Introduction à M.P.I. en FORTRAN

Nicolas KIELBASIEWICZ *

17 novembre 2009

M.P.I. signifie *Message Passing Interface*. C'est une bibliothèque disponible en C, C++ et FORTRAN qui permet dans un code parallèle ou réparti de gérer les échanges de messages entre les différents processeurs (ou nœuds). Toutes les informations concernant cette librairie sont disponibles sur le site :

http ://www.mpi.org/

L'objectif de ce petit document n'est pas de donner une liste exhaustive des diverses fonctionnalités de la librairie M.P.I.— la documentation disponible sur le lien précédent est faite pour ça —, mais plutôt de donner un aperçu des fonctions les plus couramment utilisées, ainsi que les fonctionnalités que j'ai personnellement eu du mal à trouver dans les docs disponibles sur le net, dans l'esprit d'un aide-mémoire. Il ne sera pas question ici non plus de présenter MPI-2. Par ailleurs, seules les syntaxes en Fortran seront données, avec le type de chacun des arguments. Sous cette forme, il y a un argument supplémentaire `ierr` qui désigne la variable d'erreur. Elle n'existe pas en C/C++.

Table des matières

1 L'environnement de base 2
- 1.1 Que faire pour pouvoir écrire des commandes M.P.I. dans un code ? 2
- 1.2 Initialisation / Finalisation 2
- 1.3 Prendre des mesures de temps 2

2 Communicateurs et topologies 3
- 2.1 Définir un communicateur 3
- 2.2 Rang et taille d'un communicateur 3
- 2.3 Subdiviser un communicateur 3
- 2.4 Créer une topologie cartésienne 3
- 2.5 Coordonnées, rang et voisins dans une topologie cartésienne 4

3 Types de données 4
- 3.1 Types de base 4
- 3.2 Types dérivés 5

4 Communications point à point 5
- 4.1 Communications bloquantes 5
- 4.2 Communications non bloquantes 7
- 4.3 Communications persistantes 8
  - 4.3.1 Définition 8
  - 4.3.2 Équivalences entre communications persistantes et communications non bloquantes 8

*Unité de Mathématiques Appliquées, École Nationale Supérieure de Techniques Avancées

5 Communications collectives **9**
5.1 Envois et réceptions collectifs 9
5.2 Opérations de réduction 9

6 Et en C, comment ça marche ? **10**

1 L'environnement de base

1.1 Que faire pour pouvoir écrire des commandes M.P.I. dans un code ?

Quel que soit le langage, il suffit de charger un fichier d'en-tête qui s'appelle mpif.h. Eventuellement, il faudra préciser à la compilation où se trouve ce fichier.

Fortran90 et ses successeurs proposent une autre solution : utiliser le module mpi à l'aide de la commande use mpi. La seule contrainte est que le module doit avoir été compilé avec le compilateur que vous utilisez pour votre propre code de calcul. Le cas échéant, il faudra le faire soi-même, ce qui n'est pas très compliqué.

Cette solution propose un avantage indéniable du fait que M.P.I. est considéré ici comme un module : la vérification de type et de syntaxe, évitant ainsi les conversions de type implicites, comme entre les entiers et les booléens. C'est la raison pour laquelle non seulement la syntaxe Fortran qui donnée dans ce document sera celle de Fortran90, respectant ainsi le type véritable de chacun des arguments, mais aussi la solution que je vous conseille d'utiliser, de par cette rigueur syntaxique imposée, les compilateurs Fortran actuels gérant tout aussi bien Fortran77 que Fortran90.

1.2 Initialisation / Finalisation

Pour autoriser l'utilisation des commandes M.P.I., il faut que celles-ci soient comprises entre l'appel de deux routines particulières, dont la syntaxe est :

```
MPI_INIT(ierr)
        integer, intent(out) :: ierr
MPI_FINALIZE(ierr)
        integer, intent(out) :: ierr
```

Toute commande M.P.I. appelée en dehors des deux appels de ces deux routines ou l'absence de l'un de ces deux appels donnera une erreur de compilation.

