Introduction à OpenMPI

Jérôme Lelong

Ensimag

Année 2013-2014

Plan

Pourquoi une librairie en plus d'OpenMP

- OpenMP s'adresse à des machines à mémoire partagée avec peu de processeurs
- MPI permet de programmer sur des architectures à mémoire distribuée (souvent des PCs reliés entre eux)
- La gestion des communications dans MPI se fait à la main et propose de manière générale plus fin qu'OpenMP sur la manière dont le code s'exécute

Les architectures cibles

- Les clusters de PC reliés entre eux par le réseau : typiquement quelques centaines de coeurs
- Les super calculateurs disposant de plusieurs niveaux de mémoire dont une partie partagée et une partie distribuée (par exemple la dernière machine Tera 100 de Bull avec $140,000$ coeurs)
- OpenMP perd en efficacité lorsque le nombre de coeurs augmentent
- On préfère regrouper les coeurs par petits paquets (une dizaine)
  - A l'intérieur d'un paquet : programmation type OpenMP
  - Entre 2 paquets : MPI

Qu'est-ce que MPI

- MPI = Message Passing Interface, un protocole d'échange de messages entre des machines reliées par un réseau.
- Permet la communication et la synchronisation de processus s'exécutant sur des machines différentes.
- La communication s'effectue grâce au protocole SSH

Compiler

- Il faut inclure `mpi.h`
- Compilateurs : mpicc ou mpic++. Ce sont des wrappers vers gcc ou g++ avec les bons flags automatiquement ajoutés.
  - Flags de compilation : mpicc -showme:compile
  - Flags de link : mpicc -showme:link ou mpic++ -showme:link
  - `mpicc` ≡ `gcc `mpicc -showme:compile``
  - `mpic++` ≡ `g++ `mpic++ -showme:compile``
  - La connaissance des flags est indispensable si l'on veut créer des librairies dynamiques avec libtool

```
CFLAGS=`mpicc -showme:compile`
CXXFLAGS=`mpicc -showme:compile`
LDFLAGS=`mpicc -showme:link`
```


Exécuter

Pour exécuter un programme compilés avec MPI dans un environnement parallèle : mpirun

- Définir comment communiquer à travers le réseau : export RSHCOMMAND=ssh (peut être mis dans $HOME/.bashrc)
- Implémentation OpenMPI

```
mpirun -machinefile fic -np X executable
```

`X` est le nombre de processeurs à utiliser. `fic` est un fichier du type

```
host1.example.com slots=X1 max_slots=Y1
host2.example.com slots=X2 max_slots=Y2
```

X : nb CPU sur la machine
Y : nb maximal de processus que MPI peut lancer sur la machine.
La partie `slots=X et max_slots=Y` est facultative.

Un programme de base

```
int main (int argc, char **argv)
{
int size, rank;
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
printf ("my rank is %d/%d\n", rank, size);
MPI_Finalize ();
exit (0);
}
```

Quelques conventions

- Les fonctions MPI commencent par le préfixe `MPI_` suivi d'une majuscule. Elles renvoient un code d'erreur entier égal à MPI_SUCCESS si tout s'est bien déroulé avec la convention

  0 = MPI_SUCCCESS < tous les autres codes

  Pour savoir à quelle erreur un code correspond

```
int MPI_Error_string(int errorcode, char *str, int *len)
```

- Les constantes MPI s'écrivent en majuscule

Performances I

- Comme pour les programmes OpenMP, on mesure la performance de l'implémentation en termes de speed-up par rapport à l'implémentation séquentielle.
- Les communications réduisent l'efficacité d'un code, il faut les minimiser.
- Pour mesurer le temps d'exécution d'une partie du code, utiliser double MPI_Wtime() qui renvoie un temps en secondes (c'est le temps de la pendule, pas un temps CPU)
- La mesure obtenue peut varier d'une exécution à l'autre de la charge des machines : on mesure un temps réel et non un temps CPU.

