Plan de l'exposé

1. Introduction
2. Pourquoi le passage de message?
3. MPI
4. PVM
5. PVM vs MPI (avocat du diable!)
6. Conclusion

PROGRAMMATION D'APPLICATIONS PAR PASSAGE DE MESSAGES

Frédéric DESPREZ

LaBRI

La partie MPI du tutorial est basée sur:

- La spécification du 5 mai 1994 de MPI,
- la feuille d'errata du 28 Octobre 1994,
- le livre de W. Gropp, E. Lusk et A. Skjellum: *Using MPI: Portable Programming with the Message Passing Interface*,
- le livre de Ian Foster: *Designing and Building Parallel Programs*,
- les transparents des personnes remerciées auparavant,
- des discussions avec Greg Burns,
- différents articles.

Remerciements

- David Walker (Oak Ridge National Lab.)
- Greg Burns et Raja Daoud (Ohio State University)
- Marie-Christine Counilh (LaBRI)
- William Saphir (NASA Ames Research Center)
- Jack Dongarra et Bob Manchek (University of Tennessee)
- Edinburgh Parallel Computing Centre
- et les autres ...

A propos de ce tutorial

ne sera pas:

- Une description exhaustive de tout ce que font PVM et MPI.

sera (enfin j'espère!):

- une vue d'ensemble du passage de message aujourd'hui,
- une démonstration des possibilités avec des exemples,
- donner des idées sur quoi utiliser et pourquoi.

La partie PVM du tutorial est basée sur:

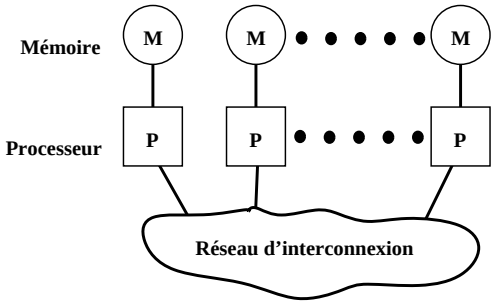
- Le livre de A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*,
- des discussions avec Bob Manchek (et ses transparents!),
- des transparents de W. Saphir (NASA),
- différents articles.

INTRODUCTION

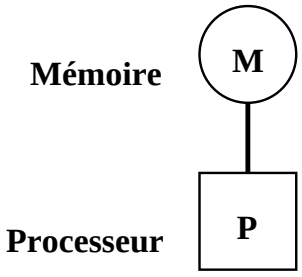
Autres bibliothèques (voire environnements)

APPL	Développée à NASA Lewis.
CMMD	Bib. native des machines de TMC.
Chameleon	Développée au Argonne Nat. Lab.
CHIMP	Développé à Edinburgh.
Express	Proprietaire (Parasoft).
LAM	OHIO State Univ., Ex Trollius, (vers. MPI).
MPL (EUI-H)	Bib. native du SP-2 d'IBM.
NX	Bib. native des machines Intel.
P4	Successeur de PARMACS (ANL).
PARMACS	Développé au Argonne Nat. Lab.
PICL	ORNL (gestion de traces en +)
TCGMSG	Modélisé à partir de PARMACS.
Zipcode	Développé à Livermore

Les machines à mémoire distribuée



La programmation séquentielle



SPMD

- *Single Program, Multiple Data*
- Le même programme s'exécute sur tous les processeurs.
- Restriction du modèle *à passage de messages* classique.

Pourquoi le passage de message?

1. Méthode de parallélisation la plus performante,
2. pas de compilateur paralléliseur réellement efficace pour toutes les applications,
3. certaines applications ne peuvent pas être parallélisées autrement,
4. il faut bien développer des outils sur une base (si possible portable et efficace),
5. la non-portabilité n'est plus un argument valable!

Sémantique vs Implémentation

a sémantique:

Ce que vous devez savoir d'une routine pour l'utiliser correctement.

'implémentation:

Détails de bas niveau qui explique comment une routine est construite pour implémenter la sémantique.

Monde idéal: Seule la sémantique est importante.
La triste réalité: L'implémentation est importante pour les performances.

Communications point-à-point

1. La forme la plus simple d'échange de message.
2. Un processus envoie un message à un autre.
3. Différents types de communications point-à-point.

Communications point-à-point: différences

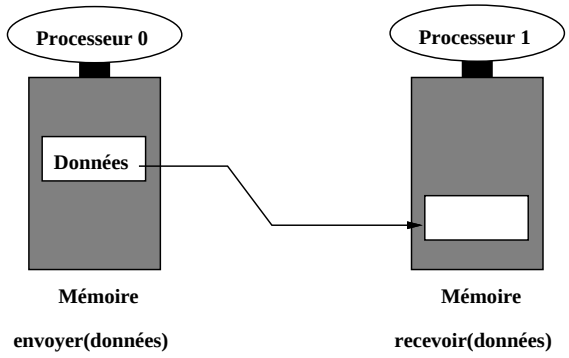
Questions:

- Que se passe-t-il si le message est envoyé avant qu'il soit reçu?
- Comment peut-on tenir compte de processeurs différents pour le calcul et les communications?
- Est-ce que l'on a un parc hétérogène de machines?
- Comment envoie(reçoit)-t-on des données non-contigues?
- Comment mesure-t-on les performances?

Réponses:

- Communications bloquantes et non-bloquantes (bufferisation).
- Communications asynchrones.
- Arguments typés.
- Types définis par l'utilisateur ou *pack/unpack*.
- Latence/bande passante.

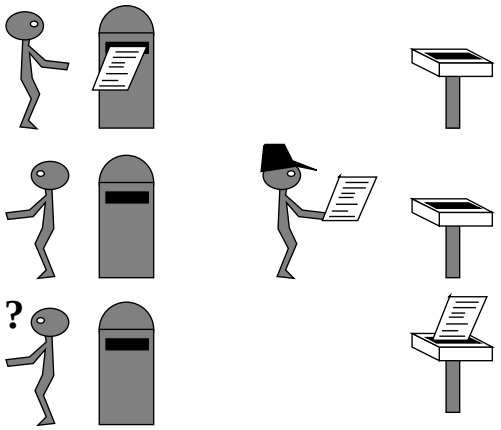
Communication de base:



send(buffer, taille_buffer, [étiquette], destination)
receive(buffer, taille_buffer, [étiquette], source)

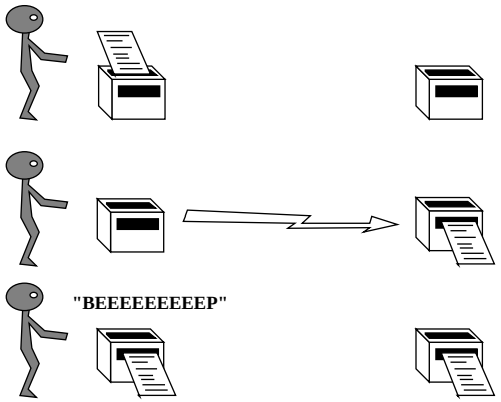
Envois asynchrones

n sait uniquement quand le message est parti (et encore!)



Envois synchrones

onne des informations sur la terminaison de la communication.

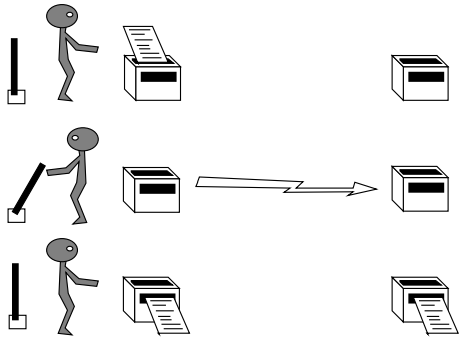


Opérations bloquantes

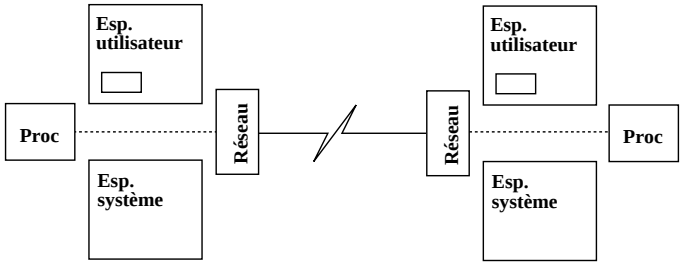
- Relatif à l'état quand on sort de l'appel de routine.
- On ne sort que lorsque l'opération est terminée.

Opérations non bloquantes

- Retourne immédiatement et autorise le sous-programme à continuer un autre travail. Par la suite, on teste ou on attend la fin de la communication.



On détaille!