1.3 Prendre des mesures de temps

Pour mesurer les performances de vos algorithmes en terme de vitesse d'exécution, M.P.I. propose une fonction permettant de récupérer le temps CPU, et ainsi de pouvoir évaluer le temps de calcul par différence.

```
double precision MPI_WTIME()
```

Il suffit donc d'appeler cette commande au début de votre algorithme et à la fin, puis de faire la différence entre les deux valeurs. Une remarque cependant : pour que ces mesures de temps aient un sens, il fait que chaque processus l'exécute en même temps. On précédera donc tout appel de `MPI_WTIME` par l'appel de routine de synchronisation de tous les processeurs d'un communicateur donné :

```
MPI_BARRIER(comm, ierr)
        integer, intent(in) :: comm
        integer, intent(out) :: ierr
```

Cette notion de communicateur fait l'objet de la section suivante.

2 Communicateurs et topologies

2.1 Définir un communicateur

Le communicateur est l'environnement dans lequel les envois et réceptions de données entre différents nœuds ont lieu. C'est donc un environnement dans lequel sera appelé les routines M.P.I.. Par défaut, il existe un communicateur utilisable avec la variable `MPI_COMM_WORLD`. Mais M.P.I. propose de définir de manière simple un communicateur, à l'aide de la commande suivante :

```
MPI_COMM_CREATE(comm, newgroup ,newcomm, ierr )
        integer , intent(in) :: comm, newgroup
        integer , intent(out) :: newcomm, ierr
```

On lui donne dans l'ordre le communicateur source existant, le numéro du groupe à créer et le nouveau communicateur à créer. Il existe d'autres façons de définir des communicateurs.

2.2 Rang et taille d'un communicateur

Dans un communicateur, chaque nœud de calcul (ou chaque processus) est identifié par un entier, son rang. À partir du moment où l'on peut définir plusieurs communicateurs, ce rang peut avoir une valeur différente dans des communicateurs différents. M.P.I. permet alors de récupérer le rang d'un nœud dans un communicateur donné, ainsi que le nombre de nœuds d'un communicateur donné :

```
MPI_COMM_RANK(comm, rank , ierr )
        integer , intent(in) :: comm
        integer ,intent(out) :: rank , ierr
MPI_COMM_SIZE(comm, nprocs , ierr )
        integer , intent(in) :: comm
        integer , intent(out) :: nprocs , ierr
```

2.3 Subdiviser un communicateur

M.P.I. permet de définir un nouveau communicateur par subdivision d'un autre communicateur suivant un critère donné (appelé couleur) :

```
MPI_COMM_SPLIT (comm, couleur , rank , nouveau_comm , ierr )
integer , intent(in) :: comm, couleur , rank
integer , intent(out) :: nouveau_comm , ierr
```

On crée ainsi un communicateur pour chaque valeur possible de la couleur. L'exemple typique est la création des communicateurs contenant respectivement les nœuds de rang pair et les nœuds de rang impair. La couleur sera alors le rang modulo 2.

2.4 Créer une topologie cartésienne

Dans un communicateur, on aimerait pouvoir maîtriser la numérotation des nœuds. Pour cela on peut définir un une topologie cartésienne à `ndims`-dimensions à l'aide de la routine suivante :

```
MPI_CART_CREATE(comm, ndims , dims , periods , reorder ,comm_cart , ierr )
        integer , intent(in) :: comm, ndims
        integer , dimension(ndims), intent(in) :: dims
        logical , dimension(ndims), intent(in) :: periods
        logical , intent(in) :: reorder
        integer , intent(out) :: comm_cart , ierr
```

Le tableau `dims` précise le nombre de nœuds dans chacune des dimensions, tandis que le tableau `periods` précise si la topologie est périodique dans une ou plusieurs dimensions données.

Le booléen `reorder` sert à autoriser/interdire le réordonnancement des nœuds entre eux par M.P.I.. Dans la grande majorité des cas, vous souhaiterez l'interdire en l'initialisant à `.false.` .

Remarque. La création d'une topologie cartésienne dans un communicateur s'accompagne de la création d'un nouveau communicateur.

2.5 Coordonnées, rang et voisins dans une topologie cartésienne

Puisque le communicateur que l'on vient de créer est muni d'une topologie cartésienne, la notion de coordonnées d'un nœud a un sens dans ce communicateur. On récupèrera les coordonnées d'un nœud donné (pas nécessairement le nœud courant) à l'aide de la routine :

```
MPI_CART_COORDS(comm, rank, ndims, coords, ierr)
integer, intent(in) :: comm, rank, ndims
integer, dimension(ndims), intent(out) :: coords
integer, intent(out) :: ierr
```

À l'inverse, il est possible à partir des coordonnées d'un nœud de retrouver son rang à l'aide de la routine :

```
MPI_CART_RANK(comm_cart, coords, rank,ierr)
        integer, intent(in) :: comm_cart
        integer, dimension(ndims), intent(in) :: coords
        integer, intent(out) :: rank, ierr
```

Si cela concerne le nœud courant, rien n'empêche toutefois d'utiliser `MPI_COMM_RANK`.