Plan

Les communicateurs

- Un communicateur est un ensemble de processus pouvant communiquer entre eux. MPI_COMM_WORLD est le communicateur par défaut regroupant tous les processus crées au lancement de l'application. MPI_COMM_WORLD est créé par MPI_Init.
- Pour communiquer entre eux, 2 processus doivent impérativement appartenir à un communicateur commun
- Un processus peut appartenir à plusieurs communicateurs et a un rang différent dans chacun d'eux.
- On peut créer des communicateurs à partir des routines `MPI_Group_range_excl` puis `MPI_Comm_create` par exemple.
- Pour obtenir la taille d'un communicateur et le rang d'un processus au sein de celui-ci

```
int MPI_Comm_size (MPI_Comm comm, int *size);
int MPI_Comm_rank (MPI_Comm comm, int *rank);
```

Les échanges de messages

Il existe différents types de communication

- Les communications bloquantes : un message envoyé doit être reçu.
  - Le processus qui initie l'envoi est bloqué tant que le message n'a pas été reçu.
  - La synchronisation entre les processus se fait naturellement.
  - Les messages ne peuvent se croiser : phénomène de *deadlocks*.
- Les communications non bloquantes :
  - L'exécution se poursuit sans attendre que le message ait été réceptionné
  - La synchronisation des processus doit être gérée manuellement
  - Les communications peuvent être recouvertes par des calculs.
  - Les messages peuvent se croiser.
- Les messages ne se doublent pas.

Les types MPI

type MPI	type C
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_CHAR	char
MPI_BYTE	entier positif 8-bit

Ces types assurent la portabilité du code entre des machines d'architectures différentes.

Spécification d'un message

- Contexte
  - source : rang du processus qui envoie
  - destinataire : rang du processus à qui le message est destiné
  - Tag : entier qui identifie le message envoyé.
  - le communicateur dans lequel on se trouve
- Message en lui-même
  - buffer contenant les données, c'est une adresse, un pointeur pas une référence.
  - count : nombre de données de type "datatype" à transmettre
  - datatype : c'est le type des données à transmettre. C'est **obligatoirement** un type MPI

MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

- `buf` : adresse de la donnée à envoyer
- `count` : le nombre d'éléments à envoyer
- `datatype` : le type de chaque élément (`MPI_INT`, `MPI_DOUBLE`,...)
- `dest` le rang du processeur à qui le messages est destiné
- `tag` : un entier pour marquer le message
- `comm` : le monde dans lequel la communication doit avoir lieu (pour nous `MPI_COMM_WORLD`)

MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

Exemples

```
/* Envoyer un entier a */
MPI_Send (&a,1,MPI_INT,dest,tag,comm);
/* Envoyer un tableau t de n entiers */
MPI_Send (t,n,MPI_INT,dest,tag,comm);
/* Envoyer un entier a NON portable */
MPI_Send (a,sizeof(int),MPI_BYTE,dest,tag,comm);
/* Envoyer un tableau t de n entiers NON portable */
MPI_Send (t,n*sizeof(int),MPI_BYTE,dest,tag,comm);
```


MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fonction bloquante. L'exécution est bloquée tant `buf` ne contient pas le nouveau message reçu. Attention, cela ne présage en rien que l'intégralité du message a effectivement été reçu.

MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- `buf` : adresse du buffer où écrire le message reçu. `buf` doit avoir été préalablement alloué.
- `count` : le nombre d'éléments à recevoir de type `datatype`
- `datatype` : le type MPI de chaque élément
- `source` : le rang du processeur dont on attend le message
- `tag` : un entier pour marquer le message
- `comm` : le monde dans lequel la communication doit avoir lieu (pour nous `MPI_COMM_WORLD`)
- `status` : objet MPI contenant des informations sur le message reçu.

MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Exemples

```
int a, *t;
/* Recevoir un entier a */
MPI_Recv (&a,1,MPI_INT,source,tag,comm,status);
/* Recevoir un tableau t de n entiers */
t = malloc (sizeof(int)*n);
MPI_Recv (t,n,MPI_INT,source,tag,comm,status);
/* Recevoir un entier a NON portable */
MPI_Recv (&a,sizeof(int),MPI_BYTE,source,tag,comm,status);
/* Recevoir un tableau t de n entiers NON portable */
t = malloc (sizeof(int)*n);
MPI_Recv (t,n*sizeof(int),MPI_BYTE,source,tag,comm,status);
```

MPI_Recv

- Un message ne peut être reçu que si les TAGs d'envoi et de réception sont identiques
- MPI_ANY_TAG : permet de recevoir un message quelque soit son TAG
- MPI_ANY_SOURCE : permet de recevoir un message de n'importe quel processus. Utilisé dans un schéma maître esclave par le maître pour se mettre en attente d'une réponse d'un des esclaves.
- Dans le cas de l'utilisation d'une valeur MPI_ANY_XXX, comment connaître sa vraie valeur ? grâce à la variable status qui contient les champs MPI_TAG et MPI_SOURCE.

Plusieurs modes d'émission de messages

- mode "standard" : MPI_Send
  Envoi bufferisé ou non suivant l'implémentation et l'OS. Peut être bloquante ou non. Ne rend la main que lorsque la variable à envoyer peut de nouveau être utilisée.
- mode "synchrone" : MPI_Ssend
  L'envoi est terminé dès qu'une opération de réception débute. Ceci ne signifie pas que le message a été reçu intégralement. N'utilise **pas** de buffer temporaire, c'est mieux que MPI_Send.
- mode "ready" : MPI_Rsend
  Ne peut être utilisé que si le processus destinataire a déjà appelé une fonction de réception. Fonction dangereuse à utiliser
- mode "buffer" : MPI_Bsend
  L'envoi utilise un buffer géré par le programmeur. Un buffer doit être attaché avec MPI_Buffer_attach. Rend la main dès que le message est copié dans le buffer de communication. Ne doit être utilisé qu'en cas de nécessité absolue.

La synchronisation

- Les communications bloquantes permettent de synchroniser un ou plusieurs processus.
- ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  Bloque l'exécution du processus appelant tant que tous les membres du communicateur ne l'ont pas appelée.

MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- Agrège les résultats de tous les processus au travers d'une opération (MPI_SUM, MPI_PROD, MPI_MAX, ...)
- L'utilisateur peur définir de nouvelles opérations avec MPI_Op_create et MPI_Op_free.
- Utilisation typique : le calcul d'une somme par différents processus
- Avec MPI_Allreduce, tous les processus reçoivent le résultat.

```
for(i=0; i<N; i++) myresult += data[i];
MPI_Reduce(&myresult, &result, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);
```

`result` contient la somme de toutes les sommes partielles `myresult`
```

La trame d'un programme Monte-Carlo sous MPI

```
MPI_Init(&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

PnlRng *rng;
rng = pnl_rng_dcmt_create_id(rank, seed_search);
pnl_rng_sseed(rng, myseed);
/* calculer */
pnl_rng_free (&rng);

if ( rank == 0) {
  for (j=1;j<size;j++) /* Recevoir la réponse de j */
} else {
  /* envoyer la réponse au maître */
}
MPI_Finalize ();
```


Communications collectives

- Permettent à un groupe de processus (un communicateur) de communiquer entre eux.
- Une seule opération pour plusieurs communications point à point : beaucoup plus efficace de répéter plusieurs fois les mêmes opérations élémentaires.
- L'exécution du programme ne se poursuit que lorsque l'opération collective est terminée, i.e. lorsque la communication est terminée pour tous les processus. Tous les processus sont synchronisés.
- Pas de possibilité de tagger les messages.

MPI_Bcast

Envoi d'un même message à tous les processus d'un groupe, y compris le processus qui réalise l'envoi

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm)
```

