

MPI & Calculs parallèles

François Willot
(département Mathématiques & Systèmes, ENSMP)
francois.willot@ensmp.fr

ESIEE, 14 octobre 2012

Plan

Plan

0. Introduction

1. MPI et le calcul distribué

2. Un programme minimal, compilation et exécution

3. Communication point à point

4. Communications collectives

5. Communications non-blocantes

6. Fonctions avancées MPI-2

7. Évaluation des performances, débogage

8. Applications : décomposition des données

9. Conclusions

NB : utilisation de la terminologie anglaise *en italique* en supplément du français lorsqu'il n'y a pas d'équivalent français ou que le nom français est inusité

Introduction

Principe du calcul parallèle

- Faire travailler simultanément un nombre important de machines dotées de multi-processeurs
- Modélisation physique 3D (type champs de phase) depuis les années 90

But

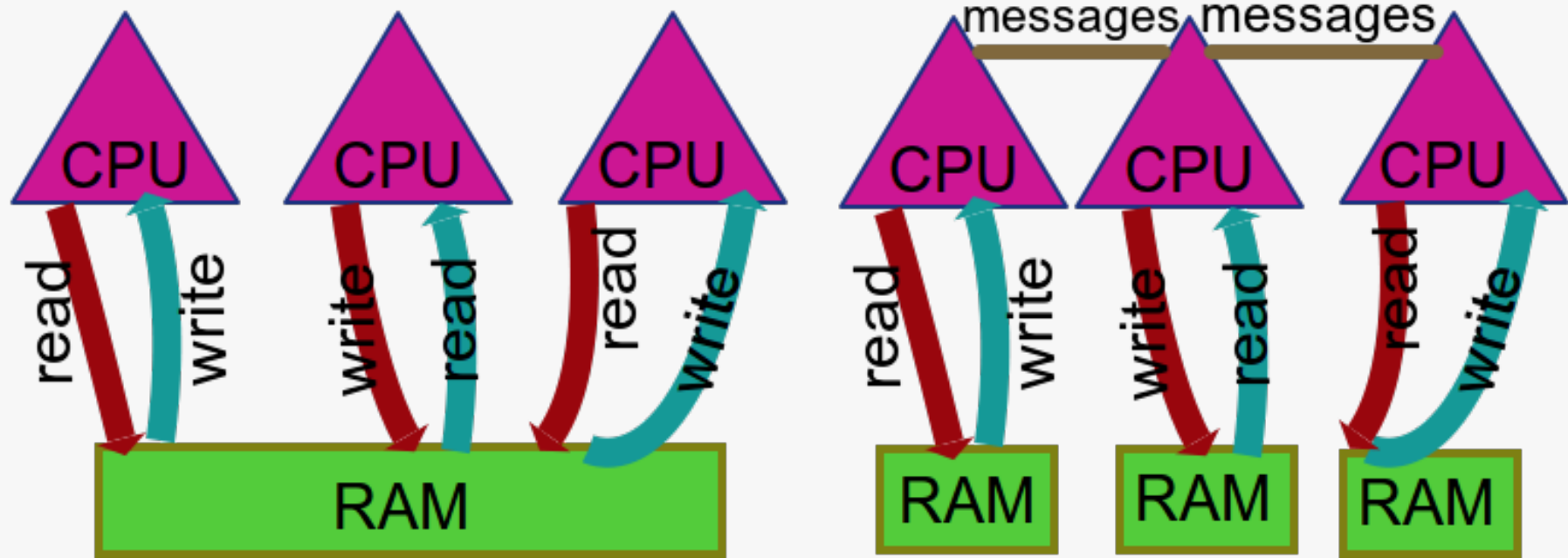
- Effectuer plus rapidement des calculs numériques
- Effectuer des calculs sur des volumes de données inatteignables sur des machines classiques

La programmation en parallèle nécessite :

- Des interconnexions rapides entre machines
- Un protocole de connexion
- La séparation de l'algorithme de calcul et des données en tâches parallèles

MPI et le calcul distribu  

Parallélisation en mémoire partagée et distribuée



Mémoire partagée (SMP) par
ex. programmation OpenMP
Nombre de processeurs
limité (rarement > 32)

Mémoire distribuée (type MPI)
Nombre de processeurs
théorique « sans limite »

NB : rien n'empêche d'utiliser les deux, par exemple sur des
clusters de multi-processeurs

Le modèle MPI

Un processus (*process*) est traditionnellement constitué d'un pointeur d'instruction (*program counter*) et d'un espace d'adresses (*address space*)

Un processus peut avoir plusieurs fils d'exécution (*threads*) avec leur propre pointeur d'instruction et pile (*stack*) mais partageant le même espace d'adresses.

MPI sert aux communication entre processus, dont les espaces d'adresses sont séparés

Deux types de communications entre processus :

- Synchronisation
- Transfert de données entre les espaces d'adresses de deux processus

Qu'est-ce que MPI

MPI pour *Message Passing Interface* : standard de librairie logicielle de passage de message

- standard, i.e. spécifications formelles
- pas un langage de programmation, ni des spécifications de compilation
- passage de message étendue à tout type de données
- pas une implémentation spécifique ni un produit

Librairie pour ordinateurs en parallèle, grappes de serveurs (*clusters*) et réseaux hétérogènes

Qu'est-ce que MPI

Système complet pour le développement de programmes en parallèle, avec fonctionnalités de « topologie virtuelle »

MPI s'adresse aux développeurs de librairie, aux programmeurs et utilisateurs

Portabilité : nombreuses implémentations MPI pour tout type de système d'exploitation et de langages : C, C++, Fortran 77/90, python, java et potentiellement beaucoup d'autres

MPI : liens et sources

Le standard en lui-même

<http://www.mpi-forum.org> (toutes les versions officielles)

Livres de référence

MPI: The Complete Reference, Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996

Designing and Building Parallel Programs, Ian Foster, Addison-Wesley, 1995

Parallel Programming with MPI, Peter Pacheco, Morgan-Kaufmann, 1997

MPI: The Complete Reference Vol 1 and 2, MIT Press, 1998

Sources en ligne

<http://wotug.org/paralle/environments/mpi/> (nombreux exemples)

<http://www.mcs.anl.gov/mpi> (toutes sortes de liens)

Pourquoi utiliser MPI ?

MPI est une façon flexible, efficace et portable de programmer un algorithme parallèle

MPI permet expressément de générer des bibliothèques...

... un utilisateur final n'a donc pas besoin de connaître beaucoup de choses sur MPI pour l'utiliser

MPI : un bref historique

MPI a été conçu par un consortium d'universités et d'entreprises, la standardisation a débuté en 1992

Environ 40 organisations et 60 représentants aux États-Unis et en Europe

Après de multiples réunions, MPI 1.1 en juillet 1995

MPI 2.0 en juillet 1997

- Créations et gestion de processus
- *One-sided communication*
- Opérations collectives étendues
- Lecture/écriture en parallèle (*parallel I/O*)

Implémentations

Trois implémentations libres souvent utilisées

- MPICH & MPICH2 (Argonne National Lab, Chicago)
- LAM/MPI (Indiana University)
- OpenMPI (issu de LAM/MPI)

OpenMPI équipe le « K Computer », Kobe, Japon, plus gros super ordinateur au monde (65000 CPUs, 8×10^{24} opérations à virgule flottante par seconde)

Environnement

Note : il existe des distributions Linux spécialement conçues pour les cluster, par exemple *Rocks Cluster Distribution*

- Réinstallation du système sur chaque nœud à partir du frontal
- Gestion des jobs
- Connectiques standards peu chères (infiniBand).

Pour les gros clusters, réseau très optimisé (tore de dimension grande). Consommation électrique importante, risque de défaut matériel important.

MPI : un programme minimal

Un programme MPI minimal (C)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Exécuter MPI

En C :

- mpi.h doit être inclus
- les fonctions **MPI_*** retournent un code d'erreur ou **MPI_SUCCESS**

Le standard MPI-1 ne définit pas comment on exécute un programme MPI, de la même façon que Fortran ne dit rien sur l'exécution d'un programme Fortran

En général, le lancement d'un programme dépend de l'implémentation de MPI que vous utilisez, et nécessite divers scripts, options de compilation et variables d'environnement

Exécuter MPI

À titre d'exemple, dans un environnement type Linux avec l'une des configurations les plus simples, on peut exécuter les commandes suivantes :

```
$ mpicc -o hello hello.c
```

```
$ mpirun -np <number of procs> -H <host1,host2,...>  
./hello
```

NB : mpirun utilisera typiquement ssh pour lancer le programme sur chaque noeud (les communications elles ne passent pas par SSH)

(en supposant que divers variables d'environnement type LD_LIBRARY_PATH n'ont pas à être initialisées, qu'il n'y a pas de programme de lancement de jobs etc.)

Obtenir des informations sur l'environnement local

Deux questions qui se posent régulièrement avec MPI :

- Combien de processus participent au calcul ?
- Quel processus suis-je ?