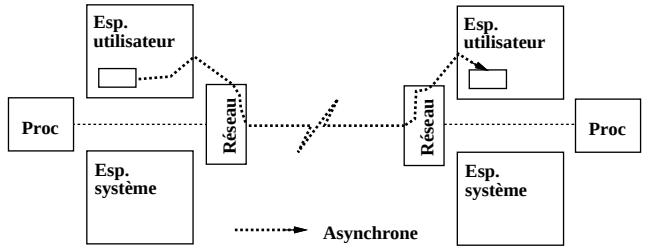


- Toutes les communications non-bloquantes doivent avoir un *wait* correspondant. Certains systèmes ne relachent pas les ressources tant qu'il n'a pas été effectué.

- Une opération non-bloquante suivie immédiatement d'un *wait* est équivalente à une opération bloquante (attention à l'implémentation!).

Communication asynchrone

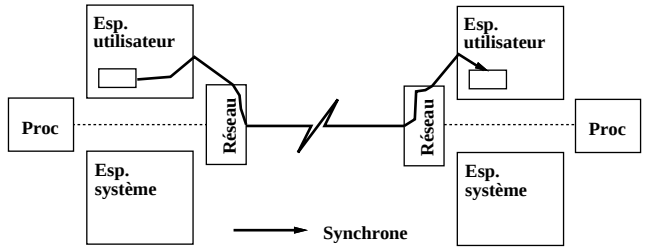
`send_async` indique au système qu'il faut envoyer un message quand la réception sera initiée. Des routines d'attente et de test de terminaison sont disponibles.



id = send_async(buffer, taille_buffer, [étiquette], destination)
id = receive_async(buffer, taille_buffer, [étiquette], source)
wait(id) ou test(id)

Communication non bufferisée (bloquante)

On ne sort pas du `send()` tant que la com. n'est pas terminée.



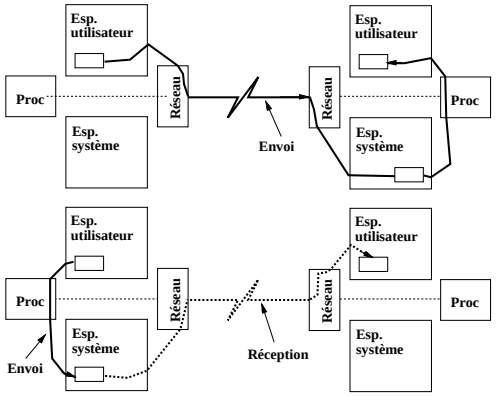
send_block(buffer, taille_buffer, [étiquette], destination)
receive_block(buffer, taille_buffer, [étiquette], source)

Communications collectives

- Routines de plus haut niveau avec plusieurs processus prenant part à l'opération.
- Peuvent bien sûr être bâties avec des routines point-à-point.

Communication bufferisée

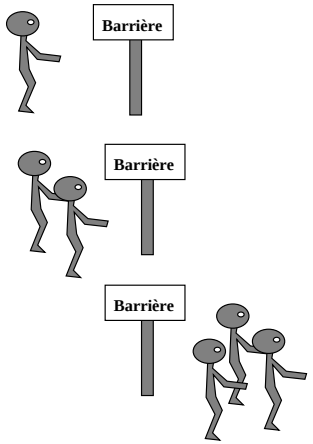
e message est copié dans l'espace système. `send()` peut terminer vant `receive()`.



send_bufferisé(buffer, taille_buffer, [étiquette], destination)

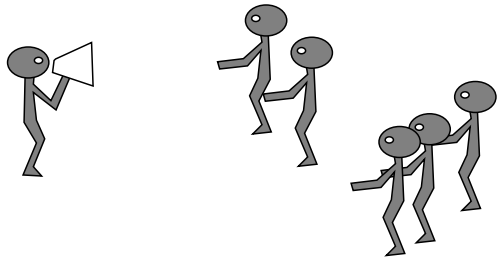
Barrière de synchronisation

- Synchroniser les processus.



Diffusion

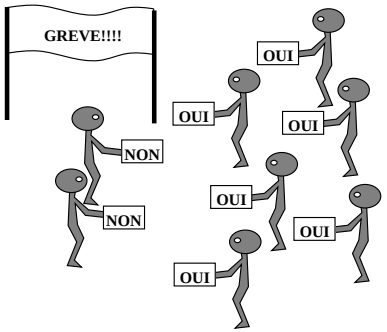
- Opération un-vers-tous (voire un-vers-quelques)



OPTIMISATION DES COMMUNICATIONS

Opérations de réduction

- Combine des données de plusieurs processus pour produire un résultat unique (diffusé ou pas).



Performances des communications p. à p.

Comment mesurer les performances des communications p. à p.?

Bonne réponse	faire tourner votre application!
Mauvaise réponse	lire les chiffres donnés par le constructeur!
Réponse classique	Mesurer les nombres "importants", généraliser.

Nombres importants et utiles:

- temps d'initialisation (latence)
- bande passante
- surcoût
- vitesse de recopie de buffer

L'extensibilité est importante mais généralement fonction du réseau, pas de la bibliothèque, pour les communications p. à p.

- bande passante agrégée
- contentions

Optimisation des communications

Trois optimisations importantes.

1. Minimiser le rapport calculs/communications.

Les communications apparaissent aux frontières. Minimiser le rapport surface/volume.

2. Cacher le coût des communications.

Recouvrir les calculs et les communications en tenant compte du fait que le réseau et les processeurs tournent de manière asynchrone.

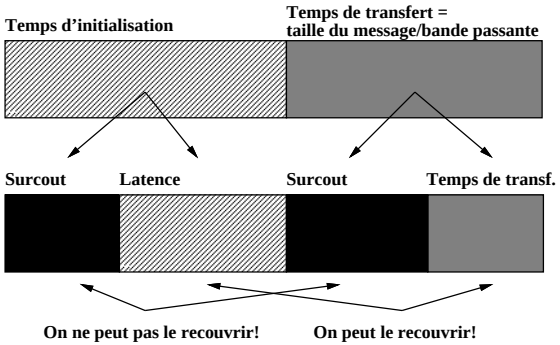
3. Eviter la bufferisation.

Quand un message est bufferisé, il doit être recopié entre les différentes mémoires.

Surcoût des communications

ombien de temps de communication peut être recouvert en ilisant des communications asynchrones (et la bufferisation)?

ela dépend du matériel.



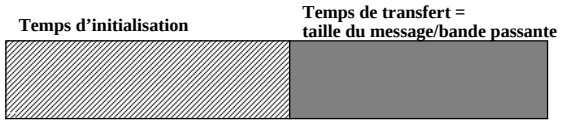
MPI

Transfert de messages: des détails

es temps de transfert des données sont exprimés en termes de:

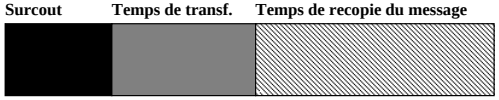
- temps d'initialisation (latence)
- bande passante

temps de transfert = temps d'initialisation + taille_message/bande_passante



Bufferisation

- Quand un message est bufferisé, il est copié de l'espace système vers l'espace utilisateur par le processeur.
- la bufferisation réduit la bande passante effective.
- La bufferisation ne peut être recouverte par les communications.
- L'importance de la bufferisation dépend de la bande passante mémoire.
- La bufferisation peut apparaître dans les programmes utilisateurs (envoyer un tableau non-contigu) ou dans les routines de bibliothèque.



Pourquoi a-t-on besoin d'un standard?

- Portabilité et facilité d'utilisation,
- offre aux constructeurs un ensemble bien défini de routines à implémenter efficacement,
- pré-requis pour le développement d'applications parallèles dans l'industrie,
- apportera une utilisation plus importante des ordinateurs parallèles.

Qu'est ce que MPI?

ne proposition d'interface de passage de message standard pour,

- **des échanges de messages explicites** dans
- **des programmes d'applications** sur
- **des machines parallèles MIMD à mémoire distribuée**.

Qu'est-ce qu'il y a dans MPI?

- Routines de communication point-à-point,
- routines de communication collectives,
- support pour les groupes de processus statiques,
- support pour les contextes de communication,
- support pour les topologies d'application,
- routines de récupération d'environnement,
- interface pour le profiling.

MPI **n'est pas un environnement complet** de programmation pour machines parallèles!

Dans MPI, il n'y a pas:

- de support pour la manipulation de processus,
- de support pour le debug,
- de support pour les canaux virtuels,
- d'opérations type mémoire partagée (remote get, remote put),
- d'opérations *interrupt driven*,
- de support pour les messages actifs,
- de support pour les processus légers.