La fonction Bcast doit être appelée par tous les processus du groupe.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```


MPI_Gather

Chaque processus y compris le processus root envoie le contenu de son buffer au processus root.

```
int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- les arguments recvbuf, recvcount et recvtype ne sont pris en compte que sur le processus root
- On doit avoir sendcount = recvcount et sendtype = recvtype.

MPI_Gather

```
int gsize,sendarray[100];
int root, *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT,
rbuf, 100, MPI_INT, root, comm);
```

équivalent à si tous les processus appellent

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...)
```

et le processus root appelle *n* fois

```
MPI_Recv(recfbuf+i*recvcount, recvcount, recvtype, i, ...)
```


MPI_Scatter

Envoi d'un bloc différent à tous les processus d'un groupe, y compris le processus qui réalise l'envoi.

```
int MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- les arguments sendbuf, sendcount et sendtype ne sont pris en compte que sur le processus root
- On doit avoir sendcount = recvcount et sendtype = recvtype.
- Si recvbuf = MPI_IN_PLACE, alors le processus root ne s'envoie pas de données à lui-même.

MPI_Scatter

```
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100,
MPI_INT, root, comm);
```

équivalent à si le processus root effectue *n* opérations d'envoi

```
MPI_Send(sendbuf+i*sendcount, sendcount, sendtype, i, ...)
```

et tous les processus effectuent

```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...)
```


MPI_AllGather

Récupère les données de tous les processus et renvoie le résultat (la concaténation) à tous les processus. Même fonction que MPI_Gather mais tout le monde reçoit le résultat.

```
int MPI_Allgather(void *sendbuf, int  sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

```
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf,
100, MPI_INT, comm);
```

MPI_Alltoall

Tous les processus envoient leurs données à tous les autres. Toutes les données échangées doivent être de même taille

```
int MPI_Alltoall(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Comm_size(comm, &n);
for (i = 0, i < n; i++)
MPI_Send(sendbuf + i * sendcount,
sendcount, sendtype, i, ..., comm);
for (i = 0, i < n; i++)
MPI_Recv(recvbuf + i * recvcount,
recvcount, recvtype, i, ..., comm);
```


Communications collectives vectorielles

- Les fonctions MPI_Scatter, MPI_Gather, MPI_Alltoall et MPI_Allgather possèdent des version vectorielles MPI_Scatterv, MPI_Gatherv, MPI_Alltoallv et MPI_Allgatherv.
- Elles permettent d'envoyer ou de recevoir des blocs de taille variable
- Ces blocs ne sont pas forcément contigus en mémoire

Quelques conseils

- Préférer l'envoi d'un gros message à l'envoi de plusieurs petits messages.
- Recouvrir l'envoi ou la réception par des calculs en utilisant des communications non bloquantes. Attention le code est nettement plus délicat à écrire.

MPI_Isend

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

- Initialise un envoi
- Envoi non bloquant : l'exécution se poursuit immédiatement. Pas de blocage en attendant la terminaison de l'envoi
- Mêmes arguments que MPI_Send avec en plus un argument identifiant la requête.
- L'argument requête permet de tester l'état d'avancement de la requête (MPI_Test) ou de bloquer l'exécution tant que la requête n'a pas abouti (MPI_Wait).
- Ne pas utiliser la variable `buf` tant que le message n'a pas été reçu.

Attendre ou tester une requête

- MPI_Test : teste si une requête est terminée ou non

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Si la requête est terminée, flag=true et status contient les informations relatives à la requête, si status n'est pas utile peut être remplacé par MPI_STATUS_IGNORE (permet d'économiser des ressources).

- MPI_Wait : bloque le processus appelant tant que la requête n'est pas terminée.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

status contient les informations relatives à la requête, si status n'est pas utile peut être remplacé par MPI_STATUS_IGNORE (permet d'économiser des ressources).