MPI fournit les fonctions suivantes pour répondre à ces questions :

- **MPI_Comm_size** retourne le nombre de processus
- **MPI_Comm_rank** retourne le *rang*, nombre compris entre 0 et *size-1* identifiant le processus

Un meilleur *Hello World* (C)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Communications point à point

Opérations de communication coopératives

L'approche par passage de message rend la communication *coopérative*

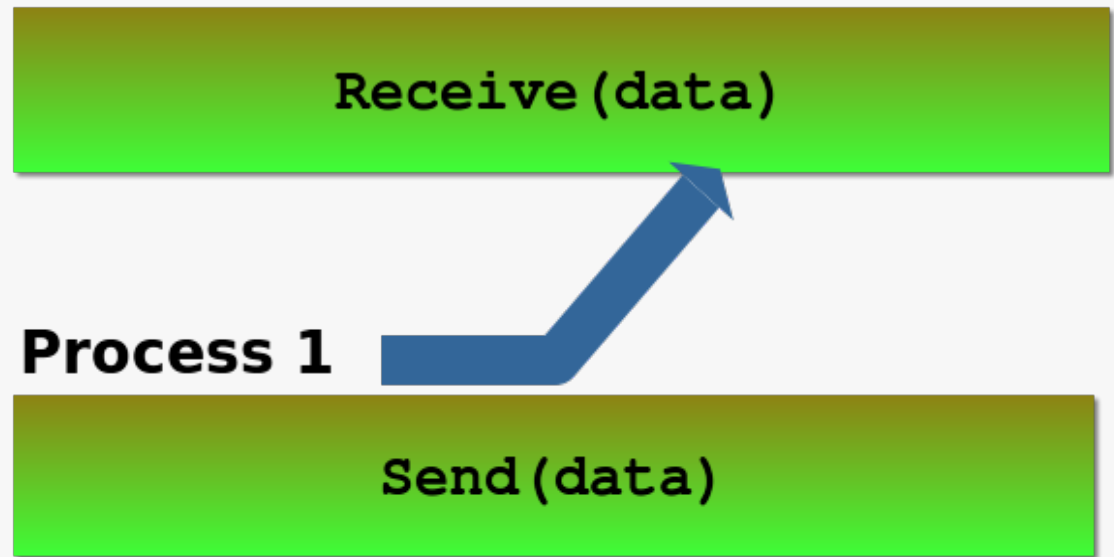
Les données sont explicitement **envoyées** par un processus et **reçues** par un autre

Process 0

Receive (data)

Process 1

Send (data)



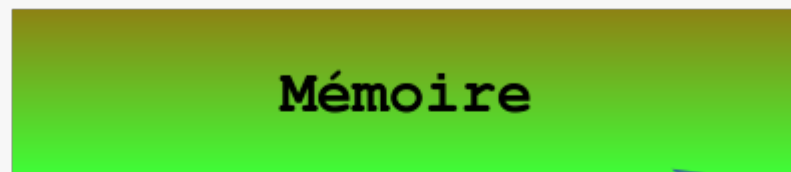
- Ainsi tout changement dans la mémoire d'un des processus se fait avec la participation explicite de celui-ci
- Communication et synchronisation sont combinés

Communications non coopératives

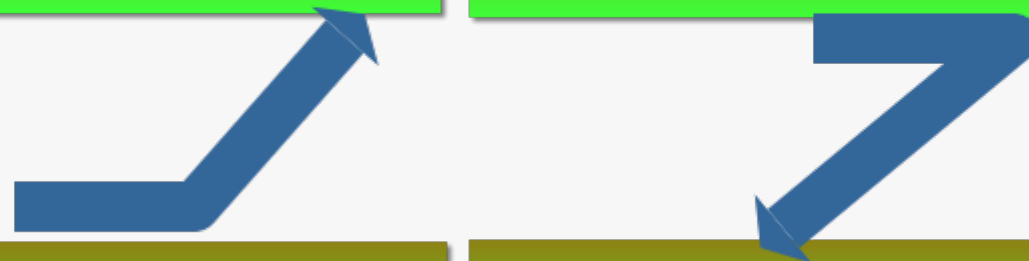
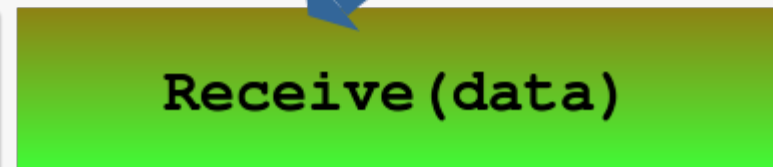
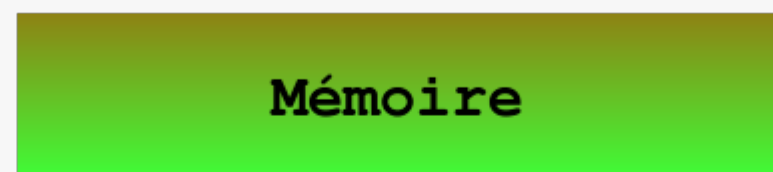
Communications non coopératives (*one-sided operations*) : lecture et écriture de la mémoire à distance

- Seulement un processus participe explicitement au transfert
- Avantage : communication et synchronisation découplées
- Standard MPI-2 seulement

Process 0

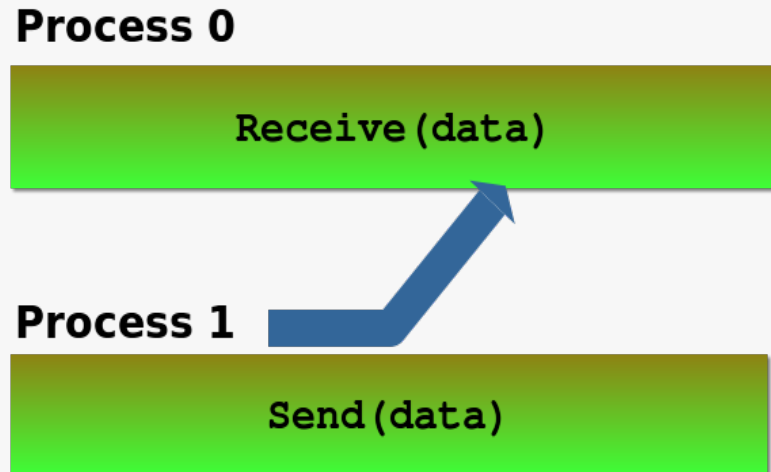


Process 1



Opération d'envoi/réception basique

Quels sont les détails de :

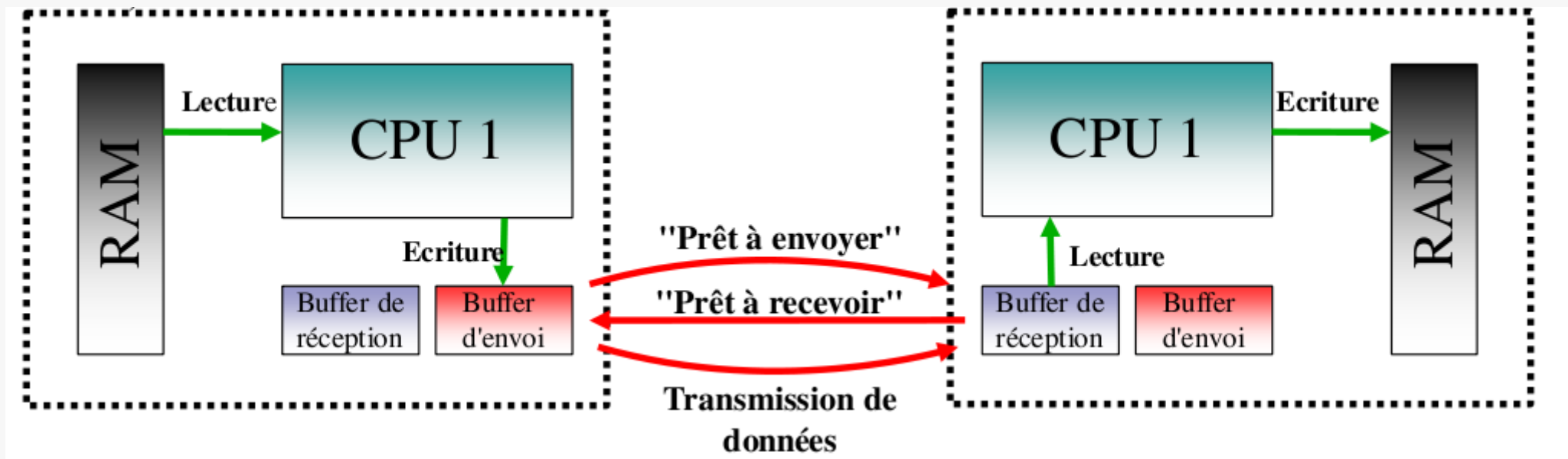


Il faut notamment préciser :

- Comment décrire les données
- Comment identifier les processus ?
- Comment le receveur passe en revue et identifie les messages
- Ce qui se passe lorsque l'opération se termine

Opérations de communication point à point

Que se passe-t-il dans la réalité ?



L'envoi et la réception donnent lieu à une *transaction*, ici les deux processeurs interviennent et gèrent la communication

La transmission est *synchrone*. En mode *asynchrone*, le processeur (0) n'attend pas que (1) ait signalé qu'il est prêt

Quelques concepts de base

- Les processus peuvent être regroupés en groupes (*groups*)
- Chaque message est envoyé dans le cadre d'un contexte (*context*), et doit être reçu dans le même contexte
- Un groupe et un contexte forment un *communicator*
- Un processus est identifié par son *rang* dans le groupe associé à un *communicator*
- Il existe un *communicator* par défaut dont le groupe contient tous les processus initiaux, appelé **MPI_COMM_WORLD**.