Communications point-à-point

- MPI offre des communications point-à-point entre des paires de processus.
- La sélection de messages est faite par le rang et l'étiquette du message.
- Le rang et l'étiquette sont interprétés relativement à l'étendue de la communication.
- l'**étendue** est spécifiée par le communicateur.
- Le rang et l'étiquette peuvent être des jokers.

Comment ont-ils fait?

- Pas de fonctionnalités nouvelles.
- Fonctionnalités similaires à celles proposées par des systèmes de transmission de messages très utilisés:

  Express, NX, Vertex, PARMACS et P4.

- Des idées prises dans des systèmes plus récents:

  CHIMP, Zipcode, EUI d'IBM.

Modèle de processus et de groupes

- L'unité de calcul fondamentale est le processus.
  Chaque processus possède:
  - un flot de contrôle indépendant,
  - un espace d'adressage séparé.
- Les processus MPI s'exécutent en mode MIMD, mais:
  - il n'y a aucun mécanisme pour charger le code sur les processeurs ou assignés les processus aux processeurs,
  - et aucun mécanisme pour créer ou détruire les processus.
- MPI supporte les groupes de processus dynamiques.
  - Les groupes de processus peuvent être créés ou détruits,
  - l'appartenance est statique,
  - les groupes peuvent se recouvrir.
- Pas de support explicite pour les processus légers, mais MPI a été conçu pour être *thread-safe*.

Achèvement d'une communication

- Une communication est **terminée localement** sur un processus si un processus a terminé sa part de l'opération.
- Une communication est **terminée globalement** si tous les processus participants ont terminé leur part de l'opération.

Une communication est **globalement** terminée
si et seulement si
elle est terminée **localement** pour tous les processus.

PROGRAMMES MPI

Etendue des communications

- Dans MPI, un processus est spécifié par:
  - un **groupe**,
  - un rang relatif dans le groupe $(0, 1, \ldots, N-1)$.
- Une étiquette de message est spécifiée par:
  - un **contexte** de message,
  - un indice de message relatif au contexte.
- Les groupes sont utilisés pour partitionner l'espace des processus.
- Les contextes sont utilisés pour partitionner l'espace des étiquettes de messages.
- Les groupes et les contextes sont regroupés pour former un **objet communicateur**. Les contextes ne sont pas visibles au niveau application.
- Un communicateur définit **l'étendue** de l'opération de communication.

Format des fonctions MPI

- C

```
error = MPI_xxxxx(parameter,...);
MPI_xxxxx(parameter,...);
```

- Fortran

```
call MPI_xxxxx(parameter,..., ierror)
```


Fichiers entêtes

- C

```
#include<mpi.h>
```

- Fortran

```
include 'mpif.h'
```

Sortir de MPI

- C

```
int MPI_Finalize()
```

- Fortran

```
MPI_FINALIZE(IERROR)
INTEGER IERROR
```

- doit être appelé par **tous** les processus.

Initialiser MPI

- C

```
int MPI_init(int *argc, char ***argv)
```

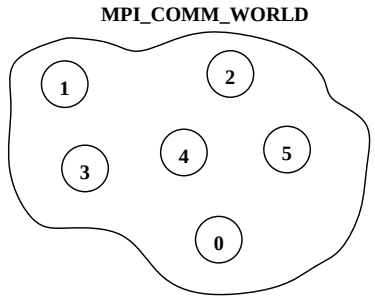
- Fortran

```
MPI_INIT(IERROR)
INTEGER IERROR
```

- doit être la première routine appelée.

Communicateur MPI_COMM_WORLD

- Modèle orienté processus.
- Chaque processus possède un rang unique.



Rang

- Comment identifier différents processus?

```
MPI_Comm_Rank(MPI_Comm comm, int *rank)
```

```
MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR
```

Hello World!

```
program main
begin
  MPI_INIT()                                  Initialiser MPI

  MPI_COMM_SIZE(MPI_COMM_WORLD, count)        Combien sommes-nous?
  MPI_COMM_RANK(MPI_COMM_WORLD, myid)         Qui suis-je?

  print("Hello world de",myid,"sur",count)

  MPI_FINALIZE()                              Shut down
end
```


PREMIÈRES COMMUNICATIONS

Taille

- Combien de processus sont contenus dans le communicateur?

```
MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR
```


Les messages

- Un message contient un nombre d'éléments de types particuliers.
- types MPI:
  - Types de base
  - Types dérivés
- Les types dérivés peuvent être construits à partir des types de base.
- Les types C sont différents des types Fortran.

Réception bloquante

```
mpi_recv (
            OUT start_of_buffer,
            IN max_number_of_items,
            IN datatype,
            IN source_rank,
            IN tag,
            IN communicator,
            OUT return_status,
            OUT error_code)
```

ans une routine de réception, `source_rank` et `tag` peuvent avoir
s valeurs `MPI_ANY_SOURCE` et `MPI_ANY_TAG` (jokers).

Envoi bloquant

```
mpi_send (
            IN start_of_buffer,
            IN number_of_items,
            IN datatype,
            IN destination_rank,
            IN tag,
            IN communicator,
            OUT error_code)
```


Types de blocages

- **Envoi non bloquant**:
  - Retourne “immédiatement”.
  - Le buffer ne doit pas être écrit au retour.
  - Il faut vérifier la terminaison.
- **Envoi bloquant**:
  - Retourne quand l’envoi est terminé localement.
  - Le buffer peut être écrit au retour.
- **Réception non bloquante**:
  - Retourne ”immédiatement”.
  - Le buffer ne doit pas être lu au retour.
  - Il faut vérifier la terminaison.
- **Réception bloquante**:
  - Retourne quand la réception est terminée localement.
  - Le buffer peut être lu au retour.

Objet statut retourné

- L’objet de statut retourné est utilisé à la fin d’une réception pour trouver la longueur, la source et l’étiquette (`tag`) d’un message.
- L’objet de statut est un type MPI prédéfini.
- En C, `statut` est une structure.
  - `statut.source` donne le processus source.
  - `statut.tag` donne l’étiquette du message.
- En Fortran, `statut` est un tableau d’entiers.
  - `statut(MPI_SOURCE)` donne le processus source.
  - `statut(MPI_TAG)` donne l’étiquette du message.
- Le nombre d’éléments dans le message est donné par:
  - `MPI_Get_count(statut, datatype, count)`

Réception non bloquante

```
mpi_irecv (
            OUT start_of_buffer,
            IN max_number_of_items,
            IN datatype,
            IN source_rank,
            IN tag,
            IN communicator,
            OUT request_id,
            OUT error_code)
```

e receive non bloquant ne possède pas de paramètre `statut` (il
ra passé à la routine d'attente de terminaison).

Envoi non bloquant

```
mpi_isend (
            IN start_of_buffer,
            IN number_of_items,
            IN datatype,
            IN destination_rank,
            IN tag,
            IN communicator,
            OUT request_id,
            OUT error_code)
```


Routines de terminaison

- Deux moyens de base pour vérifier la complétion des envois et réception non bloquants:
  - utiliser une routine qui bloque jusqu'à la terminaison,
  - utiliser une routine qui retourne un drapeau,
- L'utilisation des routines non bloquantes et d'attente permet un recouvrement des communications et des calculs.

  `mpi_wait(request_id, return_status, ierr)`
  `mpi_test(request_id, return_status, ierr)`
- `mpi_wait` bloque jusqu'à ce que la communication soit terminée.
- `mpi_test` retourne "immédiatement", et met `flag` à `vrai` si la communication est terminée.

Terminaisons multiples

- Versions des routines `wait` et `test` qui travaillent sur plusieurs communications.
- Bloquer jusqu'à ce que toutes les communications d'une liste donnée soient terminées.

`mpi_wait_all( count, list_requests, list_status, ierr)`

- Bloquer jusqu'à ce qu'une communication d'une liste donnée soit terminée.

```
mpi_waitany( count, list_requests,
             index, return_status, ierr)
```

- Bloquer jusqu'à ce qu'au moins une communication soit terminée.

```
mpi_waitsome( count, list_requests, count_done,
              list_index, list_status, ierr)
```

- Il existe des versions similaires pour la routine `test`.