De même, la topologie cartésienne permet d'introduire la notion de voisins à une distance donnée dans un direction donnée et de les récupérer à l'aide de la routine :

```
MPI_CART_SHIFT(comm_cart, direction, distance, rank_previous, rank_next,
   ierr)
        integer, intent(in) :: comm_cart, direction, distance
        integer, intent(out) :: rank_previous, rank_next, ierr
```

Si le voisin n'existe pas, alors le rang sera initialisé à -1, ou `MPI_PROC_NULL`, qui est la variable consacrée (et vaut -1).

3 Types de données

Lors des communications, nous allons devoir préciser de quel type de données il s'agit. Pour les types usuels, il existe des types de données M.P.I..

3.1 Types de base

Tous les types de base FORTRAN ont leur correspondance en terme de type de donnée M.P.I..

Type FORTRAN	Type M.P.I.
integer	mpi_integer
real	mpi_real
double precision	mpi_double_precision
complex	mpi_complex
logical	mpi_logical
character	mpi_character

3.2 Types dérivés

Tout comme en FORTRAN, il est possible en M.P.I. de définir ses propres types de données. Par exemple, on souhaiterait définir un type de donnée permettant d'extraire des blocs d'éléments non continus en mémoire mais régulièrement espacés. Imaginez par exemple que vous voulez envoyer une ligne ou un bloc de lignes d'un tableau 2D, en se rappelant qu'en FORTRAN, les éléments d'un tableau 2D sont rangés dans l'ordre des colonnes. En M.P.I., vous pouvez réaliser cette opération en définissant un type dérivé à l'aide des routines :

```
MPI_TYPE_VECTOR(Nelem, tailleBloc, decalage, typeElem, nouveauType,ierr)
        integer, intent(in) :: Nelem, tailleBloc, decalage, typeElem
        integer, intent(out) :: nouveauType, ierr
MPI_TYPE_COMMIT(nouveauType,ierr)
        integer, intent(in) :: nouveauType
        integer, intent(out) :: ierr
```

`MPI_TYPE_VECTOR` permet de définir son type de données tandis que `MPI_TYPE_COMMIT` permet de publier un type que vous avez défini dans l'environnement M.P.I., vous permettant ainsi de l'utiliser à loisirs comme un type de base dans vos communications.

Reprenons notre exemple de lignes d'un tableau 2D. Considérons une matrice rectangulaire entière contenant 5 lignes et 6 colonnes, et que vous souhaitez par exemple pouvoir échanger 3 lignes complètes. Puisque les éléments d'une colonne sont continus en mémoire, vous allez donc définir un type de donnée que vous appelerez par exemple `LIGNE` de la manière suivante :

```
call MPI_TYPE_VECTOR(6,3,5,MPI_INTEGER, LIGNE, ierr)
```

Il s'agit en effet de blocs de 3 éléments (3 lignes), qu'il y a 6 blocs à assembler (parce que 6 colonnes) et que ces blocs sont espacés en mémoire de 5 (nombre de lignes).

Remarque. En FORTRAN, il est inutile de définir un type colonne parce que les éléments sont continus en mémoire. Si vous le souhaitez toutefois, vous avez 2 possibilités. Dans notre exemple, vous pouvez définir une colonne comme un bloc de 5 éléments, ou comme 5 blocs de 1 élément et espacés de 1.

Nous allons donc maintenant aborder le cœur de la librairie M.P.I., à savoir la façon de communiquer des données entre nœuds.

4 Communications point à point

On appelle communication « point à point » toute communication entre exactement 2 nœuds. On l'appelle aussi « communication de un à un ». Ces communications permettent d'envoyer une ou plusieurs données d'un type M.P.I. donné à condition qu'elles soient continues en mémoire, un tableau de doubles, un paquet de lignes, ... Il en existe plusieurs types.