Les types de données

- Les données envoyées dans un message sont décrites par un triplet adresse, taille (*count*), type.
- Un type de données MPI est défini récursivement par :
 - 1) Un type prédéfini (e.g. **MPI_INT**, **MPI_DOUBLE_PRECISION**)
 - 2) Un tableau contigu (*array*) de 1)
 - 3) Une structure ou un bloque de données (*strided block*)
 - 4) Un tableau de 3)
 - 5) Une structure de données
- Il existe des fonctions MPI pour construire des types personnalisés, comme un tableau de paires (*int, double*)

Les *tags* MPI

- Les *tags* sont utilisés pour que le receveur identifie chaque message qui lui est envoyé. Le *tag* est un entier.
- Concrètement, les messages sont passés en revue jusqu'à ce que le *tag* ou l'un des *tags* attendus soient détectés, ou encore non filtrés si **MPI_ANY_TAG** est spécifié comme *tag*

Envoi de message basique (bloquant)

`MPI_SEND(start, count, datatype, dest, tag, comm)`

- La mémoire tampon du message (*message buffer*) est défini par (`start`, `count`, `datatype`)
- Le processus receveur est spécifié par `dest`, qui est le rang du processus dans le communicator donné par `comm`
- Lorsque cette fonction retourne une valeur, les données ont été transmises et la mémoire tampon peut-être utilisée à nouveau. Le message n'a pas nécessairement été envoyé à destination.

Réception de message basique (bloquant)

**MPI_RECV(start, count, datatype,
source, tag, comm, status)**

- Le processus attends tant qu'un message correspondant aux **tag** et **source** spécifiés apparaît et que la mémoire tampon est disponible
- **source** est le rang d'un processus dans le communicator **comm** ou **MPI_ANY_SOURCE**
- **status** contient d'autres informations
- on peut recevoir moins de données que ce qui est spécifié par **count**, mais en recevoir plus est une erreur

Recevoir plus d'informations sur un message

– **status** est une structure de données contenant des informations supplémentaires ou redondantes, alloué par l'utilisateur

– En C :

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...,
&status )
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype,
&recvd_count );
```


MPI est simple

Dans beaucoup de code parallèles on peut se contenter de 6 fonctions dont 2 seulement sont non-triviales :

MPI_INIT

MPI_FINALIZE

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_SEND

MPI_RECV

Cependant les communications de point à point ne sont pas la meilleure solution à tout...

Exemple MPI_SEND/MPI_RECV

```
#include "mpi.h"

main(int argc, char **argv ) {
    char message[20];
    int myrank;

    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) {    /* code for process zero */
        strcpy(message, "Hello, there");
        MPI_Send(message, strlen(message), MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
    } else {    /* code for process 1 */
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

Communications collectives

Opérations collectives

Les opérations collectives sont appelées par tous les processus dans un *communicator*.

- **MPI_BCAST** distribue les données d'un processeur maître à tous les autres dans un *communicator*
- **MPI_REDUCE** combine les données de tous les processus dans un *communicator* et les envoie vers un processus
- Dans beaucoup d'algorithmes, **SEND/RECV** peuvent-être remplacés par **BCAST/REDUCE**, améliorant les performances et la simplicité du code

Envoi d'un message à tous BCAST

**MPI_BCAST(start, count,
datatype, root, comm)**

- À la fin de l'opération, les données stockées dans **start** par le processus **root** sont recopiés dans la mémoire de tous les processus de **comm**
- MPI s'occupe lui-même de l'identification du message, il n'y a plus de **tag**

Envoi d'un message REDUCE

`MPI_REDUCE(send_start, recv_start, count, datatype, op, root, comm)`

- À la fin de l'opération, les données stockées dans `send_start` par tous les processus de `comm` sont réduits par l'opération `op` et le résultat stocké dans `recv_start` du processus `root`
- Les données d'entrée et de sortie ont la même taille `count`, le même type `datatype`, et l'opération `op` est effectuée sur chaque élément
- Opérations `op` prédéfinies : `MPI_SUM`, `MPI_PROD`, `MPI_MAX/MPI_MIN`, `MPI_MAXLOC/MPI_MINLOC` (max/min et son argument) ou définies par l'utilisateur (`MPI_OP_CREATE`, doit être associatif)

Exemple MPI_REDUCE

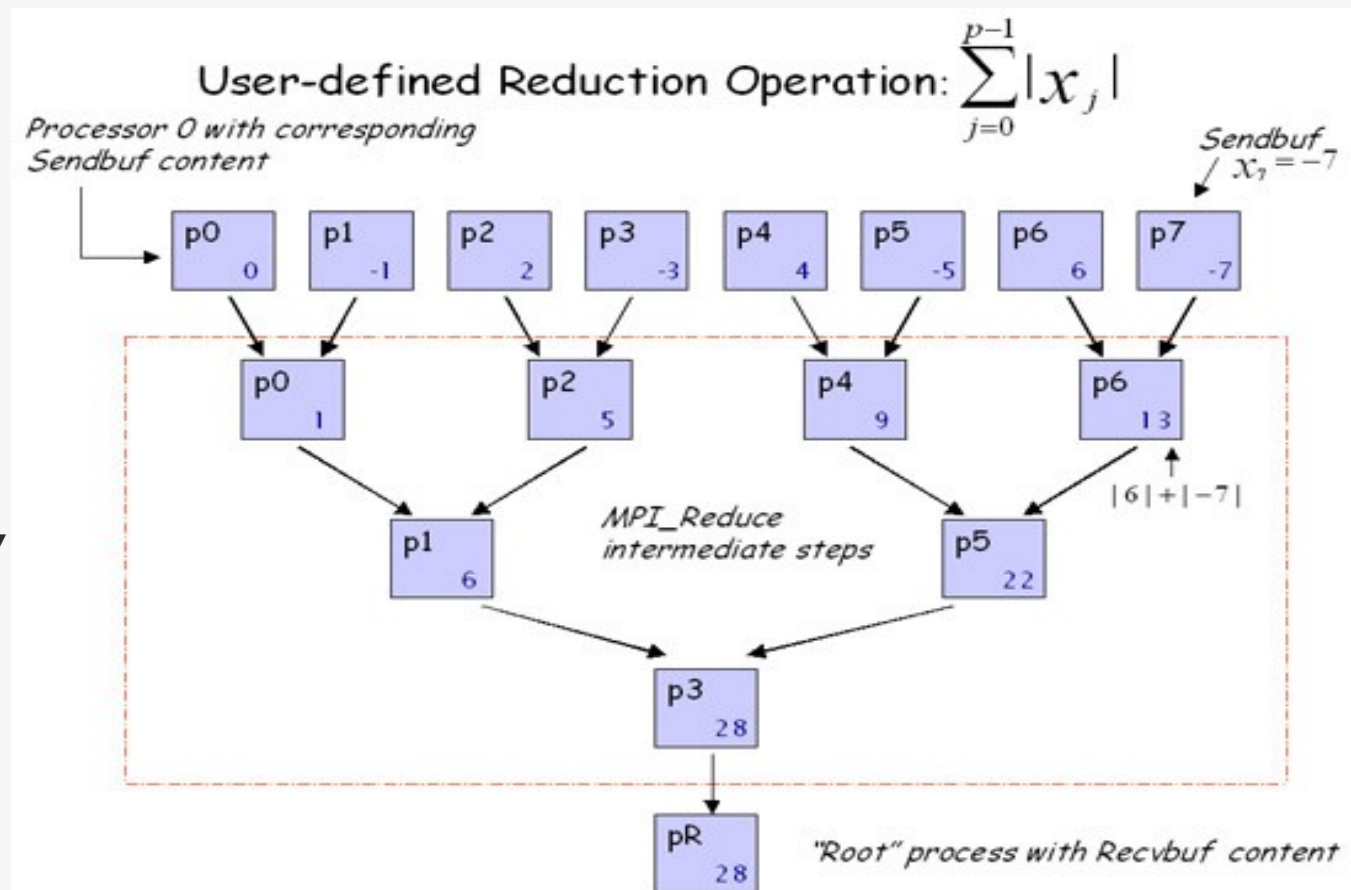
```
#include "mpi.h"

main(int argc, char **argv ) {
    int myrank, max_rank;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank ) ;
    MPI_REDUCE(&myrank, &max_rank, 1, MPI_INT,
MPI_MAX, 0, MPI_COMM_WORLD)
    if (myrank == 0) {
        printf("Max rank is :%u:\n", max_rank);
    }
    MPI_Finalize();
}
```

Example MPI_REDUCE

```
void onenorm(float *in, float *inout, int *len, MPI_Datatype
 *type) {
    int i;
    for (i=0; i<*len; i++) {
        *inout = fabs(*in) + fabs(*inout);
        in++;
        inout++;
    }
}
```

```
MPI_Op_create(onenorm,
commute, &myop);
MPI_Reduce(&send, &recv, 1,
MPI_FLOAT, myop, root,
MPI_COMM_WORLD)
```