Types de base MPI - C

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

LES TYPES MPI

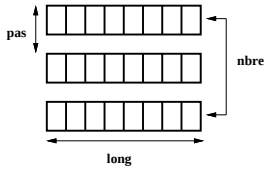
Types MPI de base - Fortran

Type MPI	**Type Fortran correspondant**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Les types non-primitifs

- MPI supporte également les sections de tableaux et les structures grâce aux **types généraux**.
- Un type général est une séquence de types primitifs et de déplacements en octets.
  $$\text{type} = \{(\text{type}_0,\text{dép}_0), \ldots, (\text{type}_{n-1},\text{dép}_{n-1})\}$$
- Avec l'adresse de base, un type définit un buffer de communication.
- Il faut *démarrer* le nouveau type avec:
  `MPI_Type_commit(&type)`
- On peut *relâcher* un type avec:
  `MPI_Type_free(&type)`

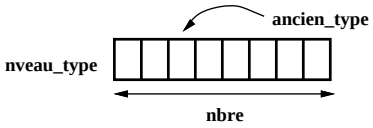
Le constructeur de type vecteur



- ce constructeur duplique un type, en prenant des blocs à des pas fixes:
  `MPI_TYPE_vector(nbre,long,pas,anc_type,&nveau_type);`
- Le nouveau type consiste en:

  - `nbre` blocs, chacun étant une répétition de `long` éléments de `anc_type`. Le début des blocs successifs est donné par le `pas`.
- `MPI_Type_hvector()` est similaire mais avec un pas donné en octets.

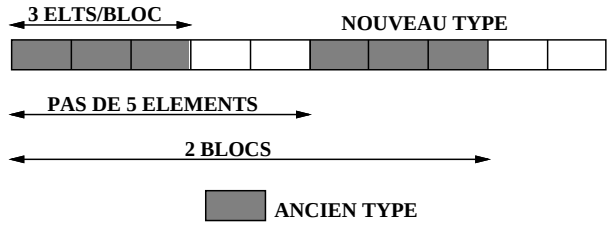
Le constructeur de type contigu



- Un type général est construit hiérarchiquement à partir de types simples.
  `MPI_Type_contiguous(nbre, ancien_type, &nveau_type)`
- l'exemple précédent construit `nveau_type` à partir de `nbre` éléments de `ancien_type`.

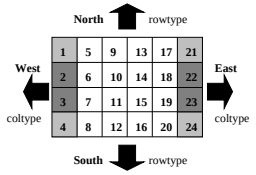
Si le type précédent (`ancien_type`) était {(int,0),(double,8)}
`MPI_Type_contiguous(2, ancien_type, &nveau_type);`
va donner:
{(int,0),(double,8),(int 16),(double,24)}

Exemple de type vecteur



- `nbre = 2`
- `pas = 5`
- `long = 3`

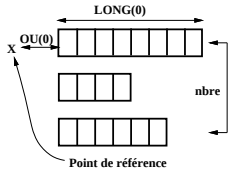

```
      integer coltype, rowtype, comm, ierr
C     Le type dérivé coltype contient 4 réels contigus
      call MPI_TYPE_CONTIGUOUS(4, MPI_REAL, coltype, ierr)
      call MPI_TYPE_COMMIT(coltype, ierr)
C     Le type dérivé rowtype contient 6 réels
      call MPI_TYPE_VECTOR(6,1,4,MPI_REAL,rowtype,ierr)
      call MPI_TYPE_COMMIT(rowtype, ierr)
      ...
      call MPI_SEND(array(1,1), 1, coltype, west, 0, comm, ierr)
      call MPI_SEND(array(1,6), 1, coltype, east, 0, comm, ierr)
      call MPI_SEND(array(1,1), 1, rowtype, north, 0, comm, ierr)
      call MPI_SEND(array(4,1), 1, rowtype, south, 0, comm, ierr)
      ...
      call MPI_TYPE_FREE(rowtype, ierr)
      call MPI_TYPE_FREE(coltype, ierr)
```



Envoi d'une matrice triangulaire supérieure

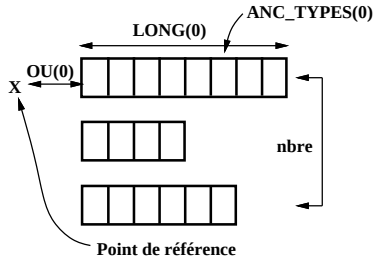
```
double a[100][100];
int disp[100], blocklen[100], i;
MPI_Datatype upper;
:
/* Calculer le début et la fin pour chaque ligne                    */
for (i=0; i¡100; i++) {
      disp[i] = 100*i + i;
      blocklen[i] = 100 - i;
}

/* Créer un type pour le triangle supérieur                         */
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);

/* Et envoyer!!                                                     */
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
```


Le constructeur de type indexé



- Ce constructeur duplique un type, en prenant des blocs à des pas donnés dans un tableau.
  `MPI_Type_indexed(nbre,LONG,OU,anc_type,&nveau_type);`
- le `nveau_type` est `nbre` tableaux de taille `LONG[i]`, chacun étant placés avec un déplacement `OU[i]` de `anc_type` (`OU` en nombre d'éléments).
- `MPI_Type_hindexed()` est similaire mais `OU` est donné en octets.

Le constructeur de type structure

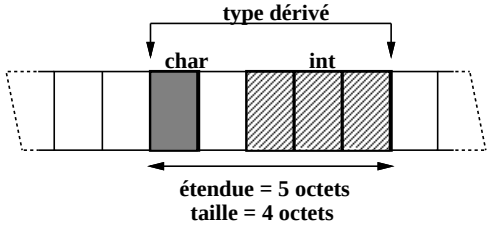


- Ce constructeur généralise le type indexé en permettant à chaque bloc d'être d'un type différent.
  `MPI_Type_struct(nbre,LONG,OU,ANC_TYPES,&nveau_type);`
- le `nveau_type` est `nbre` tableaux de `LONG[i]` éléments de types `ANC_TYPES[i]`, chacun étant placés avec un déplacement `OU[i]` (`OU` en octets).

D'autres routines pour les types

`MPI_Type_extent(type, &extent)`	Donne l'étendue d'un type.
`MPI_Type_size(type, &size)`	Donne la taille d'un type.
`MPI_Type_Count(type, &count)`	Donne le nbre d'élts d'un type.

Les offsets dans un type peuvent être donnés de manière relative à une adresse de base, donnée par la constante MPI `MPI_BOTTOM`.

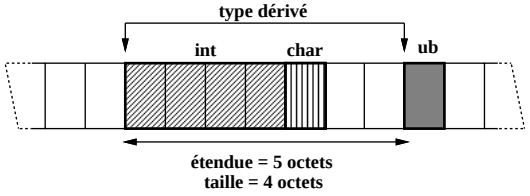


COMMUNICATIONS POINT-À-POINT MPI

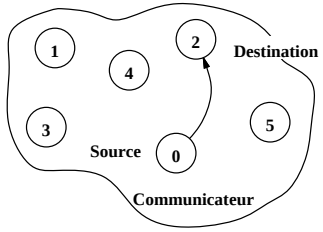
Pseudo-types

- `MPI_LB` marque la borne inférieure du type.
- `MPI_UB` marque la borne supérieure du type.

- Utilisés pour faire des *trous* dans les types.
- Ils n'ont pas de tailles ni d'étendue.



Communications point-à-point



- Communication entre deux processus.
- Un processus source envoie un message à un processus destination.
- La communication se passe à l'intérieur d'un communicateur.
- Le processus destination est identifié par son rang dans le communicateur.

Modes de communication

Le mode d'une communication point-à-point gouverne quand une opération d'envoi est initiée, ou bien quand elle se termine.

- **Mode Standard:** Un envoi peut être démarré, **même si** la réception correspondante n'a pas été initiée.
- **Mode Ready:** Un envoi ne peut être démarré **que si** la réception correspondante a été initiée.
- **Mode Synchrone:** Idem que le mode Standard, mais l'envoi **ne se terminera pas tant que** la réception n'est pas garantie.
- **Mode Bufferisé:** Idem que le mode Standard, mais la terminaison **est toujours indépendante** de la réception correspondante, et le message peut être bufferisé pour s'en assurer.