4.1 Communications bloquantes

Une communication est dite bloquante lorsqu'elle empêche la poursuite de l'exécution d'un programme tant que la communication n'est pas entièrement réalisée. C'est le comportement assez classique de toute instruction en FORTRAN, exécutée de manière séquentielle. Les routines permettant d'effectuer des communications bloquantes sont :

```
MPI_RECV(buf,count,datatype,source,tag,comm,status,ierr)
        integer, intent(in) :: source, count, datatype, tag, comm
```

```
        <type>, dimension(:), intent(out) :: buf
        integer, dimension(MPI_STATUS_SIZE), intent(out) :: status
        integer, intent(out) :: ierr
MPI_SEND(buf,count,datatype,dest,tag,comm,ierr)
        <type>, dimension(:), intent(in) :: buf
        integer, intent(in) :: dest, count, datatype, tag, comm
        integer, intent(out) :: ierr
MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,recvbuf,recvcount,
    recvtype,source,recvtag,comm,status,ierr)
        <type>, intent(in) :: sendbuf(*)
        integer, intent(in) :: dest, source, sendcount, recvcount, sendtype
            , recvtype, sendtag, recvtag, comm
        <type>, dimension(:), intent(out) :: recvbuf
        integer, dimension(MPI_STATUS_SIZE), intent(out) :: status
        integer, intent(out) :: ierr
```

Leurs noms sont assez explicites et leurs syntaxe assez proche. Quelle est la signification de chacun des arguments ?

buf, count, datatype : Ces 3 arguments permettent de spécifier le nombre de données envoyées (count), leur type (datatype) au sens de M.P.I., et qui est le premier élément envoyé / reçu (<type> désigne le type FORTRAN de la variable **buf**). C'est la raison pour laquelle on peut donner le tableau complet ou uniquement le premier élement d'un tableau en guise de premier argument.

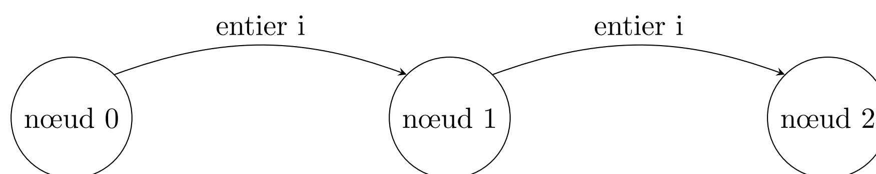
source / dest : rang du nœud qui envoie / reçoit, celui qui effectue la communication étant le nœud appelant la routine.

tag : permet de donner un numéro d'identification à la communication. L'intérêt est de pouvoir distinguer plusieurs envois ou réceptions d'un nœud à un autre (voir l'exemple ci-dessous).

comm : le communicateur dans lequel la communication est faite.

status : tableau donnant des informations sur le déroulement de l'envoi / réception.

La routine `MPI_SENDRECV` est en fait un `MPI_SEND` suivi d'un `MPI_RECV`. C'est une commande très utile car elle correspond à une bonne organisation des communications. Pour bien comprendre son rôle, traitons un cas simple : celui d'un envoi d'une donnée entière i du nœud 0 au nœud 1, et de la même donnée entière i du nœud 1 au nœud 2, schématisée ci-dessous.



Ces deux communications sont de même nature et peuvent être traitée simultanément.

Comment procéder ?

Si je regarde ces communications de leur point de départ, je vais considérer qu'il s'agit d'un envoi d'un entier à mon voisin de droite `rank_next`. SI maintenant je les regarde de leur point d'arriver, je dirai qu'il s'agit d'une réception d'un entier de mon voisin de gauche `rank_previous`. Rappelons que es deux voisins peuvent être définis à l'aide de `MPI_CART_SHIFT` que nous avons vu plus tôt.

Cette même communication peut donc être vue comme un envoi ou une réception. Il s'agit en fait des deux. C'est la raison pour laquelle je vais traiter les deux en même temps, en écrivant :

```
...
call MPI_SEND(i,1 MPI_INTEGER,rank_next,100,MPI_COMM_WORLD,ierr)
call MPI_RECV(i,1 MPI_INTEGER,rank_previous,100,MPI_COMM_WORLD,ierr)
...
```

Remarque. La valeur du tag dans les deux appels est la même, puisqu'il s'agit de la même communication. C'est là la signification de cet argument, à savoir coupler un envoi à la réception qui lui correspond, permettant ainsi d'identifier de manière unique tout échange de données.

Le nœud 2 va lui aussi envoyer l'entier i à son voisin de droite. Hors, celui-ci n'existe pas (son rang sera -1), l'envoi n'aura donc pas lieu. De même, 0 n'ayant pas de voisin de gauche, il n'en recevra pas de données.