MPI_Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- Initialise une réception
- Envoi non bloquant : l'exécution se poursuit immédiatement. Pas de blocage en attendant la terminaison de l'envoi
- Par rapport à MPI_Recv pas d'argument MPI_Status mais un argument MPI_Request pour identifier la requête.
- L'argument requête permet de tester l'état d'avancement de la requête (MPI_Test) ou de bloquer l'exécution tant que la requête n'a pas abouti (MPI_Wait).

Plan

Pack / Unpack

- Comment transmettre des messages complexes, par exemple des structures contenant des pointeurs ?
- Il vaut mieux un seul gros message que plusieurs petit. Peut-on agréger des données de type différent (par exemple une matrice et un vecteur) pour ne faire qu'un seul envoi ?
- Permet de stocker des objets évolués (tout type de structures ou tableaux) dans une chaine de caractères dans un format indépendant machine.
- On peut stocker autant d'objets que l'on veut les uns à la suite des autres.
- Utilisation dérivée : stocker des données sous forme binaire dans un fichier par exemple.

Pack

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
    void *outbuf, int outsize, int *position, MPI_Comm comm)
```

- `inbuf` : adresse où se trouvent les éléments à lire
- `incount` : nombre d'éléments
- `datatype` : type MPI de chacune des éléments
- `outbuf` : buffer où écrire les éléments
- `outsize` : taille du buffer
- `position` : position à laquelle commence l'écriture. En sortie, est incrémentée de la taille écrite pour un appel ultérieur à MPI_Pack

⇝ `inbuf` peut être écrit dans un fichier binaire pour être relu plus tard, même par un autre programme.

Pack_size

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
    MPI_Comm comm, int *size)
```

Permet de calculer la taille maximale d'un buffer pour stocker `incount` éléments de type `datatype`.

- `incount` : nombre d'éléments
- `datatype` : type MPI de chacune des éléments
- `size` : contient en sortie la taille minimale que doit avoir le buffer qui sera utilisé pour packer l'objet. C'est la valeur à donner au paramètre outsize de MPI_Pack.

Unpack

```
int MPI_Unpack(void *inbuf, int insize, int *position,
    void *outbuf, int outcount, MPI_Datatype datatype,
    MPI_Comm comm)
```

- `inbuf` : buffer à lire
- `insize` : taille du buffer
- `position` : position à laquelle commence la lecture. En sortie, est incrémentée de la taille lue pour un appel ultérieur à MPI_Unpack
- `outbuf` : adresse où écrire les éléments lus.
- `datatype` : type MPI des éléments
- `outcount` : nombre d'éléments

Exemples de Packing : le type PnlVect

```
struct _PnlVect
{
  PnlObject object;
  int size;/*!< size of the vector */
  double *array;/*!< pointer to store the data */
  int mem_size; /*!< size allocated for array */
  int owner; /*!< 1 if the object owns its array, 0 otherwise */
};
```

Quels champs envoyer ? size et array.

Exemples de Packing : le type PnlVect

```
char *buf;
int info, count, bufsize=0, pos=0;
info=MPI_Pack_size(1,MPI_INT, comm,&count);
if (info) return(info);
bufsize += count;
info=MPI_Pack_size(V->size,MPI_DOUBLE,comm,&count);
if (info) return(info);
bufsize += count;
buf = malloc(size);
info=MPI_Pack(&(V->size),1,MPI_INT,buf,bufsize,&pos,comm);
if (info) return info;
info=MPI_Pack(V->array,V->size,MPI_DOUBLE,buf,bufsize,&pos,comm);
return info;
```


Exemples de Unpacking : le type PnlVect

Lire un vecteur depuis le buffer `buf` de taille `bufsize`.

```
int n, info, pos=0;
PnlVect *V = pnl_vect_new ();
info=MPI_Unpack(buf,bufsize,&pos,&n,1,MPI_INT,comm);
if (info) return info;
pnl_vect_object_resize(V, n);
info=MPI_Unpack(buf,bufsize,&pos,V->array,n,MPI_DOUBLE,comm);
return info;
```

Comment fait-on pour connaître la taille `bufsize` ?

