomposition de domaine.
MPI permet aussi de partitionner un ensemble de processus en plusieurs sous-ensembles de processus au sein desquels on pourra effectuer des opérations de communication. Ainsi, chaque sous-ensemble dispose de son propre espace de communication ou communicateur.
Dans une nouvelle version de la bibliothèque MPI ($MPI-2$), on note deux évolutions principales : la gestion dynamique des processus et les Entrées−Sorties parallèles.