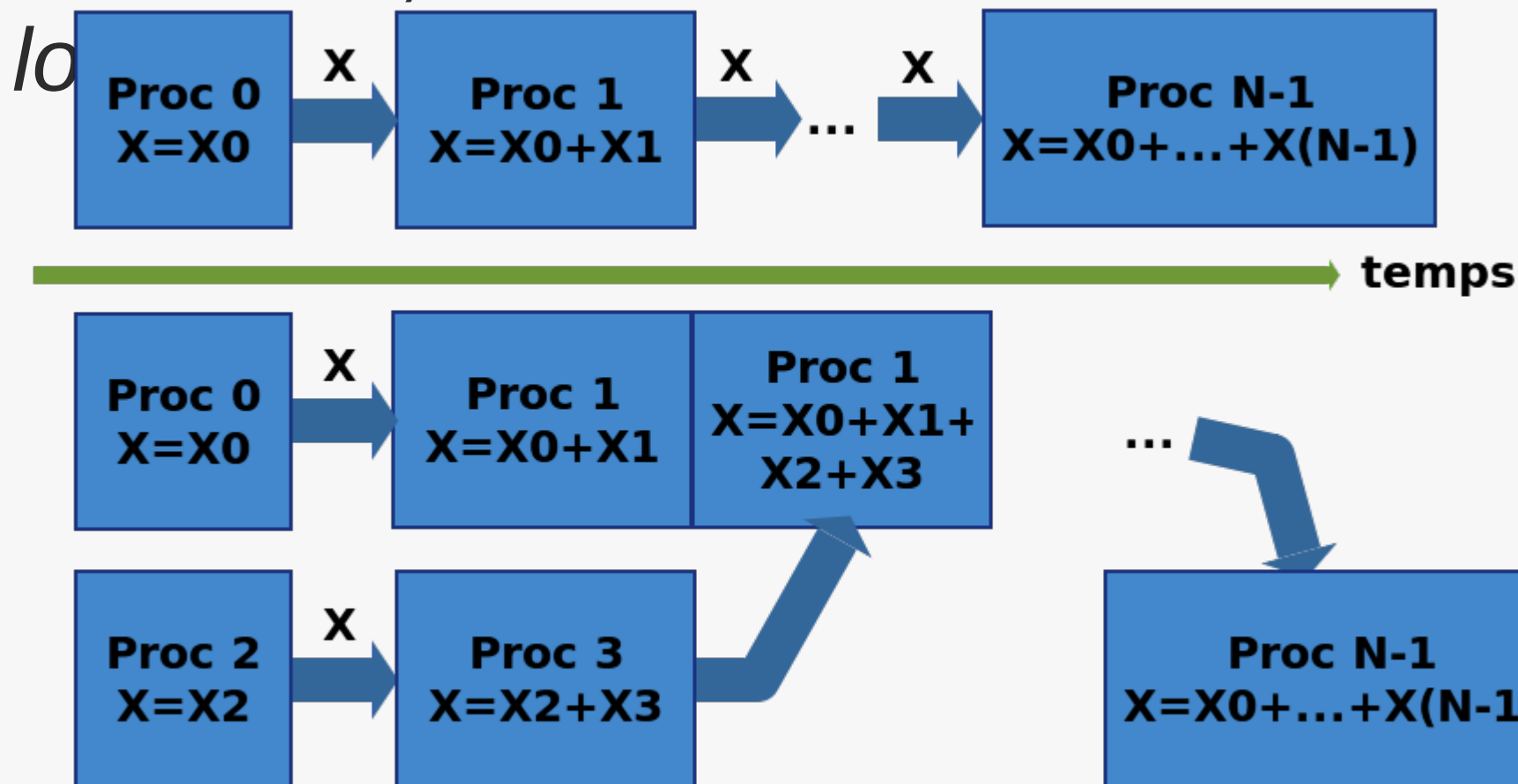
Opérations collectives

Les opérations collectives sont optimisées autant que possible par MPI.

- Il n'existe le plus souvent pas de lien dédié entre chaque couple de processeurs (trop cher)
- Plus exactement, l'optimisation dépend de la topologie du réseau (boucle, arbre hiérarchique, tore, grille)
- Si deux processus résident sur la même machine physique (par exemple pour bénéficier de nœuds multi-processeurs), une bonne implémentation MPI le détectera et utilisera des optimisations en interne

Opérations collectives

Optimisation typique pour une opération associative et commutative : méthode naïve $\sim N$, méthode de réduction $\sim N \log N$



Autres opérations collectives

`MPI_GATHER(send_start, send_count, send_type,
recv_start, recv_count, recv_type, root, comm)`

– À la fin de l'opération, les données stockées dans `send_start` par tous les processus de `comm` sont concaténées et reçues par `root`

`MPI_SCATTER(send_start, send_count, send_type,
recv_start, recv_count, recv_type, root, comm)`

– Opération inverse : les données stockées dans `recv_start` par `root` sont envoyées aux processus de `comm` et découpées en parts égales

`MPI_ALLGATHER(send_start, send_count, send_type,
recv_start, recv_count, recv_type, comm)`

– Identique à `MPI_GATHER` sauf que les données sont reçues par tous les processus et non seulement par `root`

Autres opérations collectives

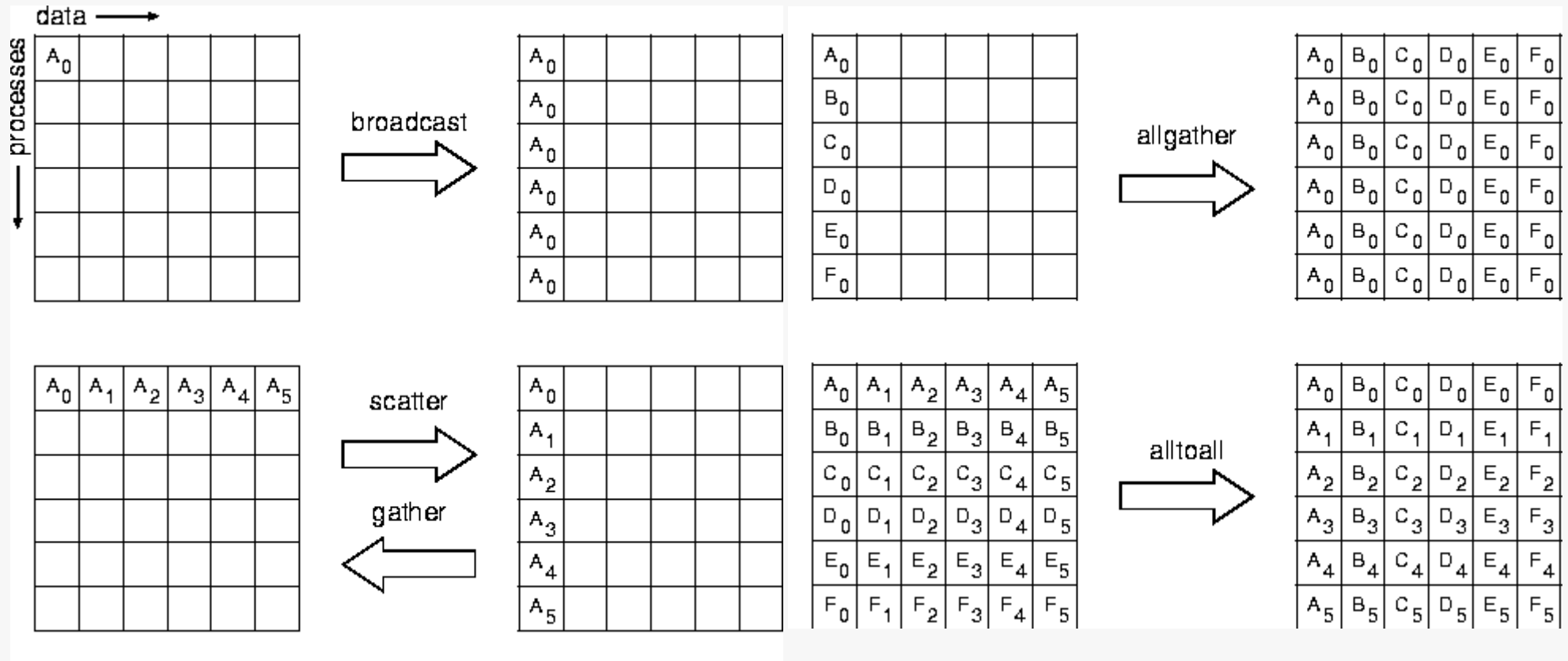
`MPI_ALLTOALL(send_start, send_count, send_type,
recv_start, recv_count, recv_type, comm)`

- Effectue une transposition des données : le bloc en position j du processus i est reçu en position i par le processus j

`MPI_GATHERV(...), MPI_SCATTERV(...),
MPI_ALLGATHERV(...), MPI_ALLTOALLV(...),
MPI_ALLTOALLW(...)`

- Données de taille variable ou dans un ordre distinct de celui du rang des processeurs

Autres opérations collectives



Représentation matricielle (processus/mémoire) des opérations collectives hors-réduction

Autres opérations de réductions collectives

MPI_ALLREDUCE(send_start, recv_start, count, datatype, op, comm)

- Identique à **MPI_REDUCE** sauf que le résultat est stockée par tous les processus de **comm**

MPI_SCAN(send_start, recv_start, count, datatype, op, comm)

- Identique à **MPI_ALLREDUCE** sauf que le résultat de l'opération jusqu'au processus *i* (inclus) est stockée dans *i*

**Communications
non bloquantes,
synchronisation**

Modes de communications optimisés

Avec **MPI_SEND**, MPI choisit entre le mode buffer et synchrone :

- **buffer** : la fonction complète dès que le message est stocké dans le buffer d'envoi
- **synchrone** : la fonction complète lorsque le message « prêt » est reçu du destinataire

On peut expliciter ce choix avec **MPI_BSEND**,
MPI_SSEND

Un troisième mode est possible :

- **ready** (**MPI_RSEND**) : le récepteur suppose que le destinataire est prêt à recevoir et envoie les données directement (meilleures performances)

Modes de communications optimisés

Toutes ces fonctions existent de plus en variante non-bloquante : la fonction complète avant que les ressources aient été libérés. Des calculs peuvent être effectués immédiatement après mais il faut s'assurer que les données ne sont pas modifiées avant que l'envoi soit terminé.

