

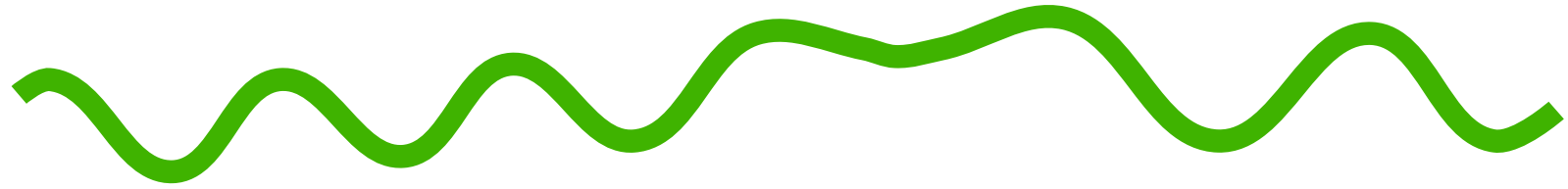
Introduction à MPI – Message Passing Interface

***Outils pour le calcul scientifique à haute performance
École doctorale sciences pour l'ingénieur
mai 2001***

Philippe MARQUET

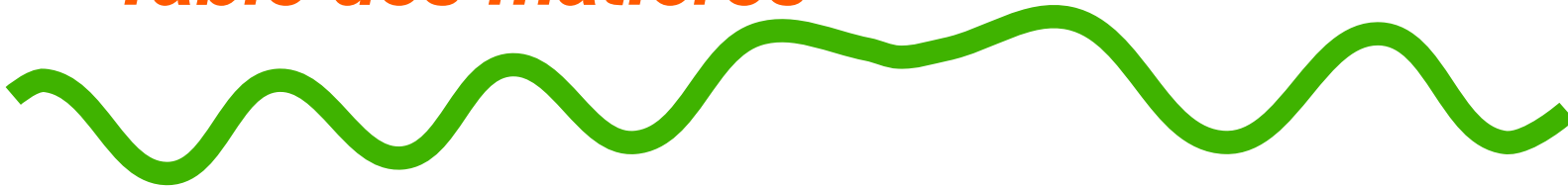
phm@lifl.fr

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille



- ~ Ce cours est diffusé sous la licence GNU Free Documentation License,
<http://www.gnu.org/copyleft/fdl.html>
- ~ La dernière version de ce cours est accessible à partir de
<http://www.lifl.fr/west/courses/cshp/>
- ~ \$Id: mpi.tex,v 1.11 2002/04/29 07:32:58 marquet Exp \$

Table des matières

- 
- ~ Hello world
 - ~ Une application
 - ~ Communications collectives
 - ~ Regroupement des données
 - ~ Communicateurs
 - ~ Différents modes de communications
 - ~ Et maintenant ?
 - ~ Compilation et exécution de programmes MPI

Remerciements



Cette présentation de MPI est essentiellement basée sur

~ *A User's Guide to MPI*

Peter S. PACHERO

University of San Francisco

<ftp://math.usfca.edu/pub/MPI/>

~ *Designing & Building Parallel Programs*

Chapitre *Message Passing Interface*

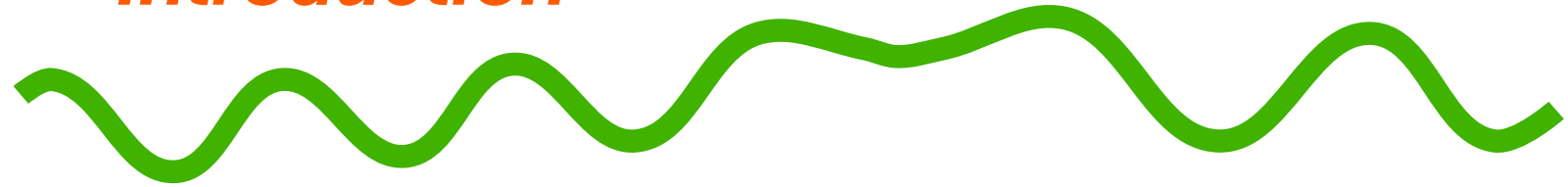
Ian FORSTER

Argonne National Laboratory

<http://www.mcs.anl.gov/dbpp>

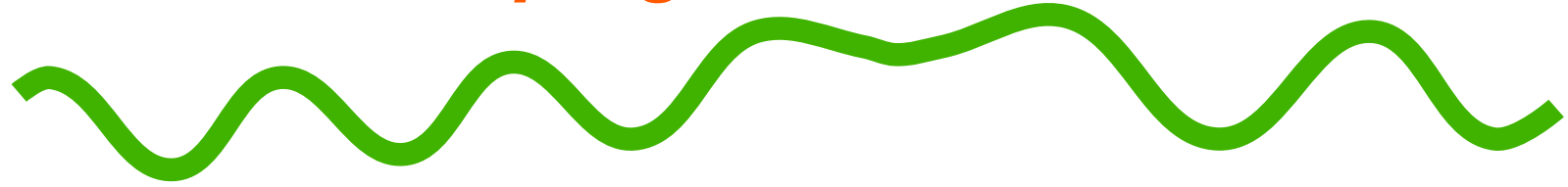
<http://www.mcs.anl.gov/dbpp/web-tours/mpi.html>

Introduction

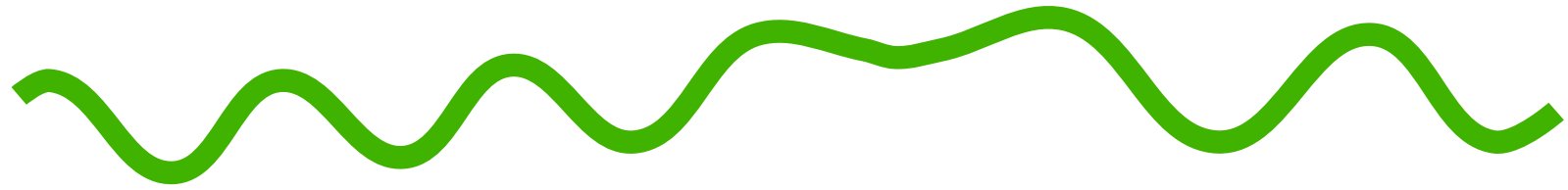


- ✓ MPI : *Message-Passing Interface*
 - ✓ Bibliothèque de fonctions utilisables depuis C, Fortran, C++
 - ✓ Exploitation des machines multi-processeurs par passage de messages
 - ✓ Conçue en 1993–94 → standard
- ✓ Cette présentation :
 - ✓ Grandes lignes de MPI
 - ✓ Exemples simples
 - ✓ Utilisation de MPI depuis C

Modèle de programmation

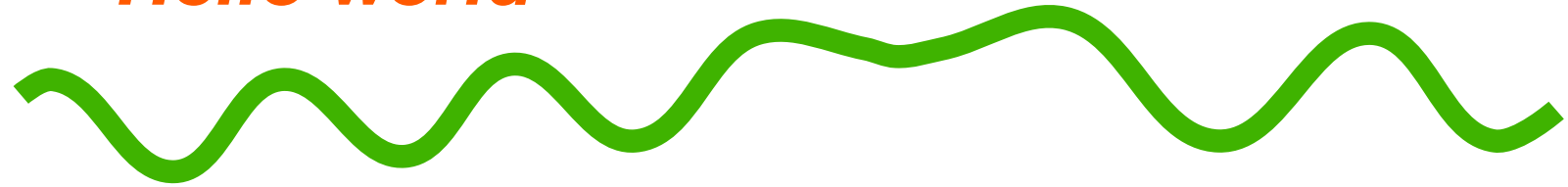


- ✓ Modèle de programmation de MPI
 - ✓ parallélisme de tâches
 - ✓ communication par passage de messages
- ✓ Même programme exécuté au lancement de l'application par un ensemble de processus
 - ✓ SPMD *Single Program Multiple Data*
 - ✓ M-SPMD *Multiple-SPMD*
- ✓ Un seul programme = un seul code source, celui d'un processus

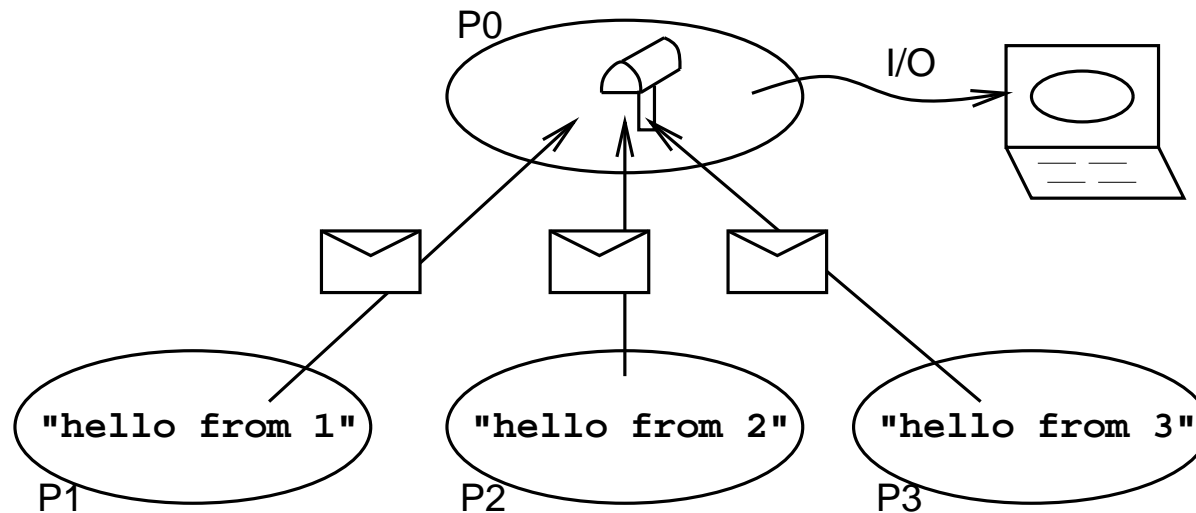


Hello world

Hello world

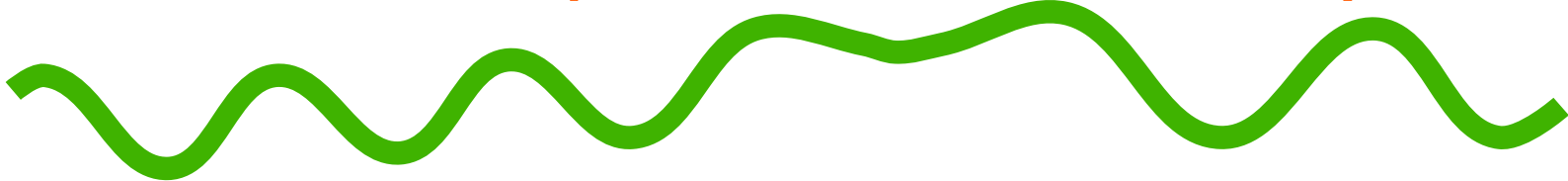


- ~ p processus : P_0 à P_{p-1}
- ~ Les processus $P_i, i>0$ envoient un message (chaîne de caractères) à P_0
- ~ P_0 reçoit $p - 1$ messages et les affiche



- ~ Programmation SPMD (*Single Program Multiple Data*) : Suis-je le P_0 ?