Types de communication

- Pour une opération send, il y a:
  - 4 modes de communication, 2 modes de blocages
  - $\rightarrow$ 4*2 = 8 types de send
- Pour une opération receive, il y a:
  - 1 mode de communication, 2 modes de blocage
  - $\rightarrow$ 1*2 = 2 types de receive
- Convention de nommage des primitives send:

`MPI_[-,i][-,r,s,b]send`
`[-,i]` = mode de blocage
`[-,r,s,b]` = mode de communication

- Convention de nommage des primitives receive:

`MPI_[-,i]recv`

Mode bufferisé

- Dans le mode bufferisé, un buffer utilisateur peut être utilisé de telle manière que le processeur émetteur peut **toujours** sortir du send avant que le message soit reçu.
- Pour donner au système le buffer utilisateur:

```
MPI_Buffer_attach(buffer, size);
```

- Pour récuperer le buffer utilisateur:

```
mpi_buffer_detach(buffer, size);
```


Opérations Send/Receive

- Dans de nombreux programmes, les processus effectuent un envoi vers un processus tout en recevant d'un autre (ex: sur un anneau).
- Des interblocages peuvent arriver si l'on ne fait pas attention.
- Si l'on fait attention, plus SPMD!!!
- MPI offre des routines pour ce type d'opérations.
- Si l'on utilise des buffers différents en réception et en émission:

`mpi_sendrecv`

- Si l'on utilise le même buffer en réception et en émission:

`mpi_sendrecv_replace`

Convention de nommage

Envoi	**Bloquant**	**Non bloquant**
Standard	`mpi_send`	`mpi_isend`
Ready	`mpi_rsend`	`mpi_irsend`
Bufferisé	`mpi_bsend`	`mpi_ibsend`
Synchrone	`mpi_ssend`	`mpi_issend`

Réception	**Bloquante**	**Non bloquante**
Standard	`mpi_recv`	`mpi_irecv`

- N'importe quel type de primitive receive peut être utilisé pour la réception de messages émis par n'importe quel type de primitive send.

- Pour démarrer un envoi ou une réception:

```
mpi_start(request, ierr)
mpi_start_all(count, requestarray, ierr)
```

- Les routines d'attente et de test peuvent être utilisées pour se bloquer en attente de terminaison ou tester le statut.

Requêtes de communications persistantes

- On peut agréger les arguments d'une communication ensembles (pour sortir le surcoût d'une boucle).
- `mpi_send_init` crée une requête de communication qui spécifie complètement une opération d'envoi classique. Cela agrège:
  - le début du buffer et le nombre d'éléments envoyés,
  - le type,
  - le rang de destination, l'étiquette du message et le communicateur.
- `mpi_recv_init` crée une requête de communication qui spécifie complètement une opération de réception.
- Il y a des routines similaires pour les modes ready, synchrone et bufferisé.
- Ces routines renvoient un `MPI_Request` (comme les communications non bloquantes).

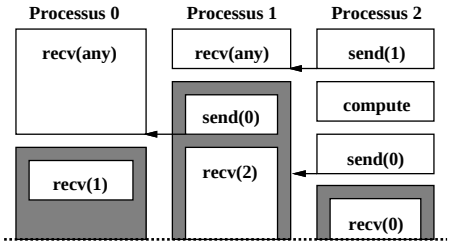
Communicateurs

- Les communicateurs servent à créer des **"univers de communication"** indépendants.
- Les communicateurs sont utilisés pour lever l'ambiguité quand une application appelle une routine de bibliothèque qui communique. Il peut y avoir des problèmes si:
  - des processus appellent la routine de bibliothèque de manière asynchrone,
  - les processus appellent la routine de bibliothèque de manière synchrone mais il y a des communications qui recouvrent l'appel.
- Un communicateur:
  - permet de regrouper les groupes et les contextes,
  - définit l'étendue de la communication,
  - est représenté par un objet *opaque*.

LES COMMUNICATEURS

Appel asynchrone de routine 2

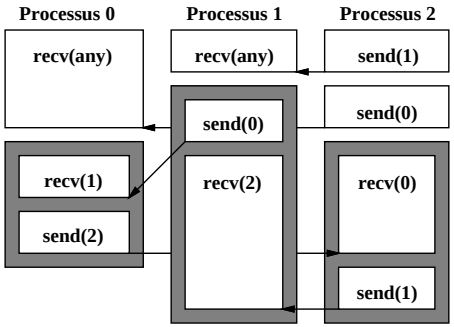
- Voici une séquence incorrecte de communications.



- Un interblocage apparaît.
- Il faut différencier les messages envoyés dans une bibliothèque du reste de l'application.

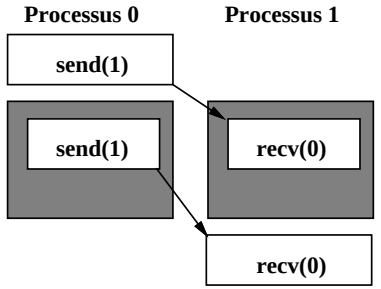
Appel asynchrone de routine 1

- Voici une séquence correcte de communications.



Appel synchrone de routine 2

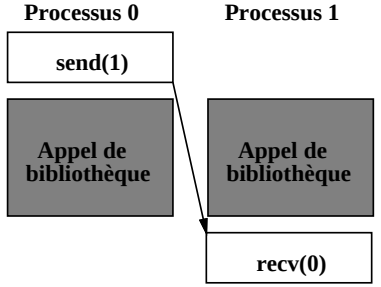
- Comportement possible:



- Nous devons séparer de nouveau les communications de la bibliothèque et ceux externes à la bibliothèque.

Appel synchrone de routine 1

- Dans ce cas, l'appel de routine est fait de manière synchrone.
- Il y a quand même des problèmes si certaines communications recouvrent l'appel.
- Comportement prévu:



Groupes de processus

- Un groupe est un ensemble de N processus

  - Ces processus sont numérotés de manière unique $(0, \ldots, N-1)$.

  - l'entier qui réfère un processus est appelé **rang** du processus.

- Un groupe est représenté par un **objet opaque**

  - l'utilisateur ne connait pas la structure interne.

  - il doit utiliser des routines pour déterminer les attributs.

```
MPI_Group_size(group, &size)
MPI_Group_rank(group, &rank)
```

- Initialement, tous les processus sont membres d'un groupe donné par le communicateur pré-défini `MPI_COMM_WORLD`.

Bibliothèques et communicateurs

- Différentes phases d'une application possèdent des contextes différents pour éviter que leurs messages se mélangent.

- Les bibliothèques doivent recevoir un communicateur en paramètre pour leur usage interne.

- **Attention!** L'utilisation de communicateurs n'empêche pas forcément les interblocages: si l'on utilise le même communicateur pour deux appels successifs. Une solution consiste à mettre une barrière de synchronisation entre les appels.

Gestion de groupes

- Les opérations sur les groupes sont locales et ne nécessitent pas de communications.

- On peut convertir le rang d'un groupe vers le rang d'un second groupe:

```
mpi_group_translate( group1, n, ranks1,
                     group2, ranks2, ierr)
```

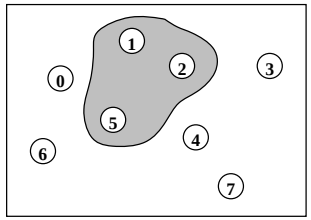
- On peut tester si deux groupes sont égaux:
`mpi_group_compare( group1, group2, result, ierr)`

- On peut extraire le groupe associé à un communicateur:
`mpi_comm_group(comm, group, ierr)`

- On peut marquer un objet groupe pour désallocation:
`mpi_group_free(group, ierr)`

Exemple de construction de groupe

- Supposons un groupe de 8 processus, et `ranks=(1,5,2)`




```
call mpi_group_incl( group, 3, ranks,
                     newgroup, ierr)
```

Nveau Rang	Ancien Rang
0	1
1	5
2	2

Gestion des communicateurs

- On peut obtenir la taille du groupe associé au communicateur:
  `mpi_comm_size(comm, size, ierr)`
- On peut obtenir le rang du processus appelant associé à `comm`:
  `mpi_comm_rank(comm, rank, ierr)`
- On peut comparer les communicateurs:
  `mpi_comm_compare(comm1, comm2, result, ierr)`

  Retourne:

  - `MPI_IDENT` si les groupes et les contextes sont identiques.
  - `MPI_CONGRUENT` si les groupes ont les mêmes membres et rangs, mais diffèrent dans les contextes.
  - `MPI_SIMILAR` si les groupes ont les mêmes membres mais diffèrent dans les rangs et les contextes.
  - `MPI_UNEQUAL` sinon.

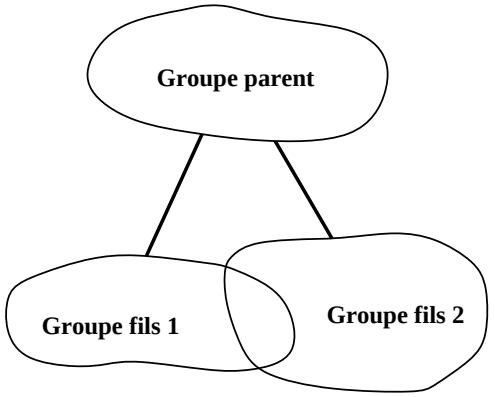
Routines de gestion de groupes

- Dans chaque groupe, les processus sont numérotés de 0 à $N-1$.
- Un groupe est représenté par un descripteur `MPI_Group`.