En résumé :

	Bloquant	Non-bloquant
Standard	MPI_SEND	MPI_ISEND
Synchrone	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Communications non-bloquantes

MPI_ISEND(start, count, datatype, dest, tag, comm, request)

Pour les communications non-bloquantes, le programme garde trace de l'envoi par un nouvel argument, *request*, de type **MPI_Request**.

Le statut de l'envoi peut-être inspecté par **MPI_REQUEST_GET_STATUS(request, status)** ou **MPI_WAIT(resquest, status)**

Exemple MPI_Isend

```
int myid, numprocs, left, right, b[10], c[10];
MPI_Request r, r2;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
right=(myid+1)%numprocs; left=(myid-1)%numprocs;
MPI_Irecv(b,10,MPI_INT,left,123,MPI_COMM_WORLD,&r) ;
MPI_Isend(c,10,MPI_INT,right,123,MPI_COMM_WORLD,&r2);
MPI_Wait(&r, &status);
MPI_Wait(&r2, &status);
MPI_Finalize();
```

Synchronisation

MPI_Barrier(comm)

Bloque l'ensemble des processus de **comm** jusqu'à ce que le dernier soit arrivé à la barrière

Pour synchroniser deux processus, on peut utiliser **MPI_SEND/MPI_RECV**

Les types personnalisés

```
int MPI_Vector3
```

```
MPI_TYPE_CONTIGUOUS(3, MPI_REAL, MPI_Vector3)
```

```
MPI_TYPE_COMMIT(MPI_Vector3)
```

```
...
```

```
MPI_TYPE_FREE(MPI_Vector3)
```

Définit un type vecteur de taille 3, stocké consécutivement

```
MPI_TYPE_VECTOR(...)
```

Données stockées non consécutivement mais avec pas constant

```
MPI_TYPE_STRUCT(...)
```

Structure (préciser l'ensemble des données, taille et pas)

Fonctions avancés (MPI-2)

Communications *one-sided*

Que faire lorsqu'un processus ne sait pas qu'il doit recevoir ou envoyer des données ?

`MPI_IPROBE(...)`

Inspecte l'état d'un envoi : synchronisation délicate

`MPI_WIN_CREATE(start, count, disp_unit, info, comm, win)`

`MPI_WIN_FREE(win)`

Définit une zone mémoire accessible par les autres processus

`MPI_PUT(loc_add, loc_count, loc_type, remote_rank, remote_disp,
remote_count, remote_type, win)`

`MPI_GET(loc_add, loc_count, loc_type, remote_rank, remote_disp,
remote_count, remote_type, win)`

`MPI_ACCUMULATE(...)`

Lecture/écriture sur un processus tiers. NB : opérations non-bloquantes.

`MPI_WIN_FENCE(...)`

Barrière

Lecture/écriture parallèle

Avec le développement du calcul parallèle, les opérations de lecture et écriture sur disque prennent des temps importants

Depuis MPI-2, des fonctions de lecture et écriture en parallèle ont été développées

Ces considérations vont au-delà du sujet de cette présentation et ne sont pas abordées ici

De même pour d'autres sujets pointus : gérer les défauts de processus et la gestion dynamique des nœuds, inter-communications

Problème du blocage
Performances
Débogage

Problèmes de *deadlock*

L'interblocage (deadlock) est un problème courant en programmation parallèle :

- Espace mémoire insuffisant dans le processus destinataire
- Croisement entre messages :

Process 0

Process 1

```
MPI_SEND(..., tag_1, ...) MPI_SEND(..., tag_2, ...)
MPI_RECV(..., tag_2, ...) MPI_RECV(..., tag_1, ...)
```

Périlleux, car le non-blocage dépend de la disponibilité des buffers

Solutions au *deadlock*

Choisir plus soigneusement l'ordre des opérations

Process 0

Process 1

<code>MPI_SEND(..., tag_1, ...)</code>	<code>MPI_RECV(..., tag_1, ...)</code>
<code>MPI_RECV(..., tag_2, ...)</code>	<code>MPI_SEND(..., tag_2, ...)</code>

Opérations non-bloquantes :

Process 0

Process 1

<code>MPI_ISEND(..., tag_1, ...)</code>	<code>MPI_Irecv(..., tag_1, ...)</code>
<code>MPI_Irecv(..., tag_2, ...)</code>	<code>MPI_ISEND(..., tag_2, ...)</code>
<code>MPI_WAITALL</code>	<code>MPI_WAITALL</code>

Évaluation des performances

Temps de calcul du programme séquentiel :

$$t_s = t_s(n)$$

n = volume de données, p = nombre de processeurs

Temps de calcul du programme en parallèle (calcul + envoie/réception de données) :

$$t_p = t_{\text{comp}}(n, p) + t_{\text{comm}}(n, p)$$

$$t_{\text{comm}}(n, p) = t_{\text{startup}} + nt_{\text{data}}(p) \quad (\text{idealized})$$

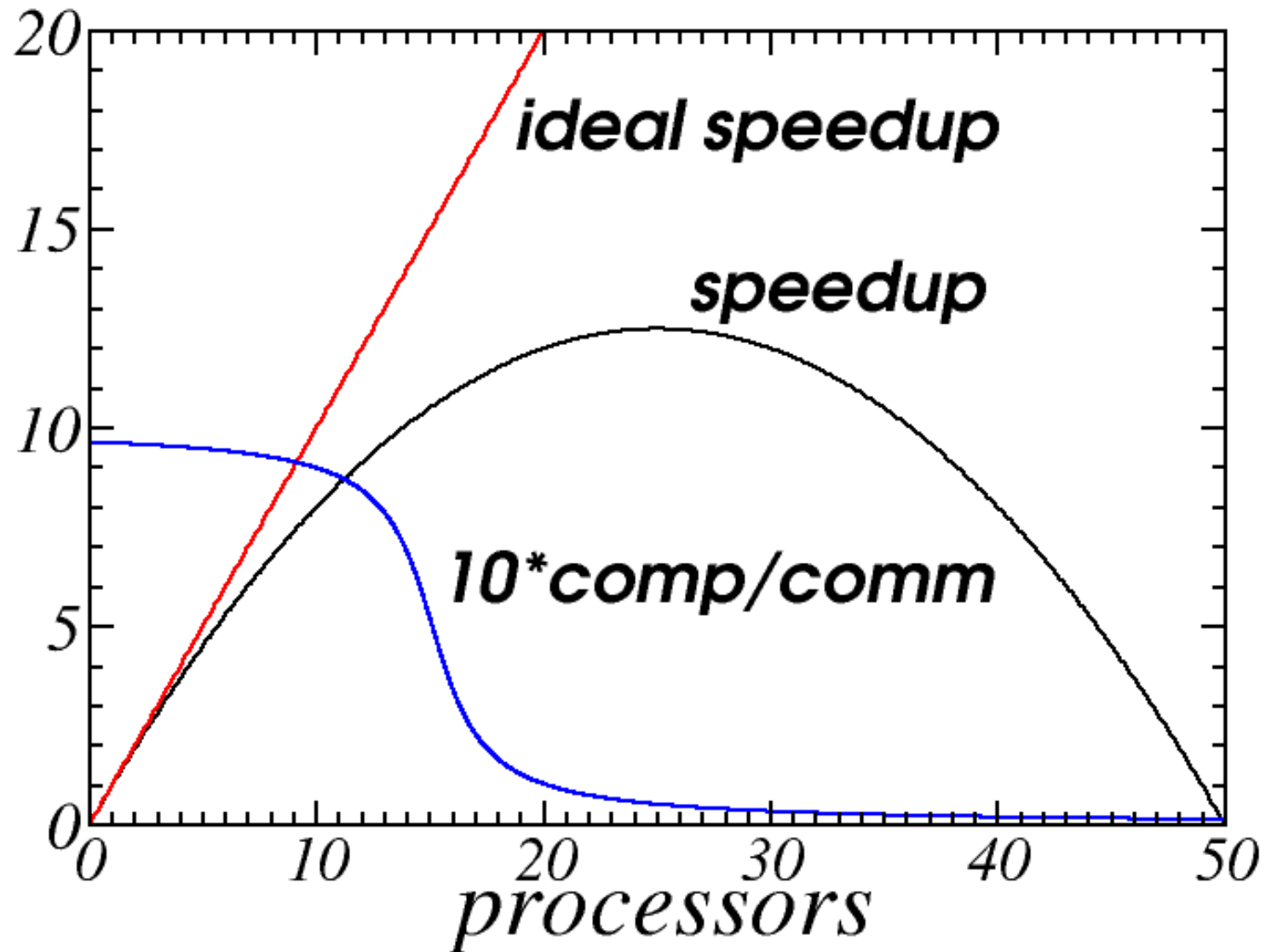
$$t_{\text{comp}}(n, p) \equiv f(p)n^m, \quad n \gg 1$$