Hello World (code C, initialisation)

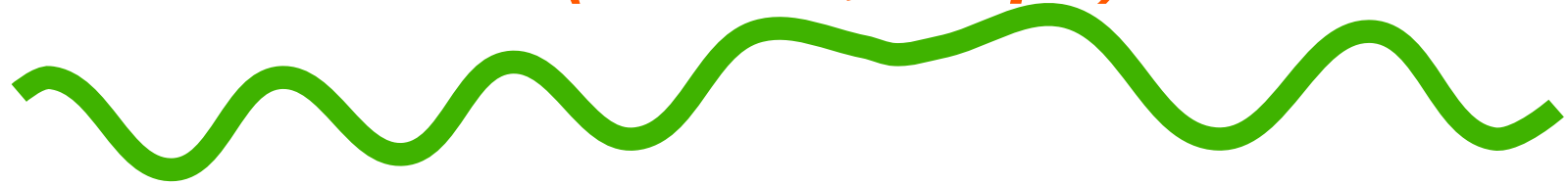


```
#include <stdio.h>
#include "mpi.h"

main(int argc, char *argv[]) {
    int my_rank;          /* Rang du processus */
    int p;               /* Nombre de processus */
    int source;         /* Rang de l'emetteur */
    int dest;           /* Rang du recepneur */
    int tag = 50;       /* Tag des messages */
    char message[100];  /* Allocation du message */
    MPI_Status status;  /* Valeur de retour pour le recepneur */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

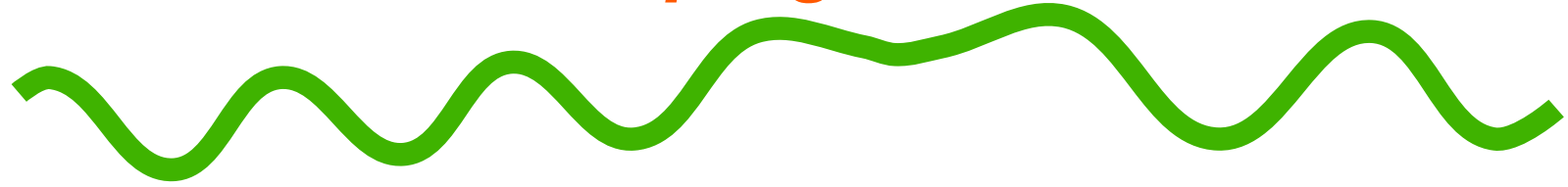
Hello World (code C, corps)



```
if (my_rank != 0) {
    /* Creation du message */
    sprintf(message, "Hello from process %d!", my_rank);
    dest = 0;
    /* strlen + 1 => le '\0' final est envoye */
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
             tag, MPI_COMM_WORLD);
} else { /* my_rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag,
                 MPI_COMM_WORLD, &status);
        printf("%s\n", message);
    }
}

MPI_Finalize();
} /* main */
```

Structure d'un programme MPI



- Utilisation de la bibliothèque MPI :
 - Enrollement dans MPI
 - Quitter MPI proprement

```
...
#include "mpi.h"
...
main (int argc, char *argv []) {
    ...
    MPI_Init (&argc, &argv) ;
    ...
    MPI_Finalize () ;
    ...
}
```

Qui ? Combien ?



~ Qui suis-je ?

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

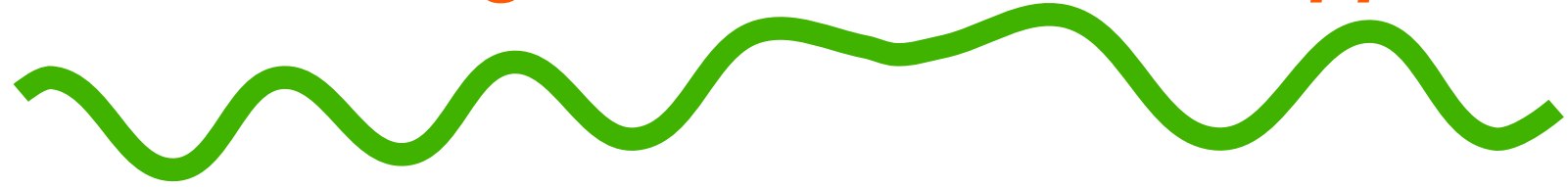
~ Communicateur : collection de processus pouvant communiquer

~ Communicateur MPI_COMM_WORLD prédéfini : tous les processus

~ Combien sommes-nous ?

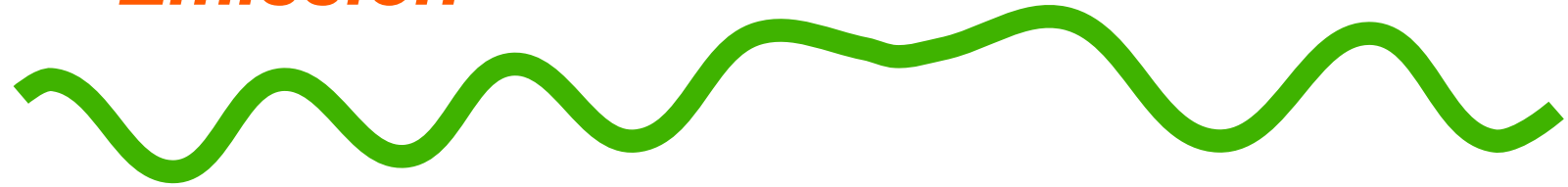
```
int MPI_Comm_size (MPI_Comm comm, int *size) ;
```

Un message = données + enveloppe



- ~ Fonctions de base d'émission (`MPI_Send ()`) et de réception (`MPI_Recv ()`) de messages
- ~ Enveloppe : informations nécessaires
 - ~ Rang du receveur (pour une émission)
 - ~ Rang de l'émetteur (pour une réception)
 - ~ Un tag (`int`)
 - ⇒ Distinguer les messages d'un même couple émetteur/receveur
 - ~ Un communicateur (`MPI_Comm`)
 - ⇒ Distinguer les couples de processus

Émission



~ Émission d'un message

(standard, bloquant ou non selon l'implantation)

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

~ Message =

~ enveloppe

~ données :

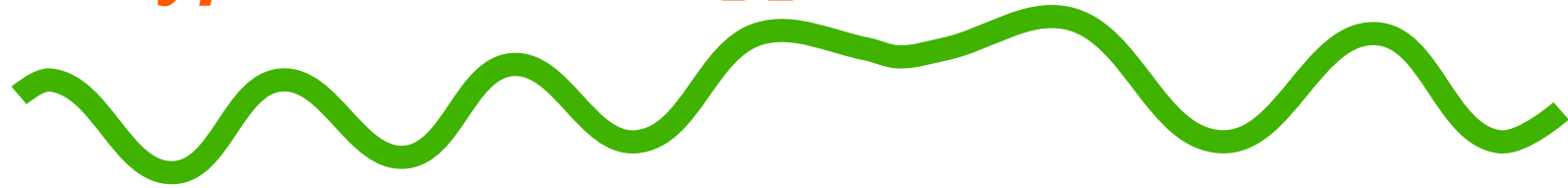
~ bloc de mémoire d'adresse message

~ de count valeurs

~ de type datatype

~ Correspondance des MPI_Datatype avec les types du langage (C ou Fortran)

Type MPI_Datatype



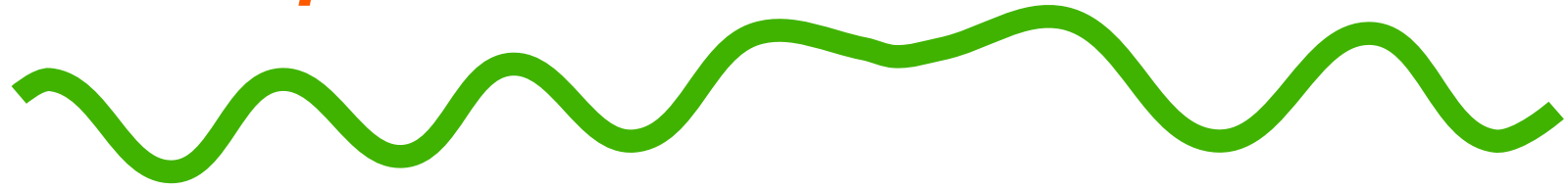
~ Correspondance des MPI_Datatype avec les types du C :

MPI_CHAR	signed char	MPI_SHORT	signed short int
MPI_INT	signed int	MPI_LONG	signed long int
...			
MPI_FLOAT	float	MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double	MPI_PACKED	<< struct >>

~ Types particuliers :

- ~ MPI_BYTE Pas de conversion
- ~ MPI_PACKED Types construits (*cf. infra*)

Réception



~ Réception d'un message

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

~ Attend la réception d'un message

~ Écrit

- ~ count valeurs
- ~ de type MPI_Datatype
- ~ à partir de l'adresse message

~ Message reçu du processus source (dans le communicateur comm)

~ Message reçu de tag tag

Réception « anonyme »



Message reçu d'un processus `source` quelconque

Joker pour `source` : `MPI_ANY_SOURCE`

Pas de joker pour le communicateur

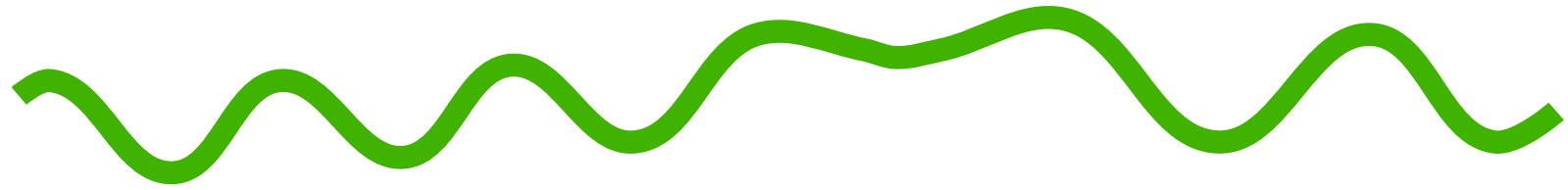
Message reçu de tag `tag` quelconque

Joker : `MPI_ANY_TAG`

Accès au tag et à l'émetteur : `status`

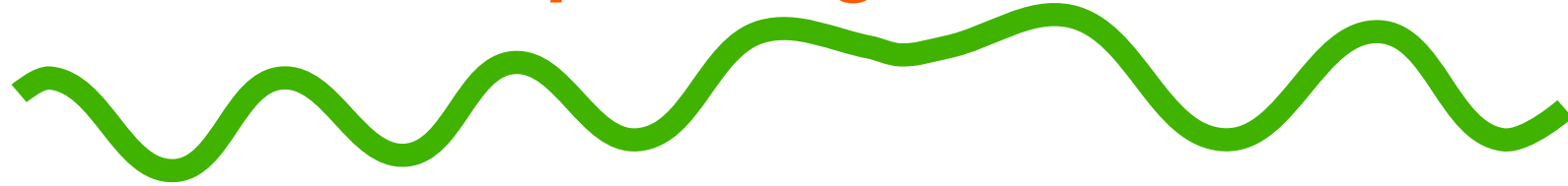
dans `mpi.h` :