`MPI_Group_size(g, &size)`	Récupérer le nbre de processus.
`MPI_Group_rank(g, &rank)`	Quel est mon rang?
`MPI_Group_union(g1,g2,g)`	Union.
`MPI_Group_intersection(...)`	intersection.
`MPI_Group_difference(...)`	Différence.
`MPI_Group_incl(g,n,r[],ng)`	Créer un ss-gpe (inclusion).
`MPI_Group_excl(...)`	Créer un ss-gpe (exclusion).
`MPI_Group_free(&g)`	Relâcher un groupe.

utres routines: `MPI_Group_translate_rank()`, `PI_Group_range_incl(?)`, `MPI_Group_range_excl(?)`.

Construction des communicateurs



Construction des communicateurs

- Créer un nouveau communicateur pour un groupe donné
  `mpi_comm_create(comm, group, newcomm, ierr)`

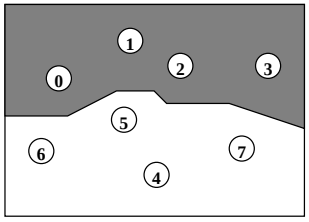
  - `group` doit être un sous-ensemble du groupe associé à `comm`.
  - doit être appelé par **tous** les processus de `comm`.
- Supposons que l'on crée des communicateurs pour deux sous-groupes. Ils peuvent se recouvrir, et doivent donc avoir des contextes différents. De ce fait, tous les processus du groupe *parent* doivent appeler la routine de création pour s'assurer que les contextes sont différents.

Exemple de découpage de communicateur

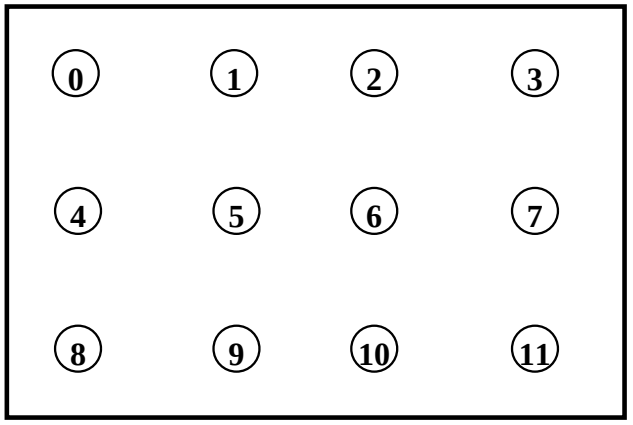
Communicateur 1		Communicateur 2	
Rang Nveau Gpe	Rang Anc. Gpe	Rang Nveau Gpe	Rang Anc. Gpe
0	3	0	7
1	2	1	6
2	1	2	5
3	0	3	4

Exemple de découpage de communicateur

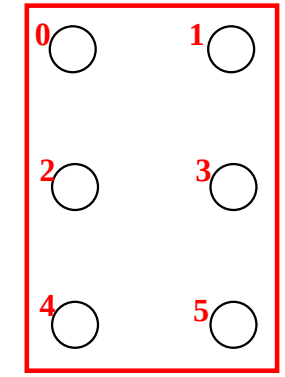
- Prenons un communicateur avec un groupe de 8 processeurs.

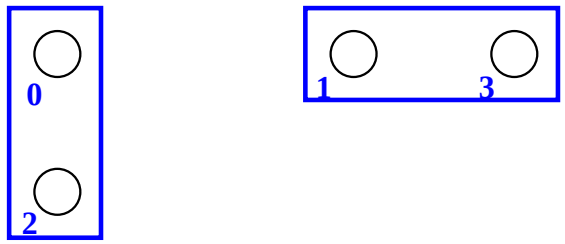


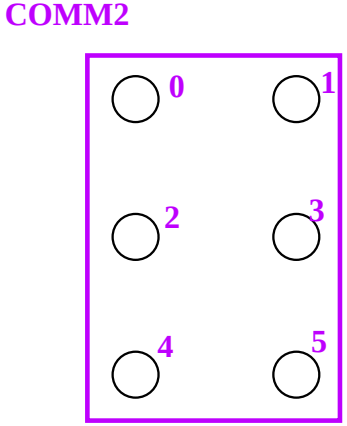

```
call mpi_comm_rank(comm, rank, ierr)
call mpi_comm_size(comm, size, ierr)
color = 2*rank/size
key = size - rank -1
call mpi_comm_split(comm, color, key, newcomm, ierr)
```

MPI_COMM_WORLD



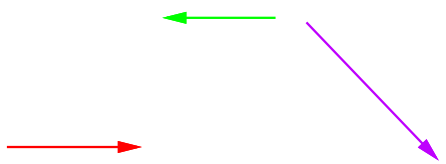


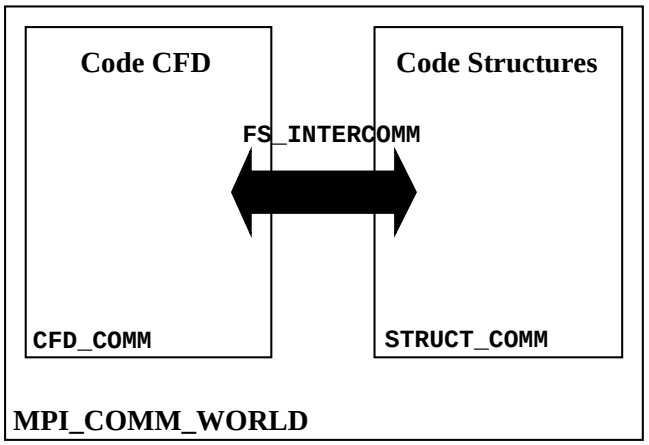


Routines de gestion de communicateurs

- Une fois qu'un groupe est défini, un communicateur peut être créé pour lui (étape de communication globale pour attribuer le contexte).
- Un communicateur est représenté par un descripteur `MPI_Comm`.

`MPI_Comm_size(c, &size)`	Récupérer le nbre de processus.
`MPI_Comm_rank(c, &rank)`	Quel est mon rang?
`MPI_Comm_group(c, &g)`	Récupérer le gpe du communicateur.
`MPI_Comm_dup(c, &nc)`	Dupliquer un communicateur.
`MPI_Comm_make(c, ng, &nc)`	Créer un comm. pour un gpe.
`MPI_Comm_split(?)`	Eclater un comm. en sous-groupes.
`MPI_Comm_free(&g)`	Relâcher un communicateur.





Communications entre les groupes

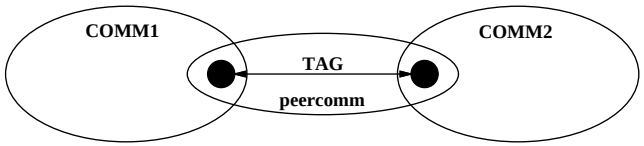
- Nous avons présenté les communications entre les membres d'un même groupe.
  - On les appelle les communications **intragroupe**.
- MPI supporte également les communications entre les membres de groupes *différents* qui ne se recouvrent pas.
  - On les appelle les communications **intergroupes**.
- Les communications intergroupes peuvent être utilisées pour:
  - les communications entre les différents modules parallèles d'une application,
  - un modèle client-serveur de calcul,
  - pour supporter des extensions de MPI pour le modèle dynamique de processus.

Communications entre les groupes

- Le processus spécifie le processus destination en donnant le rang relatif au groupe distant.
- Le récepteur spécifie le processus source en donnant le rang relatif au groupe distant.
- Les *intercommunicateurs* peuvent être utilisés pour les communications point-à-point seulement:
  - pas de communications collectives.
  - pas de topologies d'application.

Accès aux intercommunicateurs

- On peut obtenir la taille du groupe distant:
  `mpi_comm_remote_size(comm, size, ierr)`
- On peut récupérer l'objet du groupe distant:
  `mpi_comm_remote_group(comm, group, ierr)`
- Les routines `mpi_comm_size`, `mpi_comm_rank` et `mpi_comm_group` retournent des informations sur le groupe local.



Opérations intercommunications

- Pour créer un intercommunicateur, on a besoin:
  - un *group leader* pour chaque groupe, connu des membres de son groupe,
  - un intracommunicateur pour les communications entre les deux *group leaders*,
  - une étiquette pour communiquer de manière sûre entre les deux *group leaders*.
- Ensuite, un intercommunicateur peut être créé en utilisant:

```
mpi_intercomm_create( localcomm, localleader, peercomm,
                      remote_leader, tag, newintercomm, ierr)
```

- On peut créer un intracommunicateur à partir d'un intercommunicateur:

```
mpi_comm_merge(intercomm, high, newintracomm, ierr)
```

- `high` (booléen) détermine les rangs dans le nouveau groupe.