Speedup factor et ratio calcul/communication

$$\frac{t_s}{t_p}, \quad \frac{t_{\text{comp}}}{t_{\text{comm}}}$$

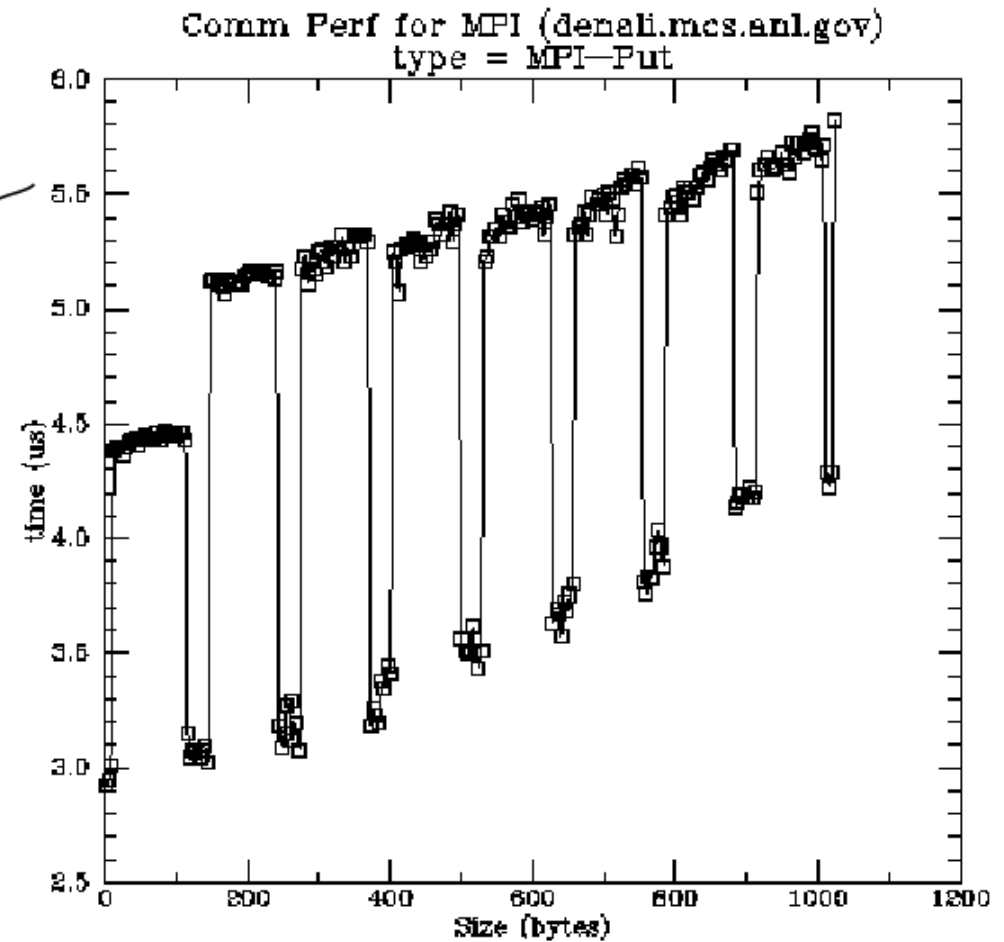
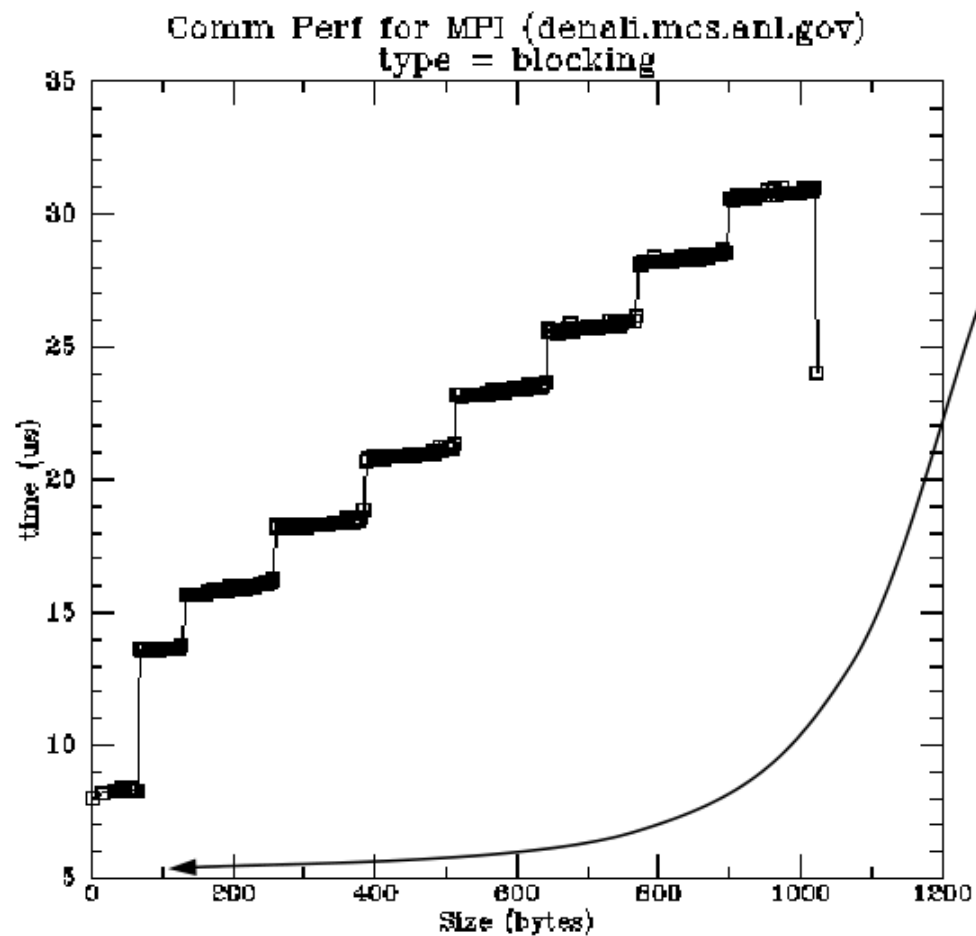
Évaluation des performances

Typiquement, les temps de communication deviennent prohibitifs pour $p > p_0$



Évaluation des performances

Un exemple réel



Évaluation des performances

```
tmptime = MPI_Wtime();  
<calcul...>  
comp_time = MPI_Wtime() - tmptime;  
if (rank == 0)  
    printf("Comp time for proc. 0 =  
%5.2f seconds", comp_time)
```

Débogage de programmes parallèles

Pour déjà essayer de les éviter :

Programmer en séquentiel, puis seulement en parallèle

Décrire l'algorithme parallèle en terme de structures globales

Dans un premier temps, programmer en parallèle sans chercher à optimiser

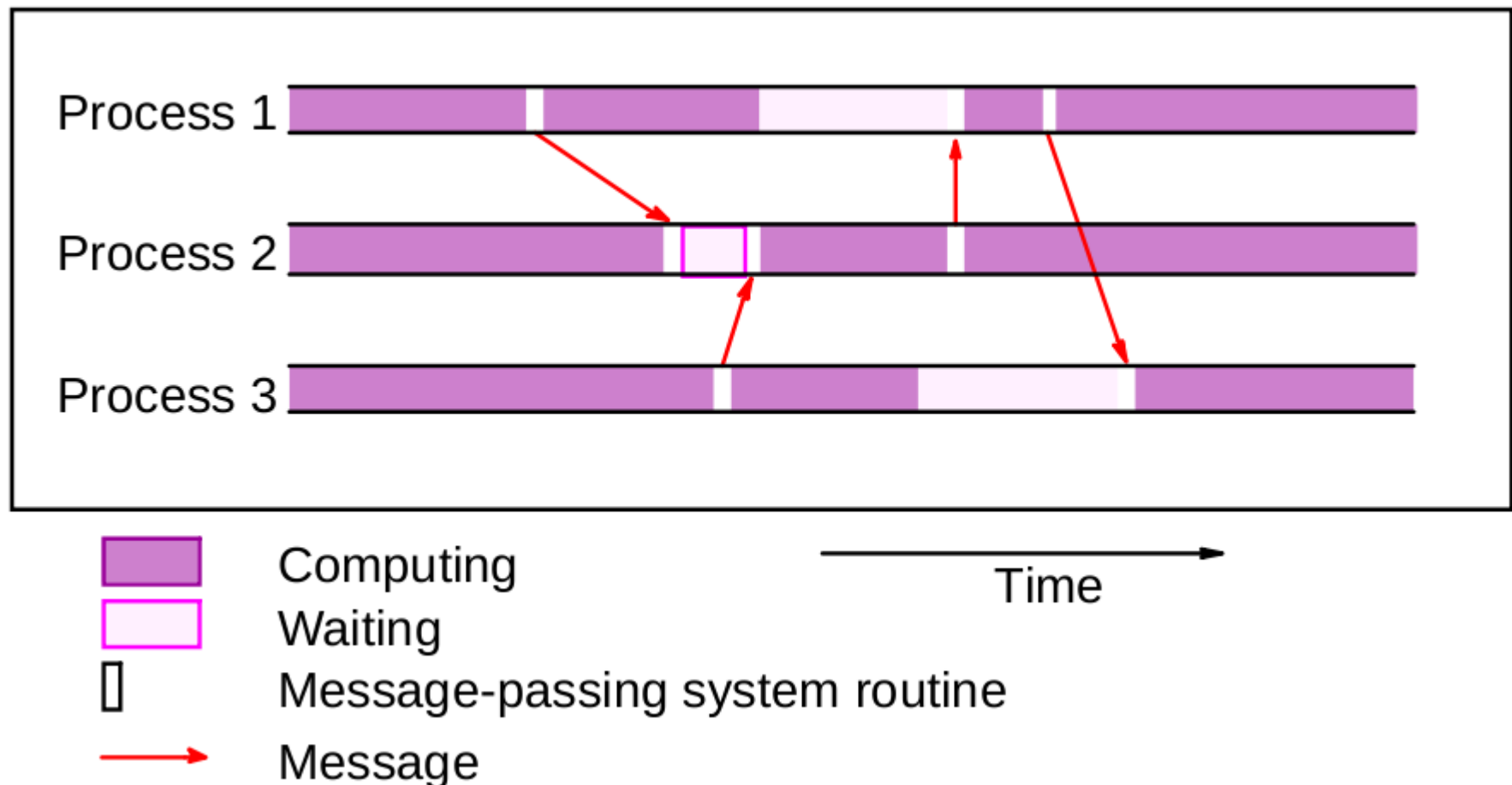
Maintenir le code séquentiel, afin de conserver l'algorithme « simple »