```
typedef struct {  
    int MPI_SOURCE;  
    int MPI_TAG;  
} MPI_Status;
```



Une application

Calcul de π par intégration

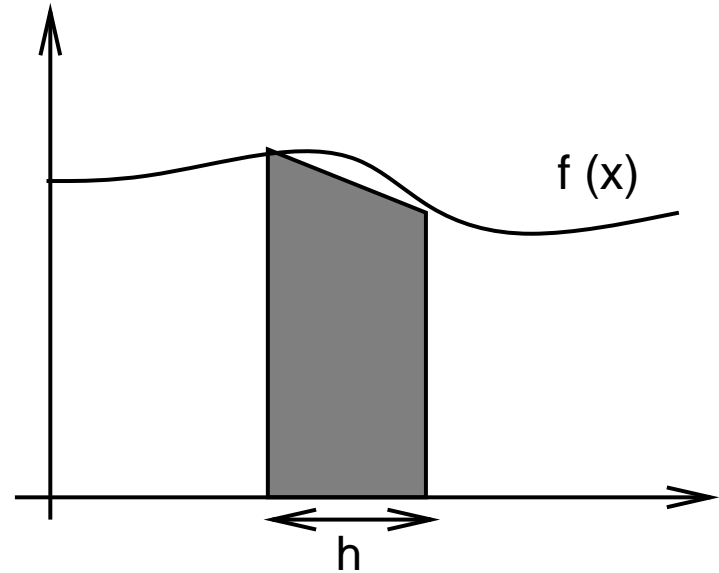


Étant donné que

$$\begin{aligned}\pi &= \int_0^1 \frac{4}{1+x^2} dx \\ &= \int_a^b f(x) dx\end{aligned}$$

On peut approximer sa valeur par

$$\sum_{i=a,b,h} h * \frac{f(i) + f(i+h)}{2}$$



Programme séquentiel

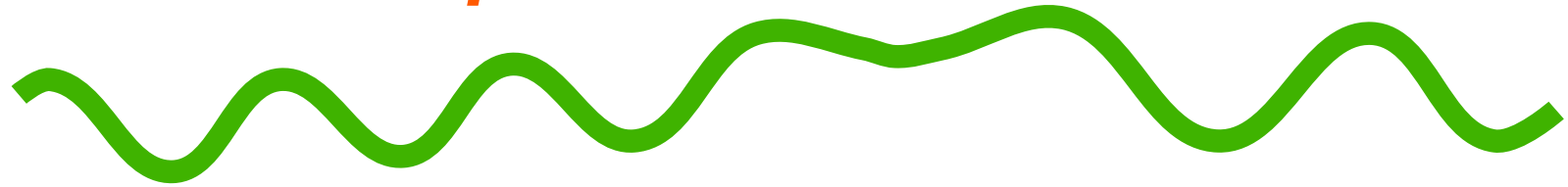
```
float f (float x) {
    float return_val ;
    /* calcul de f(x) */
    return return_val ;
}

main () {
    /* resultat */
    float integral ;
    /* points gauche et droit */
    float a, b ;
    /* nombre de trapezes */
    int n ;
    /* hauteur d'un trapeze */
    float h ;
```

```
float x ; int i ;

printf ("Donner a, b, et n\n")
scanf ("%f %f %d", &a, &b, &n)
h = (b-a)/n ;
integral = (f(a) + f(b))/2 ;
for (i=1, x=a ; i<=n-1 ; i++)
    x += h ;
    integral += f(x) ;
}
integral *= h ;
printf ("Estimation : %f\n",
        integral) ;
}
```

Première parallélisation



- Idée : partager l'intervalle $[a, b]$ entre les processus
- p processus
 n (nombre de trapèzes) est divisible par p
- Le processus q estime la valeur de l'intégrale sur

$$\left[a + q \frac{nh}{p}, a + (q + 1) \frac{nh}{p} \right]$$

- Il a besoin de
 - $p \rightarrow$ `MPI_Comm_size ()`
 - $q \rightarrow$ `MPI_Comm_rank ()`
 - $a, b, n \rightarrow$ codage en dur dans le programme (à suivre...)
- Collecte par le processus 0 des différentes intégrales

Calcul local d'une intégrale



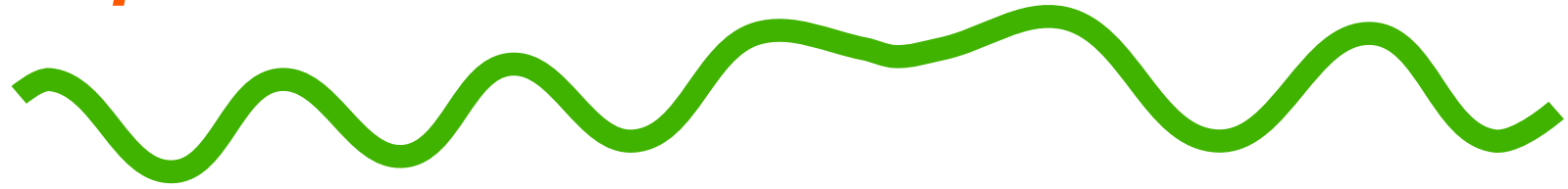
```
float Trap (float local_a,  
           float local_b,  
           int local_n,  
           float h) {  
    float integral; /* resultat */  
    float x;  
    int i;  
  
    integral = (f(local_a) + f(local_b))/2.0;  
    x = local_a;  
    for (i = 1; i <= local_n-1; i++) {  
        x += h;  
        integral += f(x);  
    }  
    integral *= h;  
    return integral;  
} /* Trap */
```

Déclarations



```
main(int argc, char* argv[]) {
    int my_rank;        /* Mon rang */
    int p;              /* Nombre de processus */
    float a = 0.0;     /* Extremite gauche */
    float b = 1.0;     /* Extremite droite */
    int n = 1024;      /* Nombre de trapezes */
    float h;           /* Largeur du trapeze */
    float local_a;     /* Extremite gauche pour mon processus */
    float local_b;     /* Extremite droite pour mon processus */
    int local_n;       /* Nombre de trapeze pour mon calcul */
    float integral;    /* Integrale sur mon intervalle */
    float total;       /* Integrale totale */
    int source;        /* Emetteur */
    int dest = 0;      /* Tous les messages vont au processus 0 */
    int tag = 50;
    MPI_Status status;
```

Calculs locaux pour tous les processus



```
/* On utilise MPI */
MPI_Init(&argc, &argv);

/* Qui suis-je ? */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Combien sommes-nous ? */
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = (b-a)/n;    /* h est identique pour tous les processus
local_n = n/p; /* idem pour le nombre de trapezes */

/* Longueur de chaque intervalle d'integration : local_n*h
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```


Collecte des résultats



Le processus 0 collecte les résultats

```
if (my_rank == 0) {
    total = integral;
    for (source = 1; source < p; source++)
    {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
        total += integral;
    }
}
```

Les autres lui retournent leur résultat local

```
else {
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
}
```

Affichage du résultat



~ Affichage du résultat

```
if (my_rank == 0) {  
    printf("Estimation : %f\n", total);  
}
```

~ Terminaison

```
    MPI_Finalize();  
} /* main */
```

Quid des entrées/sorties ?

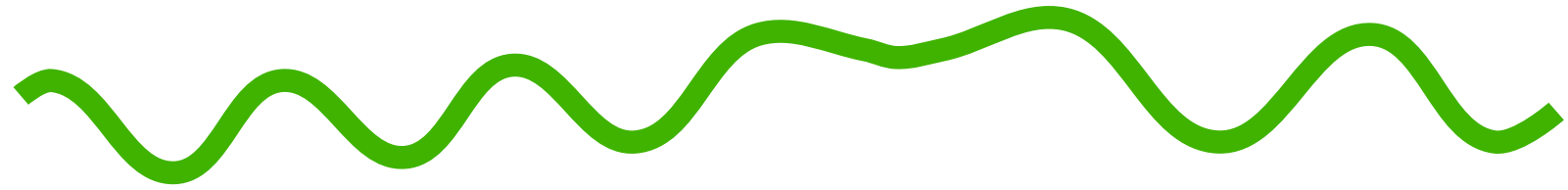


- ✓ On a supposé que le processus 0 réalisait les (I/O)
- ✓ On a codé en dur les I/O dans le programme
- ✓ Pose problème sur des implémentations de MPI sur machines parallèles (I/O parallèles...)
- ✓ Écriture d'une fonction d'I/O
- ✓ Implémentation de cette fonction sur réseaux de stations de travail :
 - ✓ Le processus 0 lit les données
 - ✓ Il les diffuse aux autres processus
- ✓ Utilisation de cette fonction en mode SPMD (appel par tous les processus) :

```
void Get_Data (int my_rank,  
              float *a_ptr, float *b_ptr, int *n_ptr) ;
```

Get_data ()

```
void Get_data(int my_rank, int p, float *a_ptr, float *b_ptr, int *n_ptr)
{
    int source = 0;
    int dest;
    int tag = 30;
    MPI_Status status;
    if (my_rank == source){
        printf("Donner a, b, et n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest < p; dest++){
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(n_ptr, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
    }
} /* Get_data */
```



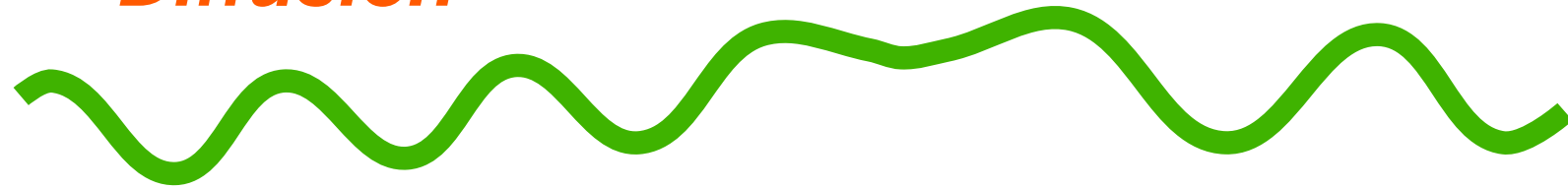
Communications collectives

Communication arborescente

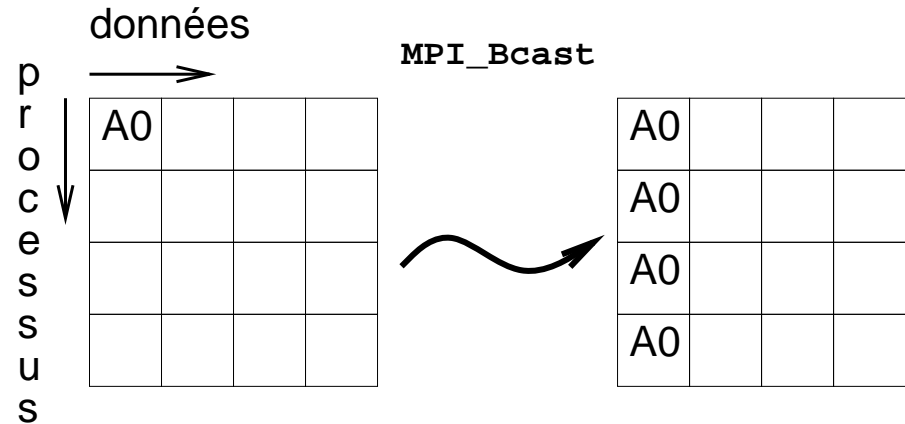


- ↯ Deux communications
 - ↯ « un vers tous » : `Get_data ()`
 - ↯ « tous vers un » : collecte des résultats
- ↯ Communication arborescente
 - ↯ Complexité en $\mathcal{O}(p)$ \rightsquigarrow complexité en $\mathcal{O}(\log_2(p))$
- ↯ Quelle structure pour mon arbre ?
 - Dépend de la structure de la machine parallèle
 - ⇒ utilisation de fonctions MPI
- ↯ Communication collective = fonction SPMD