Communications collectives

- Communications coordonnées à l'intérieur d'un groupe.
- Ces opérations n'ont pas d'étiquette.
- Toutes les routines bloquent jusqu'à terminaison locale.
- 3 classes de communications collectives:
  - Mouvements de données.
  - Calculs globaux.
  - barrière de synchronisation.

COMMUNICATIONS COLLECTIVES

- Les routines gather, scatter, allgather, et alltoall possèdent des **versions vectorielles** dans lesquelles chaque processus peut recevoir un nombre différent d'éléments.
- Par exemple, la routine:

```
mpi_gather( sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm, ierr)
```

- possède une version vectorielle:

```
mpi_gatherv( sendbuf, sendcount, sendtype, recvbuf,
             recvcounts, displs, recvtype,
             root, comm, ierr)
```

- `recvcounts` est un tableau donnant le nombre d'éléments reçus de chaque processus.
- `displs` donne le déplacement auquel sont rangés les éléments reçus.

Mouvements collectifs de données

- Trois types de mouvements collectifs de données:
  - Diffusion (*broadcast*).
  - Récupération (*gather*).
  - Distribution (*scatter*).
- Dans la routine "all-gather", tous les processus reçoivent le résultat. Elles concatènent les données de chaque processus.
- Dans la routine "all-to-all", chaque processus envoie et reçoit de chaque processus.

Routines de calculs globaux

- 2 types d'opérations:
  - réduction (*reduce*)
  - préfixe parallèle (*scan*)
- Si n est la taille du groupe, et $D_{i,j}$ la $j$ème donnée dans le processus i, alors une routine de réduction évalue:
$$D_j = D_{0,j} \oplus D_{1,j} \oplus \ldots \oplus D_{n-1,j}$$
- $\oplus$ la fonction de réduction. Elle doit être associative et commutative.
- les D_j sont rangés dans un processus spécifié.
- Une routine de préfixe parallèle évalue dans le processus $k = 0, 1, \ldots, n-1$:
$$D_{k,j} = D_{0,j} \oplus D_{1,j} \oplus \ldots \oplus D_{k,j}$$

Routines de calculs globaux

- La fonction passée à la routine de calcul global peut être:
  - une routine prédéfinie dans MPI (ex. `mpi_sum`),
  - une routine donnée par l'utilisateur.
- La fonction utilisateur doit être validée en utilisant:
  `mpi_op_create(function, commute, op, ierr)`
- Trois versions de réduction peuvent retourner le résultat dans:
  - un seul processus,
  - tous les processus,
  - distribuer le vecteur résultat sur les processus.
- Des préfixes segmentés peuvent être effectués en créant un sous-groupe pour chaque segment.
- Il y a 4 routines de calculs globaux, plus deux routines auxiliaires (`mpi_op_create` et `mpi_op_free`).

Barrière de synchronisation

- On effectue une synchronisation globale: tous les processus sont bloqués tant qu'ils n'ont pas tous *passé* la barrière.

```
MPI_Barrier(comm)
```


Fonctions prédéfinies

Nom MPI	**Fonction**
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Somme
MPI_PROD	Produit
MPI_LAND	ET logique
MPI_BAND	ET bit à bit
MPI_LOR	OU logique
MPI_BOR	OU bit à bit
MPI_LXOR	OU logique exclusif
MPI_BXOR	OU bit à bit exclusif
MPI_MAXLOC	Maximum et où
MPI_MINLOC	Minimum et où

Topologies virtuelles

- Dans de nombreuses applications, les processus sont rangés avec une topologie particulière (ex. grille 2D).
- C'est un moyen pratique de nommer les processus.
- Le schéma de nommage correspond au schéma de communication.
- Cela simplifie l'écriture du code.
- Permet à MPI d'optimiser les communications.

TOPOLOGIES VIRTUELLES

Topologies virtuelles

- MPI supporte les topologies virtuelles par un graphe dans lequel les processus sont connectés par un arc.
- MPI supporte également de manière explicite les grilles cartésiennes.

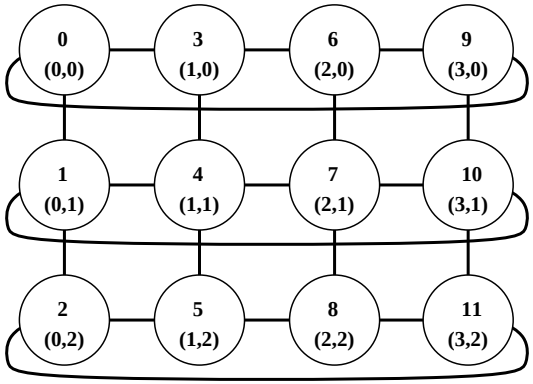
```
MPI_Cart_create( comm_old, ndims, dims,
                 period, reorder, comm_cart)
```

- La périodicité dans chaque direction peut être spécifiée.
- Des routines transforment le rang dans le groupe dans les coordonnées dans la topologie.
- La création de topologie produit un nouveau communicateur.

Routines de coordonnées de topologies

- `MPI_Topo_test` retourne le type de la topologie associée au communicateur.
- On peut obtenir le nombre de dimensions d'une topologie cartésienne:
  `MPI_Cartdim_test(comm, ndims)`
- Plus d'informations sur une topologie cartésienne peuvent être obtenues avec:
  `MPI_Cart_get(comm, maxdims, periods, coords)`
- "Mappage" de coordonnées vers le rang:
  `MPI_Cart_rank(comm, coords, rank)`
- "Mappage" d'un rang vers des coordonnées:
  `MPI_Cart_coords(comm, rank, maxdims, coords)`

Exemple: tore 2D



Utilisation des topologies

- La connaissance d'une topologie peut être utilisée pour assigner de manière précise les processus sur les processeurs.
- Les grilles cartésiennes peuvent être divisées en hyperplans en supprimant certaines dimensions.
- MPI offre des routines pour effectuer des *shifts* sur une dimension d'une grille cartésienne.
- MPI offre la possibilité de faire des communications collectives sur une dimension d'une grille.

Partitionnement de topologies cartésiennes

- Une top. cart. peut être partitionnée en un ensemble de top. cart. de dimension inférieure, en utilisant:

  `MPI_Cart_sub(comm, remaindims, newcomm)`

  `remaindims[i]` est VRAI si la $i$ème dimension est conservée et à FAUX sinon.

- **Exemple**:

  Si une topologie 2x3x4 est associée à `comm`, et

  `remaindims = (vrai, vrai, faux)`

  alors l'appel à `MPI_Cart_sub`

  - crée 4 communicateurs, chacun avec une topologie cartésienne 2x3,

  - la valeur de `newcomm` retournée est celle contenant le processus appelant dans son groupe associé.

PACK ET UNPACK

Topologies et shifts

- Soient les *shifts* suivants pour un groupe de taille N:

  - *shift* **circulaire** de J. Les données dans le processus K sont envoyées au processus $(\mathrm{mod}(J+K), N)$

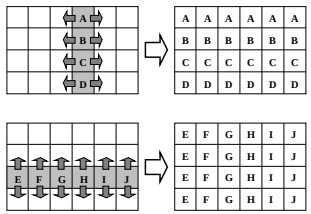
  - *shift* **non-circulaire** de J. Les données dans le processus K sont envoyées au processus $J+K$ entre 0 et $N-1$; sinon, aucune donnée n'est envoyée.

- Les *shifts* sont effectués à l'aide de `MPI_Sendrecv`.

- `MPI_Cart_shift` retourne les rangs des processus auxquels un processus doit envoyer et recevoir quand il effectue un *shift* sur un groupe de processus.

Topologies et communications collectives

- Supposons que nous voulons effectuer une communication collective le long d'une dimension d'une topologie (*multicast*):



- Nous pouvons appeler `MPI_Cart_sub` avec `remaindims = (faux, vrai)` pour générer les sous-communicateurs pour les lignes (1er cas),

- si `remaindims = (vrai, faux)`, on obtient les sous-communicateurs pour le deuxième cas.

Pack et Unpack

- MPI offre des routines pour:

  - Emballer (*pack*) des données dans un buffer contigu avant de l'envoyer,

  - déballer (*unpack*) des données d'un buffer après réception.

- Ces routines sont fournies:

  - pour assurer la compatibilité avec d'autres bibliothèques de communication (ex: PVM),

  - pour permettre de recevoir un message en morceaux,

  - pour bufferiser les messages dans l'espace utilisateur et ainsi éviter la bufferisation système.

Emballage de données

- Pour emballer des données:

```
MPI_Pack( inbuf, incount, datatype,
          outbuf, outsize, position, comm)
```

  Cet appel prend `incount` éléments de type `datatype` du buffer `inbuf` et les emballe dans `outbuf` en partant de l'offset `position`.