Débogage des programmes

Les sorties écrans n'apparaissent pas « en direct » : elles sont mises dans un buffer (cf. `fflush(stdout)` ;)

Programmes tiers pour déboguer type *totalView* : très utile mais nécessite un investissement

Diagramme espace-temps pour visualiser les temps d'attente et de calcul



Quelques applications

Parallélisation des algorithmes

Quelques exemples

Problème à 3 corps : **parallélisation impossible**
(volume de données fini restreint)

Problèmes à N corps, EDPs sur grille adaptative et/ou temps adaptatif, dynamique des dislocations :
parallélisation **délicate** (volume de données à mettre à jour)

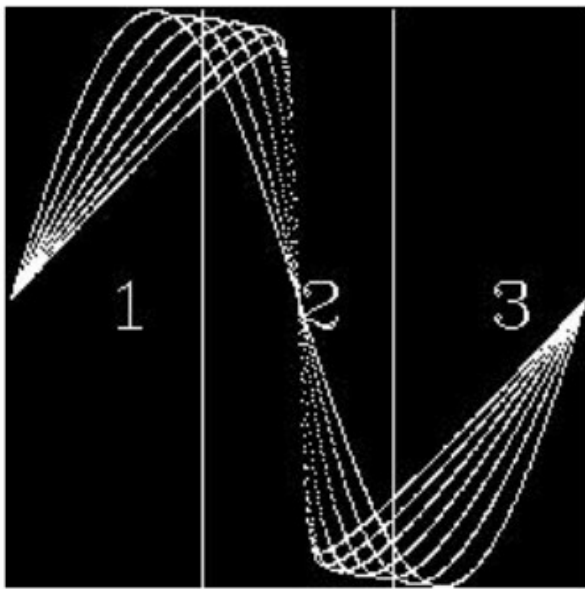
EDP sur grille régulière, simulations de Monte-Carlo
(en général) : **parallélisation simple**

Parallélisation des algorithmes

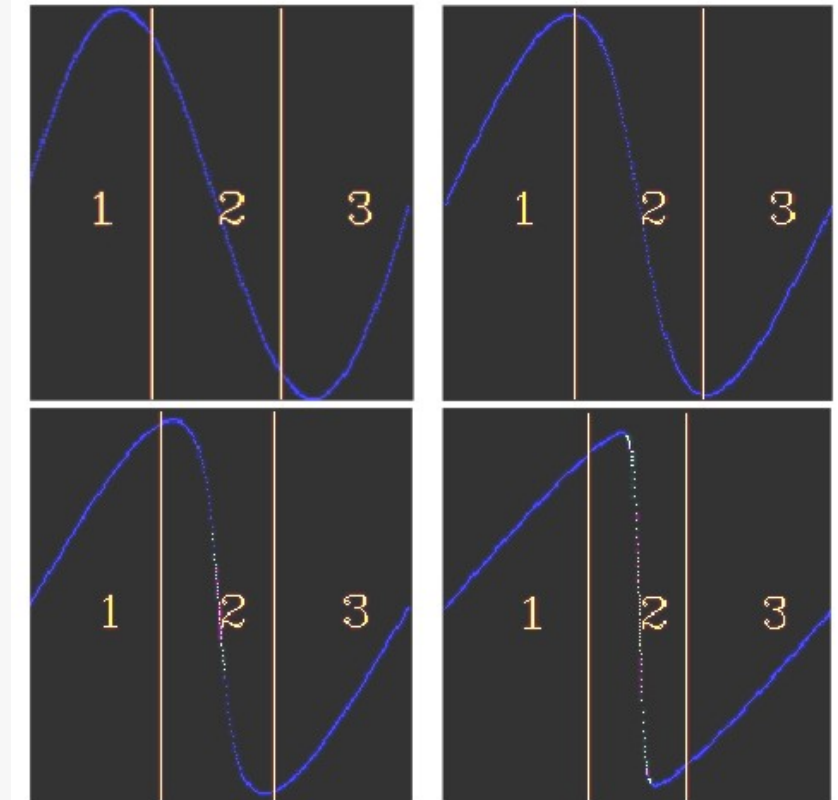
Schéma explicite en temps et grille régulière ou à maille adaptative

$$\frac{\partial \nu}{\partial t} = \mathfrak{F} \left\{ \nu, \frac{\partial \nu}{\partial x_i}, \frac{\partial^2 \nu}{\partial x_i^2}, \dots \right\}$$