Diffusion



- Un même processus envoie une même valeur à tous les autres processus (d'un communicateur)



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

- Appel par tous les processus du communicateur `comm` avec la même valeur de `root`, `count`, `datatype`
- La valeur détenue par le processus `root` sera émise et rangée chez chacun des autres processus
- ⇒ Réécriture de `Get_data ()`

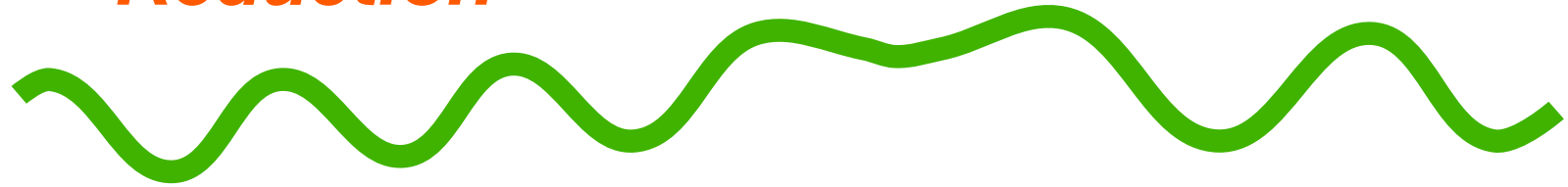
Get_data (), version 2



```
void Get_data2(int my_rank,
               float *a_ptr, float *b_ptr, int *n_ptr) {
    int root = 0;
    int count = 1;

    if (my_rank == root)
    {
        printf("Donner a, b, et n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, root, MPI_COMM_WORLD);
} /* Get_data2 */
```


Réduction

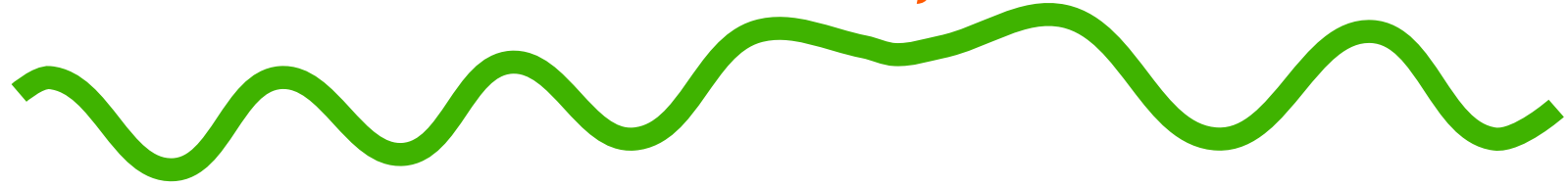


- ~ Collecte par un processus d'un ensemble de valeurs détenues par tous les processus
- ~ Réduction de cette valeur
- ~ Fonction SPMD

```
int MPI_Reduce (void *operand, void *result, int count,  
               MPI_Datatype datatype, MPI_Op op,  
               int root, MPI_Comm comm) ;
```

- ~ Appel par tous les processus du communicateur `comm` avec une même valeur de `count`, `datatype`, `op`
- ~ Opérations binaires prédéfinies par MPI (`MPI_MAX`, `MPI_SUM`...)
- ~ Possibilité de définir de nouvelles opérations
- ~ Le processus `root` détient le résultat
- ~ \rightsquigarrow Réécriture de la collecte globale du résultat

Collecte des résultats, version 2



```
Get_data2(my_rank, &a, &b, &n);

h = (b-a)/n;    /* h est identique pour tous les processus
local_n = n/p; /* idem pour le nombre de trapezes */

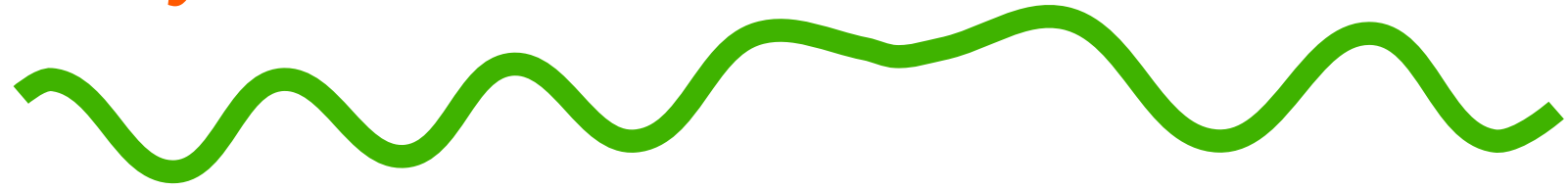
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

MPI_Reduce(&integral, &total, 1,
           MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

if (my_rank == 0) {
    printf("Estimation : %f\n", total);
}
```

Autres communications collectives

Synchronisation



- ~ Synchronisation ou rendez-vous
- ~ Pas d'échange d'informations
- ~ Tous les processus sont assurés que tous ont raliés le *point de synchronisation*

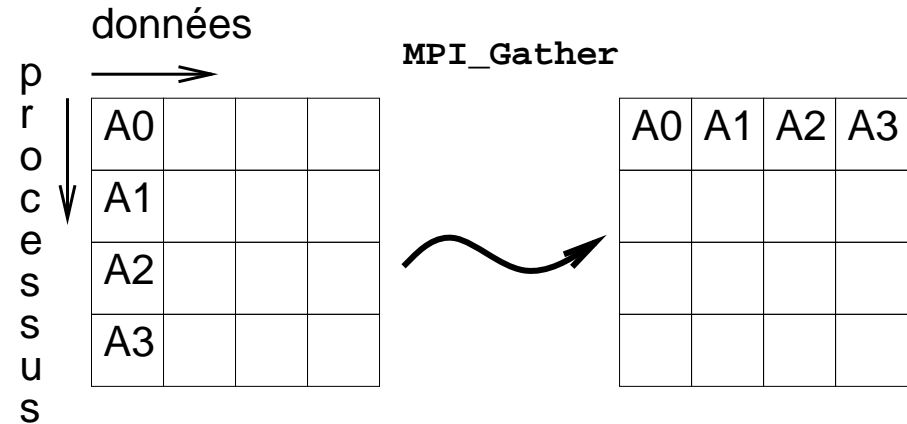
```
int MPI_Barrier (MPI_Comm comm) ;
```

Autres communications collectives

Rassemblement



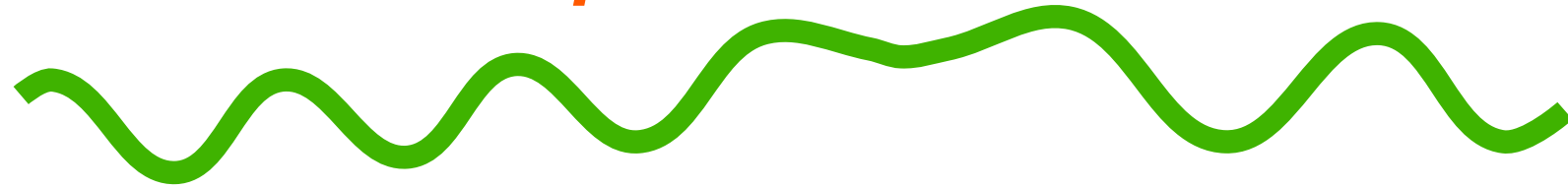
- « All to one »
- Mise bout à bout des messages de chacun des processus



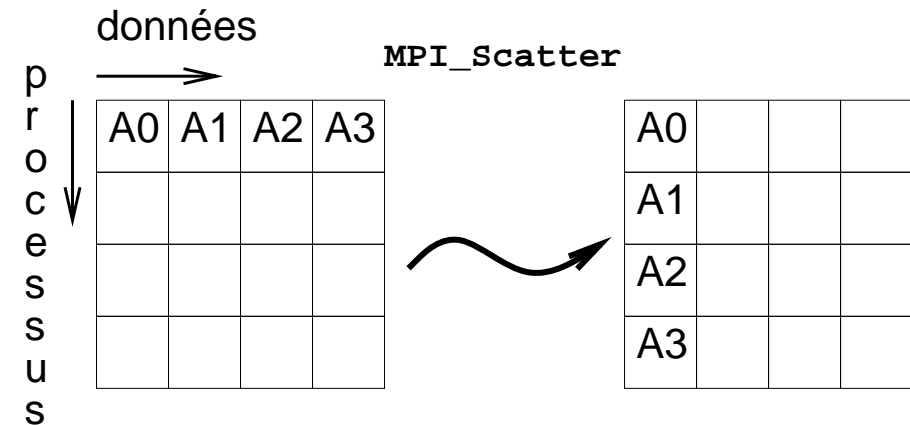
```
int MPI_Gather (void *send_buf, int send_count,  
               MPI_Datatype send_type,  
               void *recv_buf, int recv_count,  
               MPI_Datatype recv_type,  
               int root, MPI_Comm comm) ;
```

Autres communications collectives

Distribution personnalisée



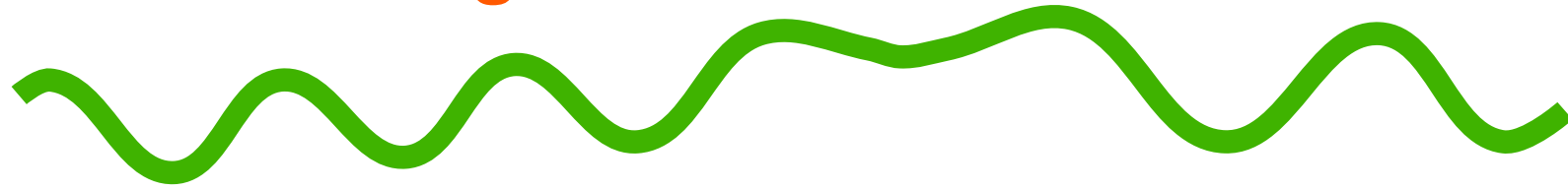
- « One to all »
- Distribution d'un message personnalisé aux autres processus



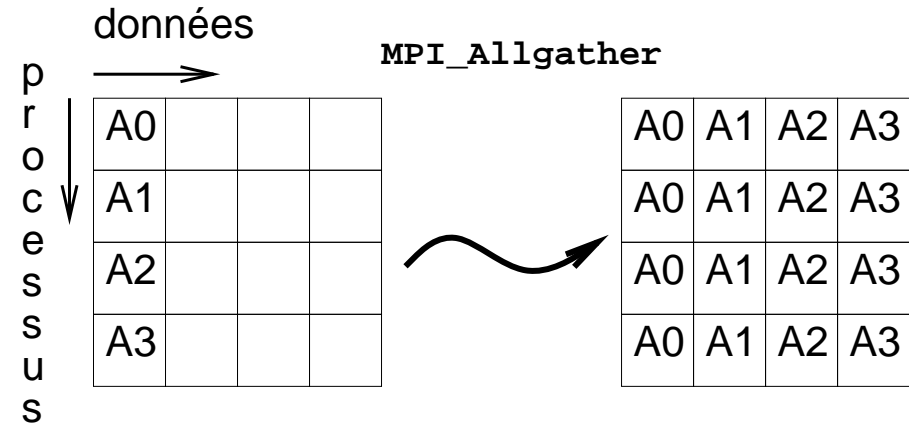
```
int MPI_Scatter (void *send_buf, int send_count,  
                MPI_Datatype send_type,  
                void *recv_buf, int recv_count,  
                MPI_Datatype recv_type,  
                int root, MPI_Comm comm) ;
```