Ces deux éléments nous permettent de comprendre qu'en dépit du bon sens, nous aurions très bien pu écrire la réception avant l'envoi. Je dis « en dépit du bon sens », car tout nœud n'a besoin d'attendre quoi que ce soit pour envoyer des données à son voisin de droite, alors qu'il doit attendre que son voisin de gauche ait envoyé des données pour qu'il les reçoive.

En pratique, vous écrirez donc toujours un appel de `MPI_SEND` suivi d'un appel de `MPI_RECV`. La commande `MPI_SENDRECV` permet de regrouper ces deux appels.

Remarque. Il n'existe pas de routine `MPI_RECVSEND`, la logique voulant que l'envoi ait lieu avant la réception, comme nous venons de le voir.

4.2 Communications non bloquantes

Les communications non bloquantes laissent le programme poursuivre son exécution, que la communication soit réellement faite ou non. Elle présentent donc un avantage indéniable en terme de temps d'exécution puisqu'un nœud peut travailler tout en envoyant des données. Les routines liées à des communications non bloquantes sont :

```
MPI_IRECV(buf,count,datatype,dest,tag,comm,request,ierr)
        integer, intent(in) :: count, datatype, dest, tag, comm
        <type>, dimension(:), intent(out) :: buf
        integer, intent(out) :: ierr
MPI_ISEND(buf,count,datatype,dest,tag,comm,ierr)
        <type>, dimension(:), intent(in) :: buf
        integer, intent(in) :: dest, count, datatype, tag, comm
        integer, intent(out) :: ierr
MPI_WAIT(request,status,ierr)
        integer, intent(in) :: request
        integer, dimension(MPI_STATUS_SIZE), intent(out) : status
        integer, intent(out) :: ierr
MPI_WAITALL(count,request_tab,status_tab,ierr)
        integer, dimension(:), intent(in) :: request_tab
        integer, intent(in) :: count
        integer, dimension(MPI_STATUS_SIZE,:), intent(out) :: status_tab
        integer, intent(out) :: ierr
```

Les deux dernières commandes permettent d'attendre que les communications non bloquantes aient été effectuées avant de poursuivre l'exécution. A utiliser par exemple lorsque les données échangées doivent être utilisées.

Les arguments ont la même signification que pour les communications bloquantes. Seul l'entier `request` est inédit, il contient des informations sur la communication elle-même. C'est cet argument qui est utilisé pour les routines `MPI_WAIT` et `MPI_WAITALL`.

Remarque. Puisque `request` contient des informations liées à une commande de communication non bloquante, et qu'il est possible de passer un tableau en argument de `MPI_WAITALL`, je vous conseille de définir `request` comme un tableau d'entiers de rang 1 et de taille le nombre de réceptions non bloquantes effectuées par votre code.

4.3 Communications persistantes

4.3.1 Définition

Les communications persistantes sont une autre façon d'écrire des communications non bloquantes. En effet, on définit les communications à effectuer, mais sans les exécuter. On dispose alors de commandes spécifiques pour lancer l'exécution de ces communications.

```
MPI_RECV_INIT(buf,count,datatype,source,tag,comm,request,ierr)
        integer, intent(in) :: count, datatype, source, tag, comm
        <type>, dimension(:), intent(out) :: buf
        integer, intent(out) :: request, ierr
MPI_SEND_INIT(buf,count,datatype,dest,tag,comm,request,ierr)
        <type>, dimension(:), intent(in) :: buf
        integer, intent(in) :: count, datatype, dest, tag, comm
        integer, intent(out) :: request, ierr
MPI_START(request,ierr)
        integer, intent(in) :: request
        integer, intent(out) :: ierr
MPI_STARTALL(count, request_tab, ierr)
        integer, intent(in) :: count
        integer, dimension(:), intent(in) :: request_tab
        integer, intent(out) :: ierr
MPI_REQUEST_FREE(request,ier)
        integer, intent(in) :: request
        integer, intent(out) :: ierr
```

Cette dernière commande permet de libérer les communications persistantes, d'empêcher leur utilisation dans ce qui suit, un peu comme de la libération mémoire ou de la désallocation.

Remarque. Il est à noter que l'argument `request` apparaît aussi dans les envois. Fort de la remarque précédente, on fixera sa taille à l'ensemble des communications persistantes.