- L'offset `position` est spécifié en octets.

- Au retour, `position` est positionné sur l'élément suivant de `outbuf`.

Déballage de données

- Pour déballer des données:

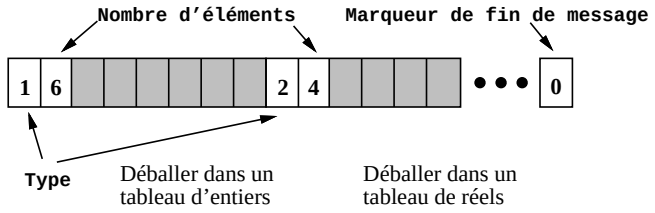
```
MPI_Unpack( inbuf, insize, position,
            outbuf, outcount, datatype, comm)
```

  Cet appel extrait `outcount` éléments de type `datatype` du buffer `inbuf` en partant de l'offset `position` et les range dans `outbuf`.

- Au retour, `position` est positionné sur l'élément suivant de `inbuf`.

Exemple de déballage

- Un message consiste en des séquences de réels ou d'entiers précédés par un identificateur de type et un nombre d'éléments.



es entiers sont déballés avec:

```
call mpi_unpack( message, size, pos, ints(nints)
                 nitems, MPI_INTEGER, comm, ierr)
nints = nints + nitems
```

es réels sont déballés avec:

```
call mpi_unpack( message, size, pos, reals(nreals)
                 nitems, MPI_REAL, comm, ierr)
nreals = nreals + nitems
```



```
ll mpi_recv(  message, size, MPI_PACKED, MPI_ANY_SOURCE,
              MPI_ANY_TAG, comm, status, ierr)
os = 0
ll mpi_unpack(  message, size, pos, typeid, 1
                MPI_INTEGER, comm, ierr)
o while (typeid .gt. 0)
   call mpi_unpack(  message, size, pos, nitems, 1,
                     MPI_INTEGER, comm, ierr)
   if (typeid .eq. 1) then
      UNPACK INTEGERS
   else
      UNPACK REALS
   end if
   call mpi_unpack(  message, size, pos, typeid, 1,
                     MPI_INTEGER, comm, ierr)
d do
```


La prise de temps

- On peut prendre le temps de manière portable grâce à MPI.:

```
MPI_Wtime()
```

- Le temps est pris en secondes.

AUTRES FONCTIONS

Ensemble minimal

Début/fin	`MPI_Init()` `MPI_Finalize()`
Qui suis-je?	`MPI_Comm_rank()` `MPI_Comm_size()`
Envoi	`MPI_Send()`

Réception	`MPI_Recv()`
	`stat.MPI_SOURCE`
	`stat.MPI_TAG`
	`MPI_Get_count()`

Types de base	`MPI_CHAR`, `MPI_SHORT`, `MPI_INT` `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`

ENSEMBLE MINIMAL

MPI sur réseaux hétérogènes

- MPI offre un certain support pour le calcul sur réseaux hétérogènes.

  - Les types fournis aux routines de communication permettent les conversions entre les différentes représentations internes.

  - Le support pour les groupes de processus autorise différents clusters de machines à travailler sur des tâches distinctes.

Mais:

- MPI n'offre aucun moyen de spécifier les ressources physiques à utiliser.
- MPI offre un support limité pour attribuer les processus aux processeurs.

IMPLÉMENTATIONS DE MPI

LE FUTUR DE MPI

Implémentations de MPI

Chameleon	Argonne National Lab.	Plateformes supportant CMMD, MPI, MPF/EUI, NX, P4, PICL, PVM
MPICH	Argonne National Lab.	Toutes les plateformes Chameleon, P4 et PVM. SUN, RS6000 et SP-1, Intel iPSC et Paragon, CM5, nCUBE, Silicon Graphics
Chimp	Edinburgh Univ.	SUN 4 (SunOS et Solaris), SGI (IRIX 4 et 5), IBM RS6000, Sequent Symmetry, DEC Alpha AXP, Meiko CS-1 et CS-2, Fujitsu AP1000
LAM	Ohio state Univ.	Toutes les plateformes UNIX, SUN4 (SunOS et Solaris), SGI, RS6000, DEC AXP, HP
UNIFY	Mississippi State Univ.	Plateformes PVM, SUN4 (SunOS), SGI
IBM (MPI-F)		SP-1 et 2
Fujitsu		AP1000
Cray		T3D

utres implémentations de Intel, Hughes Aircraft, NEC, ...

MPI-F2: le second forum MPI

- Ca démarre!!!!
- Il faut y participer!!!!
- On peut utiliser l'e-mail!!!!

MPI I/O

- Proposition faite par la NASA et IBM.
- On peut encore discuter.

CONCLUSION MPI

Manipulation de processus

- Proposition faite.
- On peut encore discuter.

PVM

le futur de MPI

- Pour l'instant, MPI a des omissions (voulues):
  - pas de gestion de processus
  - pas de spécificités F90 ou C++
  - pas de support pour la tolérance aux pannes
  - pas de support pour les Entrées/Sorties
  - pas de support pour les messages actifs
  - etc ...
- Le nouveau forum MPI va regarder ces problèmes
- Les implémentations commencent à arriver en masse!

- PVM = Standard de fait
- MPI = standard réfléchi

PVM VS MPI

Pourquoi utiliser PVM?

Trois raisons principales:

- **Pour lier ensemble des stations**

  machine parallèle du pauvre

- **Pour écrire des programmes portables**

  Stations de travail/MPP/calcul hétérogène

- **Tout le monde l'utilise**

  Pas forcément une mauvaise raison!

Pourquoi ne pas utiliser PVM?

Performances faibles:

- Bufferisation excessive.
- Communications collectives inefficaces.
- Surcoût (par rapport aux bibliothèques natives).

Manque de fonctionnalités:

- pas de vraies communications asynchrones.
- pas de bon support pour les groupes.
- pas de support pour le développement de bibliothèques.

Manque de standardisation

Communications collectives

our:

- PVM possède de nombreuses communications collectives.

ontre:

- A cause des groupes dynamiques, chaque communication collective fait appel au serveur de groupes → **Lent!**

Nouvelles fonctionnalités dans PVM 3.3

VM 3.3 résoud certains de ces problèmes:

- **Le** *Inplace packing* **est implémenté.**

  Plus de bufferisation côté émetteur.

- `pvm_psend` **agrège l'emballage et l'envoi**.

  Moins de surcoût.

- `pvm_precv` **agrège la réception et le déballage.**

Mais pas de *inplace unpack*, donc pas de gains sur les stations.

Points forts et points faibles de PVM

Points forts:

- Contrôle/configuration de machines/tâches.
- Support pour le calcul hétérogène.
- Très utilisé.

Points faibles:

- Performances.
- Fonctionnalités des communications.
- Stabilité/standardisation.
- Support pour les Entrées/Sorties.

Les contextes

- Pas de contextes dans PVM 3!

PVM/MPI: la conclusion

VM:

- PVM a été un outil formidable pour le développement du parallélisme.
- De nombreuses idées ont été reprises dans d'autres environnements (y compris MPI).

IPI:

- Si MPI continue à se développer de cette manière, il faut l'utiliser.
- MPI est disponible pour les réseaux de stations et bientôt pour toutes les machines parallèles.
- MPI peut être implémenté efficacement.
- Beaucoup (trop?) de fonctionnalités mais il n'est pas utile de **tout** connaître pour commencer à développer des programmes.

Points forts et points faibles de MPI

oints forts:

- Intérêt pour le développement de bibliothèques.
- Grande étendue.
- N'empêche pas les performances.
- Est un standard (*ou va le devenir!*).
- Va être très utilisé.

oints faibles:

- Approche *CISC* de l'échange de messages.
- Support pour les processus/machines dynamiques.
- Support pour les Entrées/Sorties.

Conclusion

- Le passage de messages a encore de beaux jours devant lui.
- De nouveaux langages et outils font leur apparition.
- Mais leur implémentation va nécessiter une interface de communication portable.

- Il faut insister auprès des constructeurs pour qu'ils implémentent MPI efficacement.
- Il faut participer au forum MPI2 et dans les news pour que cette interface soit celle que l'on souhaite.

CONCLUSION

Mes coordonnées

Frédéric DESPREZ

LaBRI
Université Bordeaux I
351, Cours de la Libération
33405 TALENCE Cedex
Tél: 56 84 69 86 **Fax:** 56 84 66 69

e-mail: desprez@labri.u-bordeaux.fr

URL: http://www/htbin/trombine?desprez

COORDONNÉES