Autres communications collectives

Comméragage



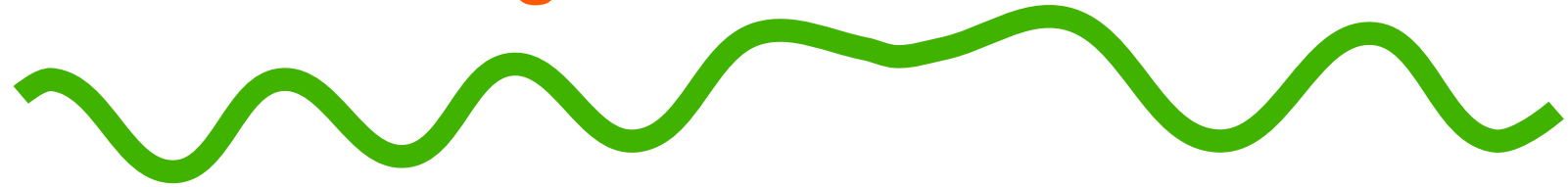
- « All to all »
- Mise bout à bout des messages de chacun des processus
- Résultat dans chacun des processus



```
int MPI_Allgather (void *send_buf, int send_count,  
MPI_Datatype send_type,  
void *recv_buf, int recv_count,  
MPI_Datatype recv_type,  
MPI_Comm comm) ;
```

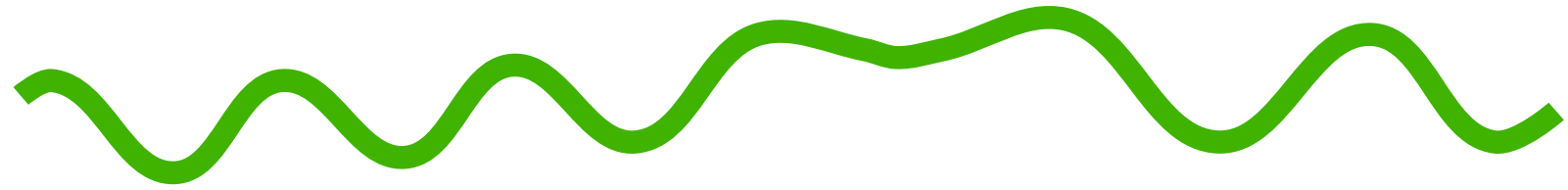
Autres communications collectives

Réduction généralisée



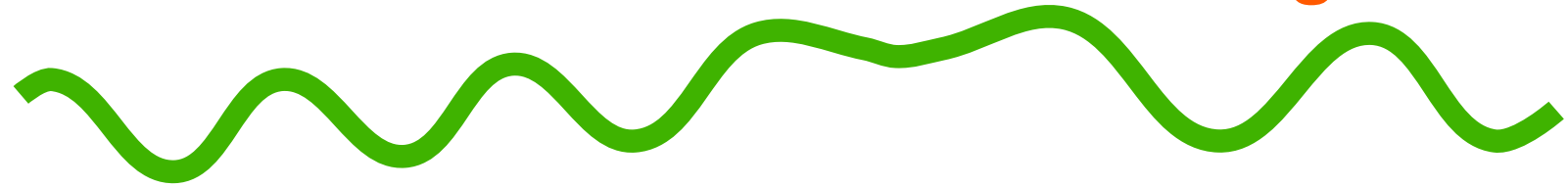
~ Réduction généralisée : réduction + résultat dans chacun des processus

```
int MPI_Allreduce (void *operand, void *result, int count,  
                  MPI_Datatype datatype, MPI_Op op,  
                  MPI_Comm comm) ;
```



Regroupement des données

Réduction du nombre de messages



- ✓ Grouper les données dans un message
 - diminuer le nombre de message
 - augmenter les performances
- ✓ Paramètre `count` de `MPI_Send`, `MPI_Recv`...
Grouper les données
 - ✓ de même type
 - ✓ contiguë en mémoiredans un message
- ✓ Autres possibilités de MPI plus générales :
 - ✓ Notion de type dérivé MPI
 - ✓ Compactage/décompactage des données (à la PVM)

Type dérivé MPI



✓ Dans notre exemple : envoyer float a, b et int n

✓ Créer un type struct :

```
typedef struct {  
    float a ;  
    float b ;  
    int n ;  
} INDATA_TYPE ;
```

✓ Problème : MPI ne connaît rien de ce type INDATA_TYPE

Il n'existe pas de type MPI prédéfini correspondant à INDATA_TYPE

✓ Solution : créer un type MPI à l'exécution

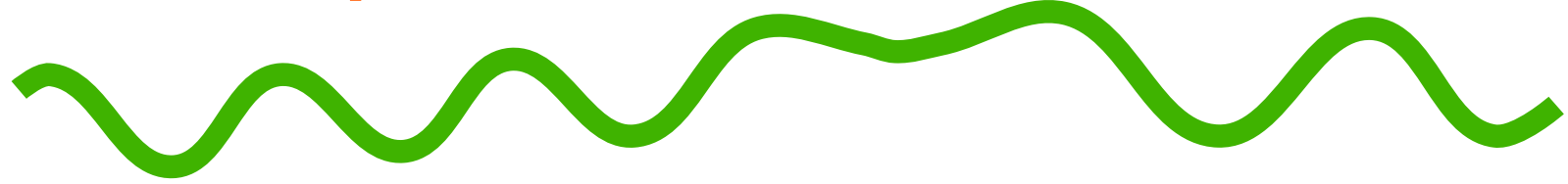
↪ Type dérivé MPI

Précise pour chacun des membres

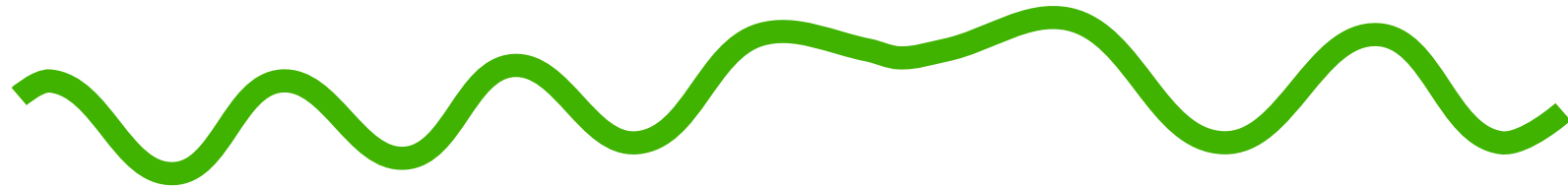
✓ son type

✓ son adresse mémoire relative

Création d'un type dérivé correspondant à *INDATA_TYPE*



```
void Build_derived_type (INDATA_TYPE* indata,  
                        MPI_Datatype* message_type_ptr) {  
    int block_lengths[3];  
    MPI_Aint displacements[3];  
    MPI_Aint addresses[4];  
    MPI_Datatype typelist[3];  
  
    /* Specification des types */  
    typelist[0] = MPI_FLOAT;  
    typelist[1] = MPI_FLOAT;  
    typelist[2] = MPI_INT;  
  
    /* Specification du nombre d'elements de chaque type */  
    block_lengths[0] = block_lengths[1] =  
        block_lengths[2] = 1;
```



```
/* Calcul du deplacement de chacun des membres
 * relativement a indata */
MPI_Address(indata, &addresses[0]);
MPI_Address(&(indata->a), &addresses[1]);
MPI_Address(&(indata->b), &addresses[2]);
MPI_Address(&(indata->n), &addresses[3]);
displacements[0] = addresses[1] - addresses[0];
displacements[1] = addresses[2] - addresses[0];
displacements[2] = addresses[3] - addresses[0];

/* Creation du type derive */
MPI_Type_struct(3, block_lengths, displacements, typelist,
               message_type_ptr);

/* Remise du type pour qu'il puisse etre utilise */
MPI_Type_commit(message_type_ptr);
} /* Build_derived_type */
```

Utilisation du type dérivé



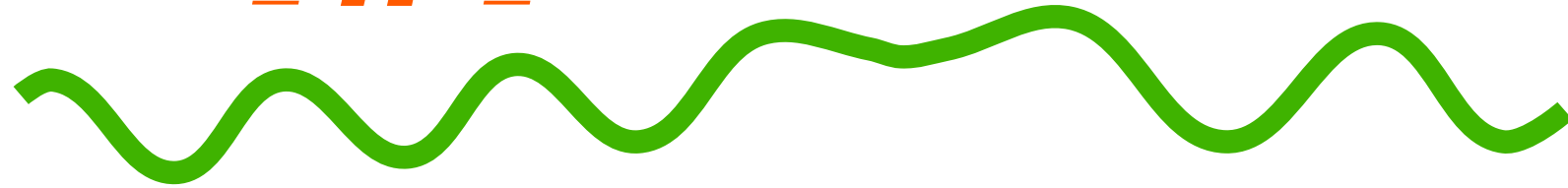
```
void Get_data3(int my_rank, INDATA_TYPE* indata) {
    MPI_Datatype message_type;
    int root = 0;
    int count = 1;

    if (my_rank == root) {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d",
            &(indata->a), &(indata->b), &(indata->n));
    }

    Build_derived_type(indata, &message_type);
    MPI_Bcast(indata, count, message_type,
        root, MPI_COMM_WORLD);
} /* Get_data3 */
```

Construction de type dérivé avec

MPI_Type_Struct



```
int MPI_Type_struct (int count, int *array_of_block_lengths,  
                    MPI_Aint *array_of_displacements,  
                    MPI_Aint *array_of_types,  
                    MPI_Datatype *newtype) ;
```

count : nombre d'éléments du type dérivé

C'est aussi la taille des 3 tableaux `array_of_block_lengths`,
`array_of_displacements`, `array_of_types`

array_of_block_lengths : nombre d'entrées de chacun des types

array_of_displacements : déplacement de chaque élément relativement au
début du message

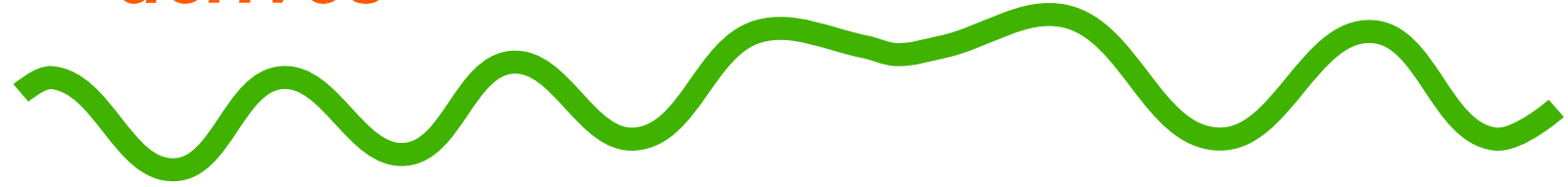
array_of_types : type MPI de chaque entrée

newtype : résultat

~ `newtype` et `array_of_type` sont de type `MPI_Datatype`

~ Possibilité d'appel récursif à `MPI_Type_Struct` pour la création de types plus complexes