4.3.2 Équivalences entre communications persistantes et communications non bloquantes

Les communications persistantes étant non bloquantes, il existe des équivalences entre les deux syntaxes, que ce soit la commuication en elle-même, où l'attente de son exécution :

```
call MPI_IRECV(..., request, ier)
...
call MPI_WAIT(request, status, ier)
```

$\Longleftrightarrow$

```
call MPI_RECV_INIT(..., request, ier)
call MPI_START(request, ier)
...
call MPI_WAIT(request, status, ier)
call MPI_REQUEST_FREE(request, ier)
```

5 Communications collectives

Jusque là, nous avons vu les communications entre un nœud et un autre nœud. M.P.I. permet également des communications dites collectives, mettant en jeu d'une façon ou d'une autre tous les processeurs.

5.1 Envois et réceptions collectifs

```
MPI_BCAST(buffer, count, datatype, root, comm, ierr)
        integer, intent(in) :: count, datatype, root, comm
        <type>, dimension(:), intent(inout) :: buf
        integer, intent(inout) :: ierr
MPI_SCATTER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm
    ,ierr)
        <type>, dimension(:), intent(in) :: sendbuf
        integer, intent(in) :: sendcount, sendtype, recvcount, recvtype,
            root, comm
        <type>, dimension(:), intent(out) ::recvbuf
        integer, intent(out) :: ierr
MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,
    ierr)
        <type>, dimension(:), intent(in) :: sendbuf
        integer, intent(in) :: sendcount, sendtype, recvcount, recvtype,
            root, comm
        <type>, dimension(:), intent(out) ::recvbuf
        integer, intent(out) :: ierr
```

`MPI_BCAST` permet au nœud `root` de communiquer des données à tous les nœuds du communicateur.
`MPI_SCATTER` permet de décomposer en part égales de données et de distribuer ces parts à chacun des nœuds. `MPI_GATHER` permet d'effectuer l'opération inverse, c'est à dire concaténer des données de même nature envoyées à un même nœud.

Ces 3 routines permettent de mettre en œuvre des « communications de un à tous ».

Il existe également des communications dites de « tous à tous ». Par exemple, la généralisation de `MPI_GATHER` où le résultat concaténé sera récupéré par l'ensemble des nœuds :

```
MPI_ALLGATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,comm,
    ierr)
        <type>, dimension(:), intent(in) :: sendbuf
        integer, intent(in) :: sendcount, sendtype, recvcount, recvtype,
            comm
        <type>, dimension(:), intent(out) ::recvbuf
        integer, intent(out) :: ierr
```

5.2 Opérations de réduction

Il est également possible de faire ce que l'on appelle une opération de réduction en appelant :

```
MPI_REDUCE(sendbuf,recvbuf,count,datatype,operation,root,comm,ierr)
        <type>, dimension(:), intent(in) :: sendbuf
        integer, intent(in) :: count, datatype, root, operation, comm
        <type>, dimension(:), intent(out) :: recvbuf
        innteger, intent(out) :: ierr
```

La variable désignée par `operation` est parmi la liste suivante : MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR.

C'est la commande que l'on appellera lorsque l'on a parallélisé le calcul d'une somme et que l'on veut calculer la somme totale comme étant la somme des valeurs des sommes partielles calculées sur chaque nœud.

Là encore, il existe une version « tous à tous » appelée `MPI_ALLREDUCE` :

```
MPI_ALLREDUCE(sendbuf,recvbuf,count,datatype,operation,comm,ierr)
     <type>, dimension(:), intent(in) :: sendbuf
     integer, intent(in) :: count, datatype, operation, comm
     <type>, dimension(:), intent(out) :: recvbuf
     innteger, intent(out) :: ierr
```

Cette fois-ci, tous les nœuds disposeront du résultat de l'opération de réduction. Très utile par exemple pour calculer l'erreur quadratique d'une méthode itérative comme Jacobi, Gauss-Seidel, ...

6 Et en C, comment ça marche ?

La syntaxe de chacune des commandes M.P.I. est assez proche de celle en FORTRAN. Il y a essentiellement 3 différences :

1. l'argument `ierr` n'existe pas en C, il est remplacé par une valeur de retour à chacune des fonctions appelée.
2. les types de données M.P.I.sont beaucoup plus nombreux. Prenons l'argument `status` de `MPI_Recv`. En FORTRAN, c'est un entier, alors qu'en C, il est de type `MPI_Status`. De même, un nouveau type défini avec `MPI_TYPE_VECTOR` est de type `MPI_Datatype` en C alors qu'il est entier en FORTRAN.
3. le nom des routines est sensible à la casse en C. Hormis MPI, chaque mot utilisé dans le nom d'une routine commence par une majuscule.