Transmission d'un PnlVect par Packing/Unpacking

```
if ( rank == 0 ) {
  /* calculer la taille du buffer avec MPI_Pack_size
  /* packer V dans buf  de taille bufsize */
  MPI_Send(buf, bufsize, MPI_PACKED, 1, tag, comm);
} else {
int info, bufsize;
  char *buf
  MPI_Status status;
  info = MPI_Probe(0, tag, comm, &status);
  if ( info ) return info;
  info = MPI_Get_count(status, MPI_PACKED, &bufsize);
  if ( info ) return info;
  buf = malloc(bufsize);
  info = MPI_Recv(buf,bufsize,MPI_PACKED,0,tag,comm,&status);
  if (info) return info;
  /* lire buf pour reconstruire V */
  free (buf);
}
```


Les types dérivés

- Construire un nouveau type à l'exécution
- Manipuler des données hétérogènes et/ou non contiguës en mémoire
- Permet ensuite d'utiliser directement MPI_Send et MPI_Recv (and co) sans passer par l'étape Pack/Unpack, ce qui évite des recopies inutiles.
- Ne fonctionne pas pour une structure dont un champ est alloué dynamiquement : le pas entre les champs d'une structure doit être le même pour toutes les instances.
  $\rightsquigarrow$ pour les objets complexes tels que vecteurs, matrices, générateurs : on est obligé de passer par Pack/Unpack.

Construire un type dérivé

Comment décrire une structure C

- le nombre de champs de la structure (int)
- les types élémentaires de chaque champs (MPI_Datatype $*$)
- nombre d'éléments dans chaque champ (int $*$)
- les adresses des différents champs, calculées relativement au début de la structure (MPI_Aint $*$)

```
int MPI_Type_create_struct(int count,
                int *array_of_blocklengths,
                MPI_Aint *array_of_displacements,
                MPI_Datatype *array_of_types,
                MPI_Datatype *newtype)
```

Une fois créé, un type doit être "enregistré"

```
int MPI_Type_commit(MPI_Datatype *datatype)
```


Un exemple élémentaire

```
typedef struct { int i, j; float f; char tab[10]; } structure;
```

```
MPI_Datatype newtype,
MPI_Aint displ [3];
int longueurs [3] = {2, 1, 10} ;
MPI_Datatype  types [3] = { MPI_INT, MPI_FLOAT, MPI_CHAR };

MPI_Get_address (&(structure.i), &(displ[0]));
MPI_Get_address (&(structure.f), &(displ[1]));
MPI_Get_address (structure.tab, &(displ[2]));

displ[1] -= displ[0];
displ[2] -= displ[0];
displ[0] = 0;

MPI_Type_create_struct(3,longueurs,displ,types,&newtype);
MPI_Type_commit (&newtype);
```


Des mécanismes simplifiés pour créer un type

- Données homogènes contiguës en mémoire : MPI_Type_contiguous.
  Exemple : une ligne d'une matrice (en stockage ligne).

```
MPI_Type_contiguous(n, MPI_DOUBLE, newtype);
```

- Données homogènes non contiguës en mémoire mais espacées d'un pas constant : MPI_Type_vector (en stockage ligne).
  Exemple : une colonne d'une matrice.

```
MPI_Type_vector(m, 1, n, MPI_DOUBLE, newtype);
```

- Données homogènes non contiguës en mémoire et espacées d'un pas variable : MPI_Type_indexed

```
int MPI_Type_indexed(int count, int *array_of_blocklen,
      int *array_of_displ, MPI_Datatype oldtype,
      MPI_Datatype *newtype)
```