Autres constructeurs de types dérivés



- Construction d'un type dérivé d'éléments contigus dans un tableau

```
int MPI_Type_contiguous (int count,  
                        MPI_Datatype element_type,  
                        MPI_Datatype *newtype) ;
```

- Construction d'un type dérivé d'éléments régulièrement espacés dans un tableaux

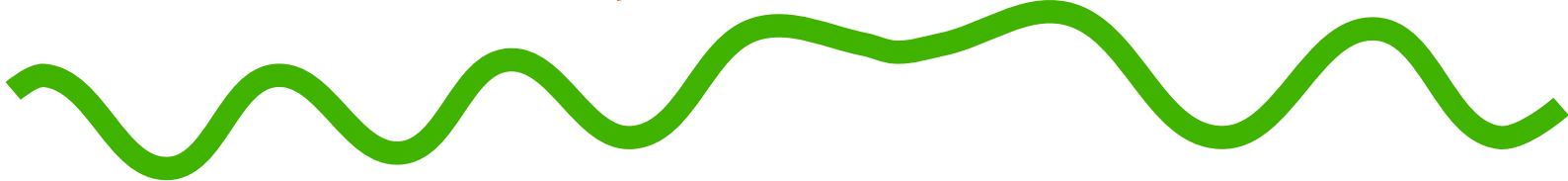
```
int MPI_Type_vector (int count, int block_length,  
                   int stride,  
                   MPI_Datatype element_type,  
                   MPI_Datatype *newtype) ;
```

- Construction d'un type dérivé d'éléments arbitraires dans un tableau

```
int MPI_Type_indexed (int count, int *array_of_block_lengths,  
                    int *array_of_displacements,  
                    MPI_Datatype element_type,  
                    MPI_Datatype *newtype) ;
```

Compactage/décompactage

Get_data (), version 4

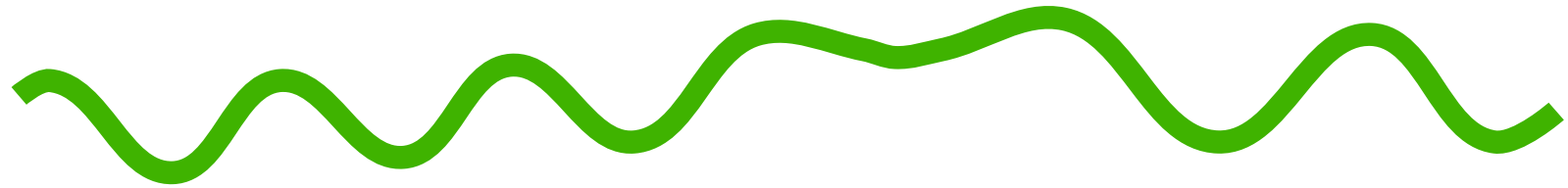


```
void Get_data4(int my_rank, float* a_ptr, float* b_ptr,
               int* n_ptr){
    int root = 0, position ;
    char buffer[100];

    if (my_rank == root){
        printf("Donner a, b, et n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

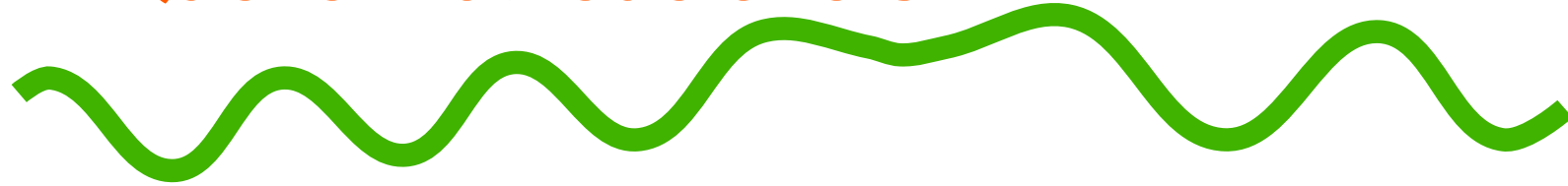
        /* Pack des donnees dans le buffer */
        position = 0; /* On commence au debut du buffer */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
        /* position a ete incremente de sizeof(float) bytes */
        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
        MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);

        /* Diffusion du contenu du buffer */
        MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
    }
}
```

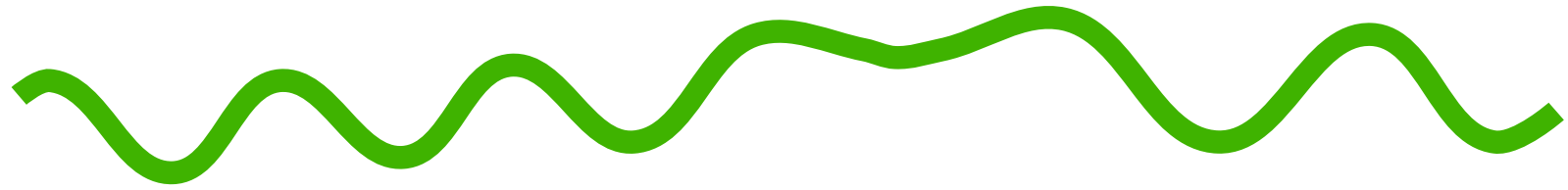



```
} else {  
    MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);  
  
    /* Unpack des donnees depuis le buffer */  
    position = 0;  
    MPI_Unpack(buffer, 100, &position, a_ptr, 1,  
               MPI_FLOAT, MPI_COMM_WORLD);  
    /* De mme, position a ete incremente de sizeof(float) bytes  
    MPI_Unpack(buffer, 100, &position, b_ptr, 1,  
               MPI_FLOAT, MPI_COMM_WORLD);  
    MPI_Unpack(buffer, 100, &position, n_ptr, 1,  
               MPI_INT, MPI_COMM_WORLD);  
}  
} /* Get_data4 */
```

Quelle méthode choisir ?

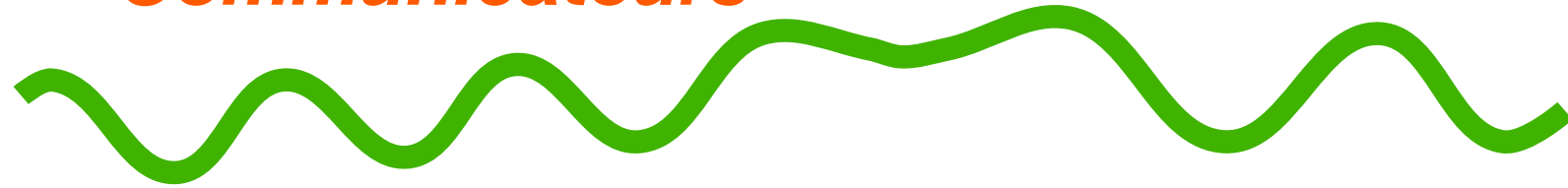


- ✓ Efficacité : utilisation de `count` de `MPI_Send` avec un type MPI prédéfini
- ✓ Généralité : type dérivé MPI ou compactage
- ✓ Type dérivé : surcoût de la création du type
- ✓ Compactage : surcoût lors de chaque compactage/décompactage
- ✓ Exemple sur nCUBE2, MPICH
 - ✓ Création du type dérivé (`Get_data3 ()`) : 12 millisecondes
 - ✓ Compactage des données (`Get_data4 ()`) : 2 millisecondes
- ✓ Autres avantages du compactage :
 - ✓ Buffer dans la mémoire utilisateur
 - ✓ Possibilité de manipuler des messages de taille variable



Communicateurs

Communicateurs



- ✔ Groupe de processus pouvant communiquer
- ✔ Communicateur prédéfini : `MPI_COMM_WORLD`
Tous les processus de l'application
- ✔ MPI : modèle « MPMD »
« M » de « MPDP » ⇒ identification de « sous-groupes » de processus
- ✔ Quatre fonctions principales :
 - ✔ `MPI_Comm_dup ()` Création d'un nouveau contexte
 - ✔ `MPI_Comm_split ()` Création de « sous-communicateurs »
 - ✔ `MPI_Intercomm_create ()` Création d'un communicateurs reliant deux communicateurs existants
 - ✔ `MPI_Comm_free ()` Destruction d'un communicateur

Création d'un communicateur

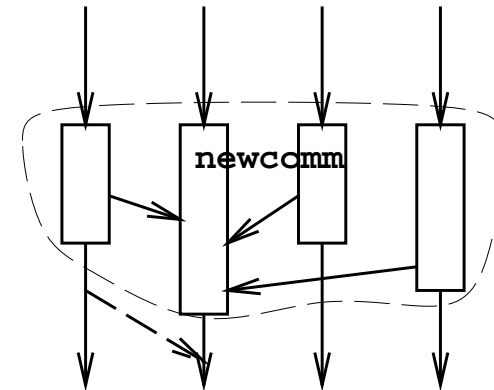
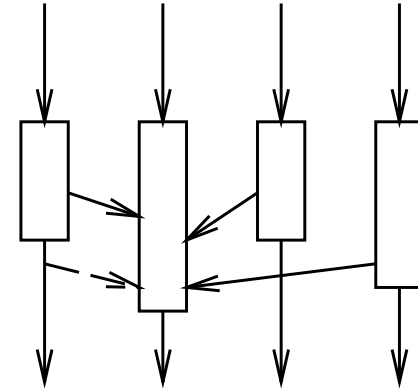
~ Distinguer les messages utilisés entre deux phases distinctes de l'algorithme

⇒ Duplication d'un communicateur `comm` existant

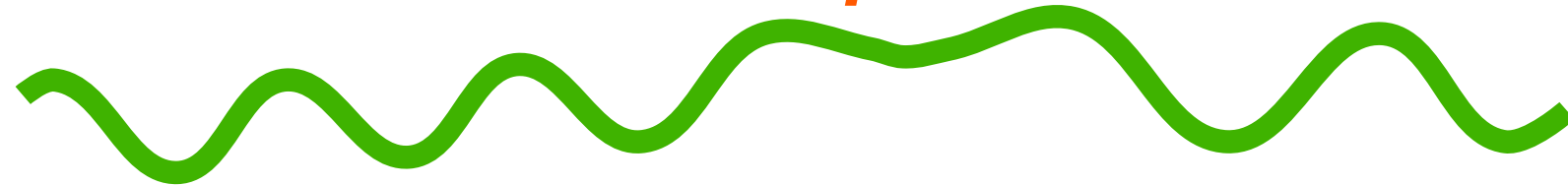
```
int MPI_Comm_dup (MPI_Comm comm,  
                 MPI_Comm *newcomm)
```

~ Exemple :

```
MPI_Comm comm, newcomm ;  
...  
MPI_Comm_dup (comm, &newcomm) ;  
Transpose (newcomm, A) ;  
MPI_Comm_free (&newcomm) ;
```



Partitionnement de processus



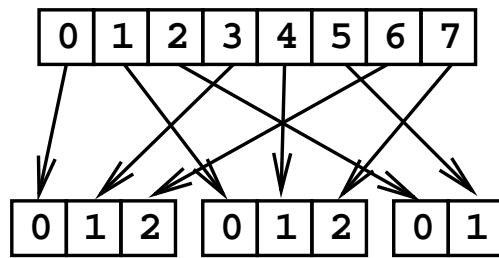
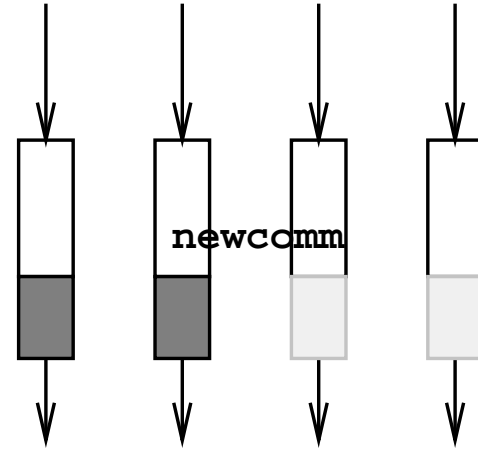
~ Séparer les processus d'un communicateur en groupes (processus de même couleur)

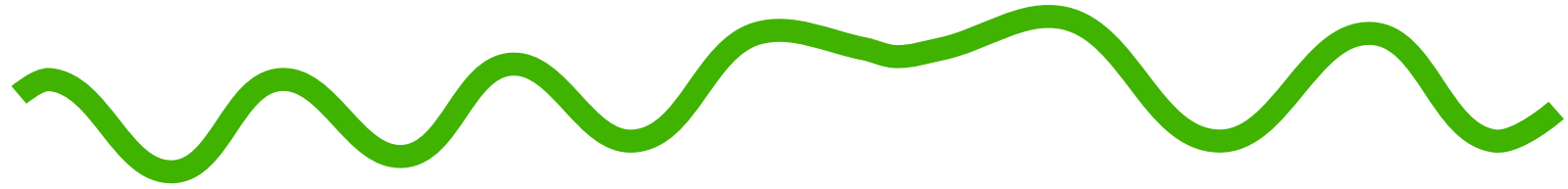
```
MPI_comm_split (MPI_Comm comm,  
               int color, int key,  
               MPI_Comm *newcomm)
```

La clé `key` permet de calculer le rang dans le nouveau groupe

~ Exemple : création de 3 communicateurs (si + de 3 processus)

```
MPI_Comm comm, newcomm ;  
int myid, color ;  
MPI_Comm_rank (comm, &myid) ;  
color = myid%3 ;  
MPI_Comm_split (comm, color, myid, &newcomm) ;
```



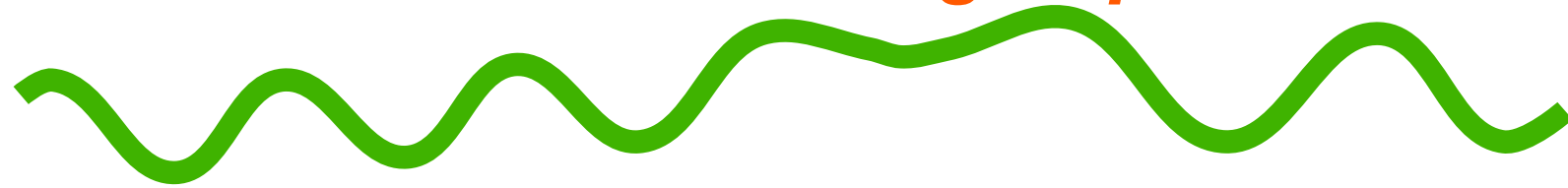


~ Exemple : création d'un groupe d'au plus 8 processus

Couleur `MPI_UNDEFINED` \Rightarrow \notin nouveau communicateur

```
MPI_Comm comm, newcomm ;  
int myid, color ;  
MPI_Comm_rank (comm, &myid) ;  
if (myid < 8)      /* les 8 premiers */  
    color = 1 ;  
else                /* les autres ne sont pas dans le groupe */  
    color = MPI_UNDEFINED ;  
MPI_Comm_split (comm, color, myid, &newcomm) ;
```

Communication entre groupes



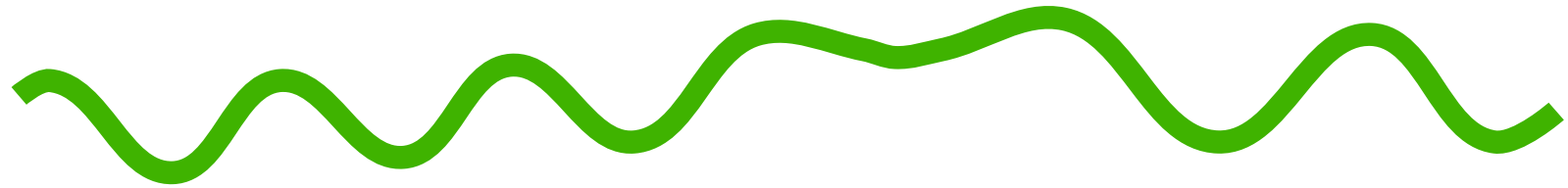
- ✓ *Intra*-communicateur : communication entre les processus du groupe
- ✓ *Inter*-communicateur entre deux groupes A et B
- ✓ Communication pour chaque processus de A avec les N_B processus de B

`MPI_Send () . . .` mais pas de communication collective

✓ Nécessite :

- ✓ Un groupe ancêtre commun (au pire `MPI_COMM_WORLD`)
Établissement des communications via ce groupe
- ✓ Un *leader* dans chaque groupe
Donné par son numéro dans le groupe ancêtre
- ✓ Un tag non-utilisé pour établir les communications entre les leaders

```
MPI_Intercomm_create (MPI_Comm local_comm, int local_leader,  
                     MPI_Comm ancetre_comm, int remote_leader,  
                     int tag, MPI_Comm &newintercomm)
```

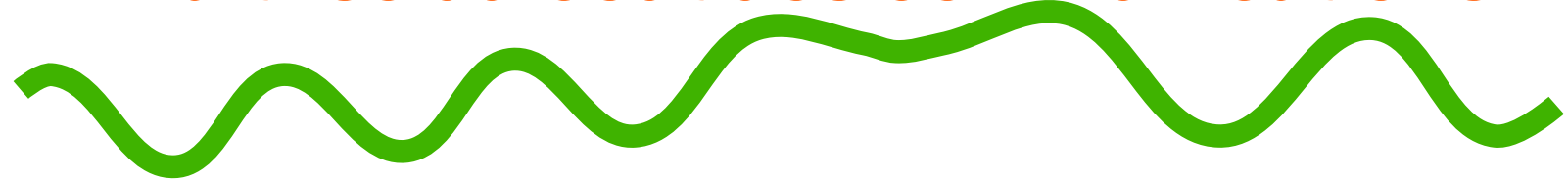
Différents modes de communications

Rappel sur les modes de communications



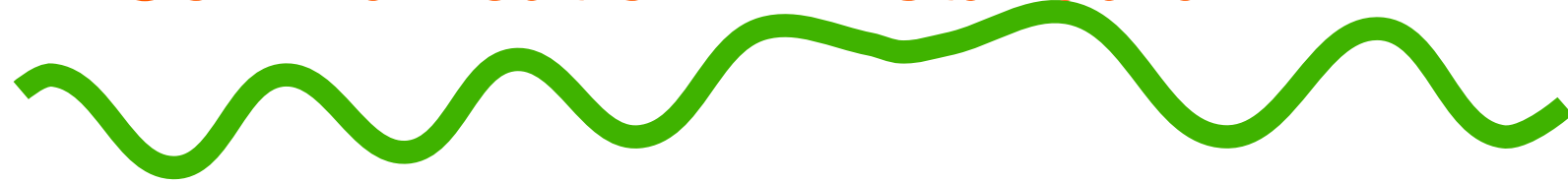
- ✓ Synchrones
 - ✓ rendez-vous entre l'émetteur et le récepteur
- ✓ Asynchrone
 - ✓ émission dès que le message est prêt
 - ✓ réception postérieure à l'émission
- ✓ Asynchrone non bloquante
 - ✓ rendre la main au plus vite à l'appelant
 - ✓ demande d'exécution de communication

Maîtrise du coût des communications



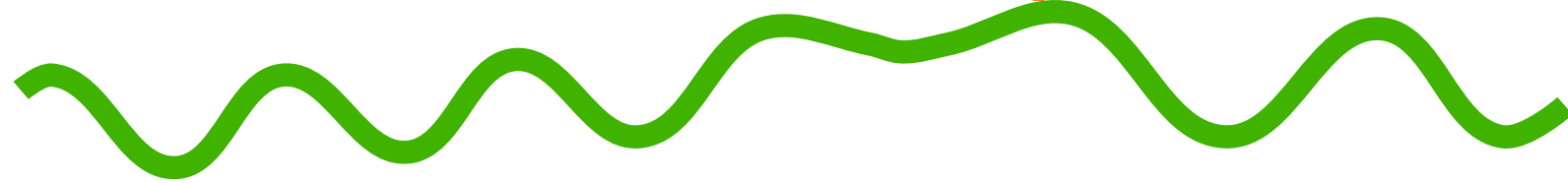
- ✓ Maîtrise du coût des communications essentiel
- ✓ Optimisation des communications :
 - ✓ recouvrir les communication par des calculs
 - ✓ éviter des copies multiples de données entre émission et réception
 - ✓ factoriser les appels répétitifs à la bibliothèque avec les mêmes paramètres

Communication MPI standard



- ✓ MPI_Send () et MPI_Recv ()
- ✓ Deux implantations possibles
 - ✓ copie du message dans un *buffer*
 - ✓ communication bloquante
- ✓ Recopie dans un buffer
 - ✓ permet de libérer l'émetteur dès la copie faite
 - ✓ coût de cette copie
 - ✓ protocole utilisée sur IBM si message inférieur à MP_EAGER_LIMIT octets
- ✓ Rendez-vous
 - ✓ évite la recopie
 - ✓ communication **synchrone**
 - ✓ attention aux étreintes fatales (*dead-lock*)
 - ✓ protocole utilisée sur IBM si message supérieur à MP_EAGER_LIMIT octets

Communication non bloquante



~ Recouvrir le coût des communications

~ L'appel bibliothèque initie la communication

~ Émission : copie des données et transmission auront lieu plus tard

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

~ Réception : lecture du message et recopie dans le buffer se feront plus tard

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

~ Attente de la terminaison de la communication : MPI_Request

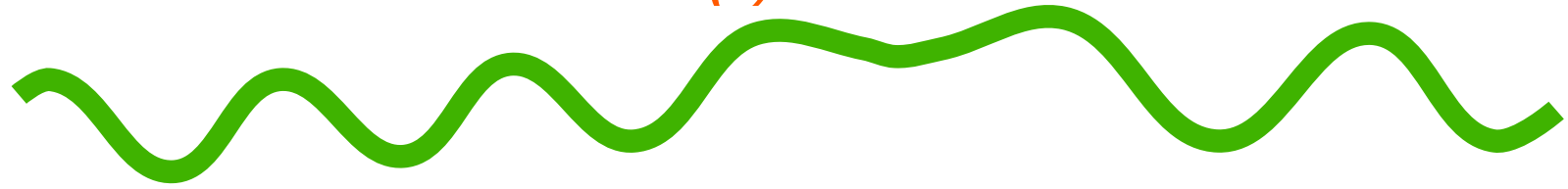
Attente de terminaison d'une communication (1)



- ~ Communication asynchrone initiée
- ~ Attente de la terminaison de la communication
 - ~ pour ré-utiliser le buffer, ou
 - ~ pour accéder au message
- ~ Attente bloquante de la terminaison d'une ou plusieurs émissions ou réceptions asynchrones

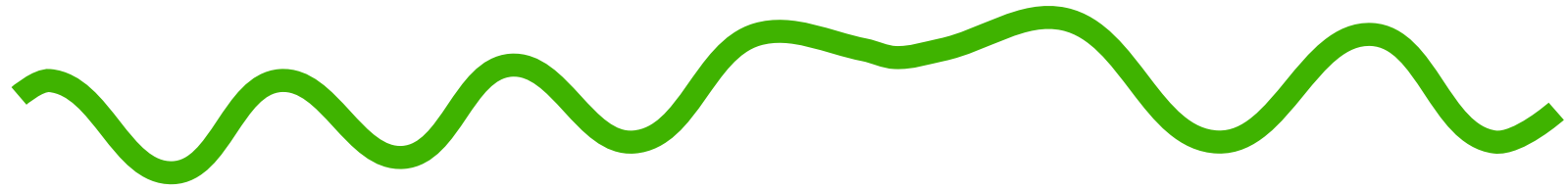
```
int MPI_Wait (MPI_Request *request, MPI_Status *status);  
int MPI_Waitany (int count, MPI_Request *array_of_requests,  
                int *index, MPI_Status *status);  
int MPI_Waitall (int count, MPI_Request *array_of_requests,  
                MPI_Status *array_of_statuses);
```

Attente de terminaison d'une communication (2)



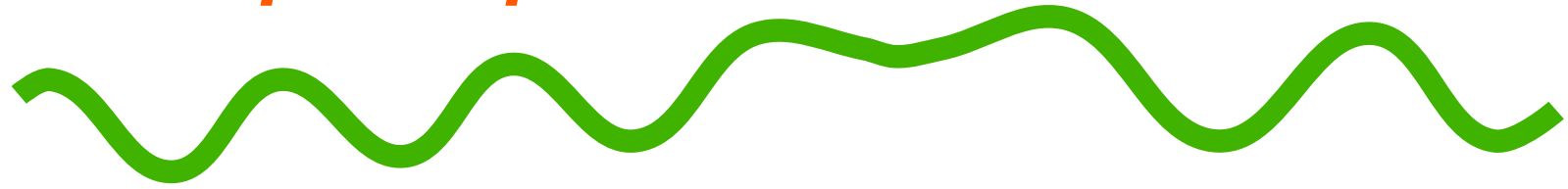
~ Test de la terminaison d'une ou plusieurs émissions ou réceptions asynchrones

```
int MPI_Test (MPI_Request *request,  
             int *flag, MPI_Status *status);  
int MPI_Testany (int count, MPI_Request *array_of_requests,  
               int *index, int *flag, MPI_Status *status);  
int MPI_Testall (int count, MPI_Request *array_of_requests,  
               int *flag, MPI_Status *array_of_statuses);  
flag est mis à vrai ssi une communication a terminée
```



Et maintenant ?

Ce qui n'a pas été vu

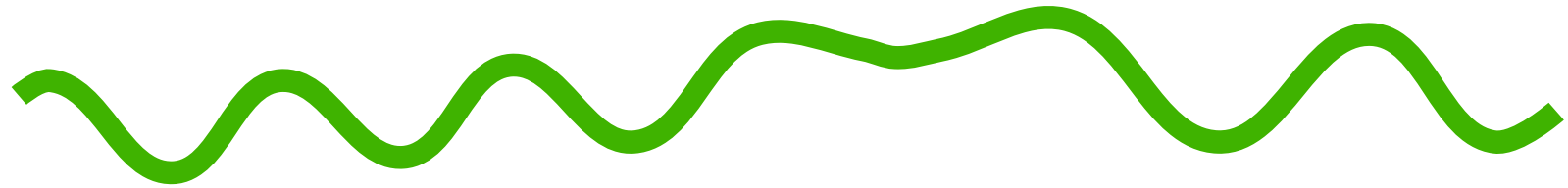


- ✓ MPI : plus de 130 fonctions
- ✓ Communication bufferisées
 - ✓ gestion du buffer de recopie par l'utilisateur
- ✓ Topologies virtuelles de processus
- ✓ Interface Fortran
- ✓ Extension MPI-2
 - ✓ gestion dynamique de processus
 - ✓ entrée/sortie parallèles (ex MPI-IO)
 - ✓ interface Fortran 95
 - ✓ quelques changement de noms dans l'API

Documentations

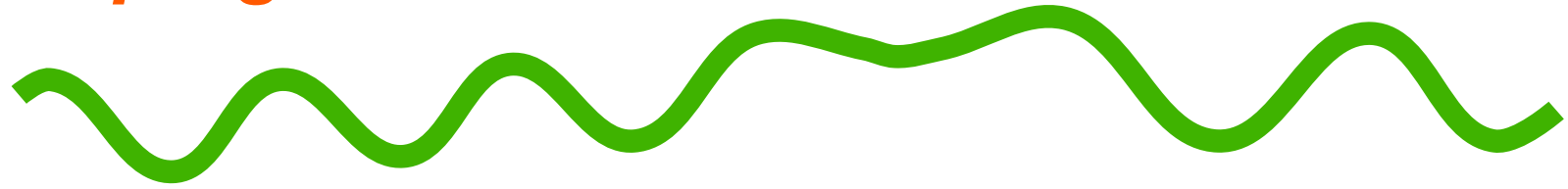


- Message Passing Interface Forum
<http://www.mpi-forum.org/>
- MPI: A Message-Passing Interface Standard
<http://www.mcs.anl.gov/mpi/>
- MPI - Message Passing Interface
<http://www.erc.msstate.edu/mpi/>
- MPICH Home Page
<http://www.mcs.anl.gov/mpi/mpich/>
- LAM Overview
<http://www.lam-mpi.org>
- Manuel en ligne
par exemple `man MPI_Send`



Compilation et exécution de programmes MPI

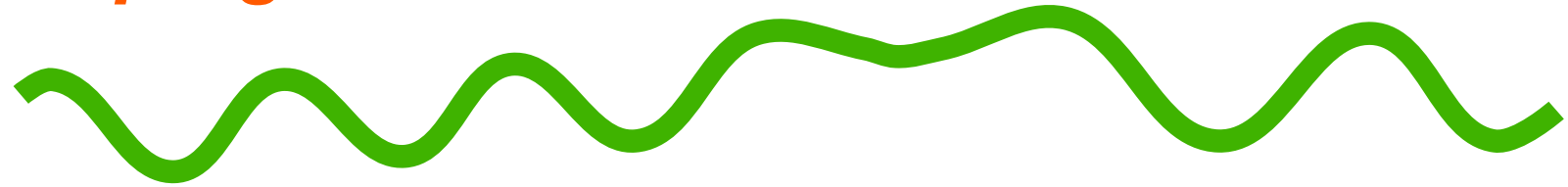
Compilation et exécution de programmes MPI



~ Variable selon l'environnement et l'implantation

- ~ MPICH
- ~ CHIMP
- ~ LAM
- ~ IBM SP-3

Compilation et exécution de programmes MPI sur IBM (1)



✓ Compilateur C/C++/Fortran

- ✓ scripts `mpcc/mpCC/mpxlf`
- ✓ compilation + édition de liens avec la bibliothèque MPI
- ✓ compilation de programmes MPI C/C++/Fortran multithreadés : `mpcc_r`, `mpCC_r`, `mpxlf_r`
- ✓ options d'optimisation `-qarch=pwr3 -O3 -qstrict`

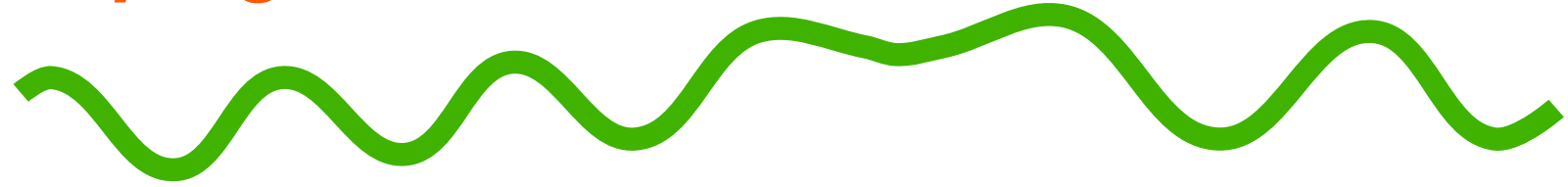
✓ Exécution d'un programme parallèle

- ✓ Exécutable accessible sur tous les nœuds de la machine (par exemple : sous le répertoire de login et pas dans `/tmp`)
- ✓ Accès par `rsh` à tous les nœuds (`$HOME/.rhosts`)

✓ Spécification des noeuds

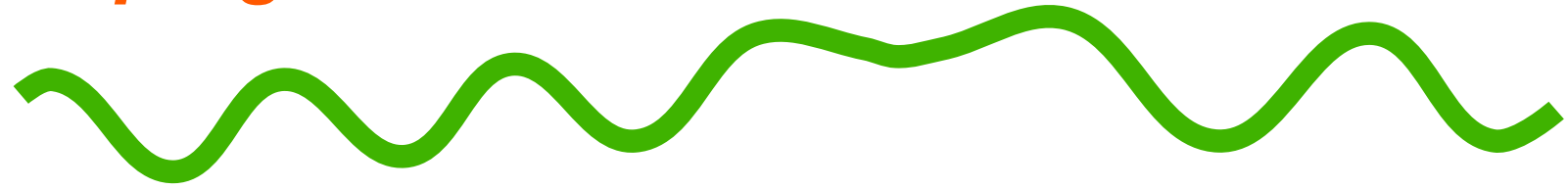
- ✓ fichier `host.list` du répertoire courant
- ✓ fichier désigné par la variable `MP_HOSTFILE`
- ✓ fichier contient une liste de noms de noeuds (un nom par ligne)

Compilation et exécution de programmes MPI sur IBM (2)



- ✓ Nombre de nœuds et tâches : 3 variables d'environnement
 - ✓ $MP_PROCS = MP_NODES \times MP_TASKS_PER_NODE$
 - ✓ option `-procs <n>` de l'exécutable
- ✓ Affichage sortie standard `stdout` et sortie d'erreur `stderr`
 - ✓ redirigé sur la console
 - ✓ variable `MP_LABEL_IO` `yes`
 - ✓ variable `MP_STDOUT_MODE` `ordered`
- ✓ Affichage de messages (de debug)
 - ✓ variable `MP_INFOLEVEL` (valeur de 1 à 6)
- ✓ Protocole pour `MPI_Send()`
 - ✓ copie si taille message inférieure à `MP_EAGER_LIMIT` octets
 - ✓ rendez-vous si taille message supérieure à `MP_EAGER_LIMIT` octets

Compilation et exécution de programmes MPI sur IBM (3)



~ Attente d'un message

- ~ variable `MP_WAIT_MODE`
- ~ attente active : valeur `pool`
- ~ passe la main : valeur `nopool`

~ Échange de messages entre deux processus sur un même nœud

- ~ par mémoire partagée ou non
- ~ `MP_SHARED_MEMORY` valeur `yes` ou `no`

~ Utilisation du réseau Colony

- ~ variable `MP_EUILIB`
- ~ IP : valeur `ip`
- ~ Colony : valeur `us` (*user space*)