Exemple : équation de Burger

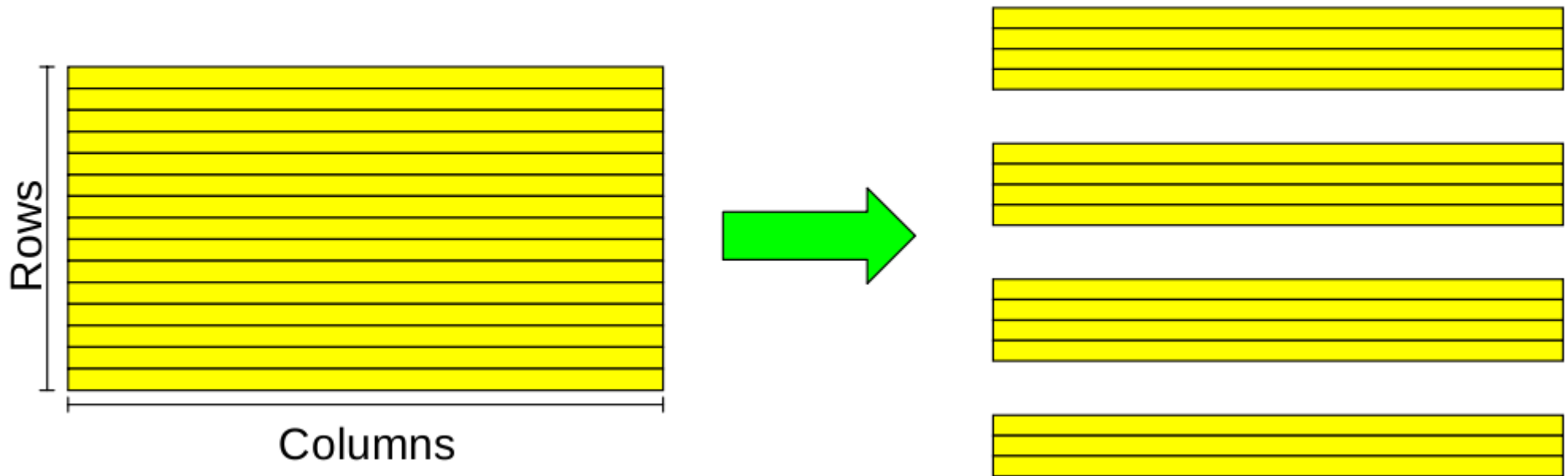


Domaines
constant ou
nombre de
point de
discrétisation
constant



Décomposition des données

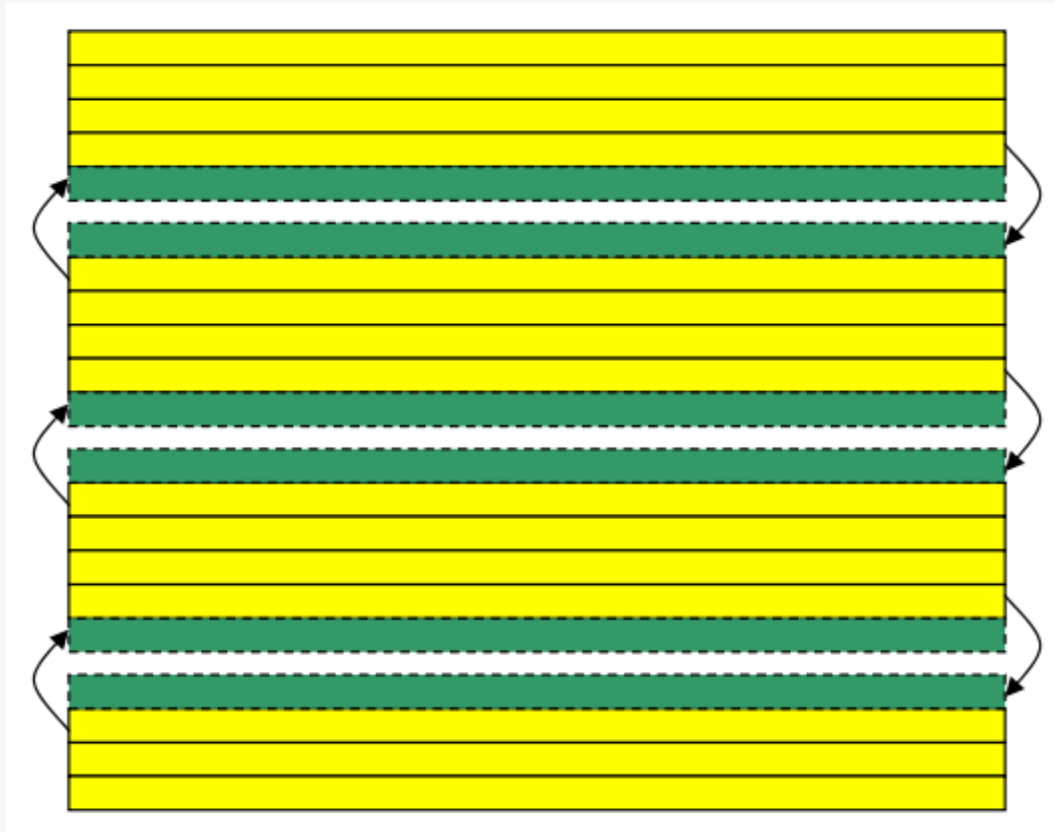
Décompositions sur mailles régulières



Problèmes : effets de bord si calculs non purement locaux

Décomposition des données

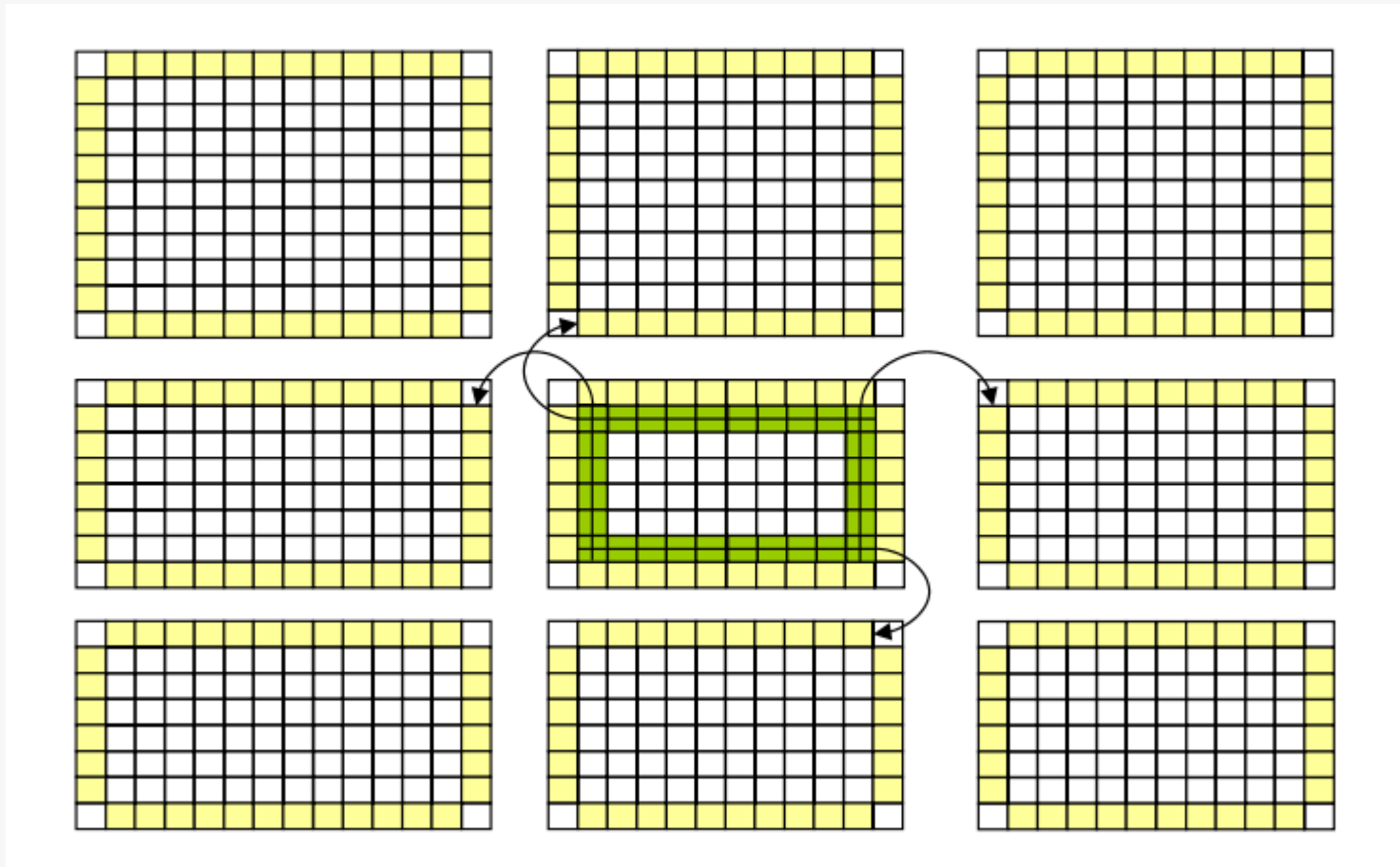
Solution : utilisation de zones de recouvrement



La taille des zones de recouvrement dépend de l'algorithme (compromis entre la fréquence des communications et les calculs supplémentaires)

Décomposition des données

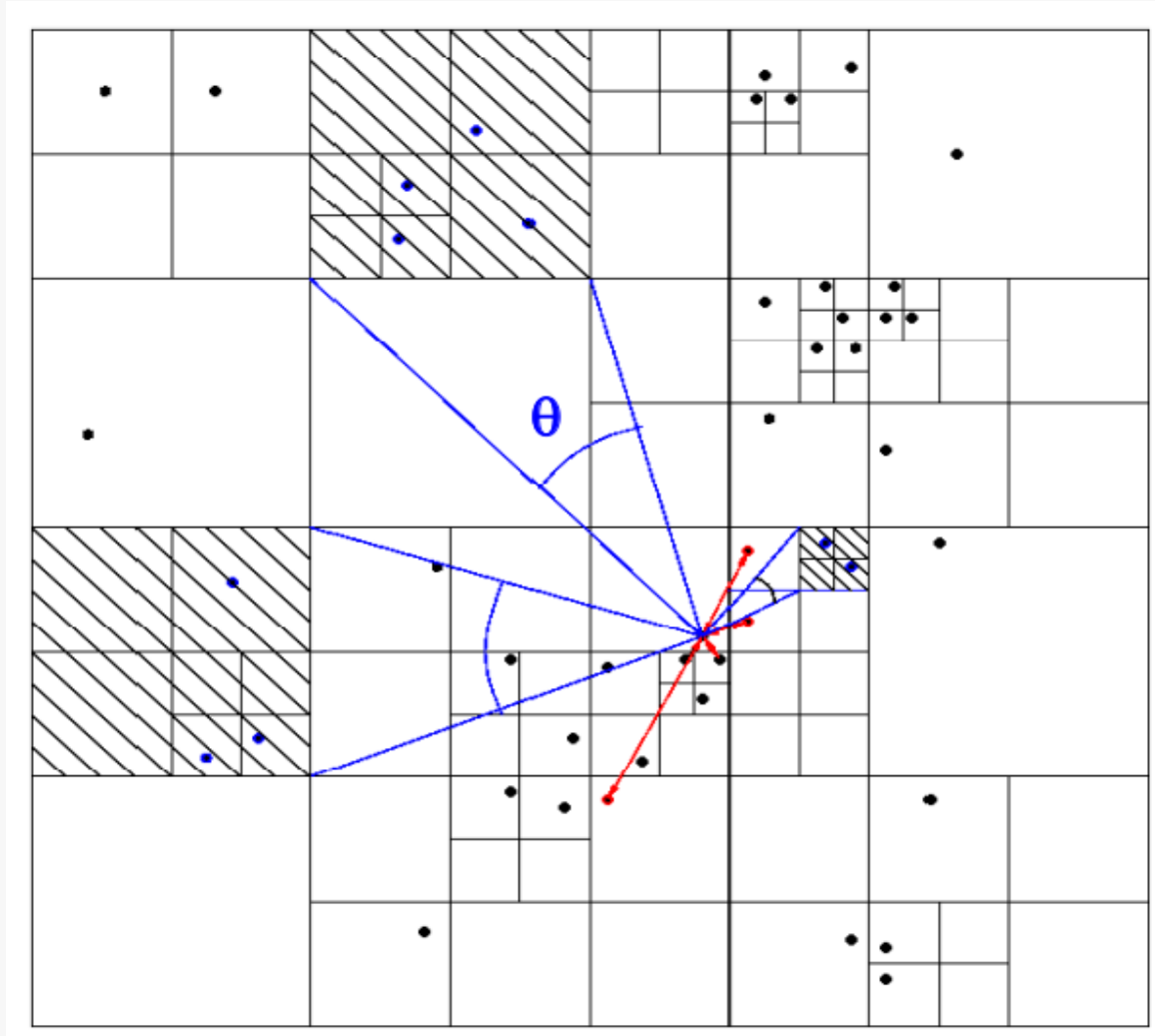
Zones de recouvrement pour décomposition en 2D



Pour ne pas trop se perdre, il est important de maintenir dans le code parallèle les références au tableau global tel que programmé en séquentiel

Décomposition des données

Exemple : décomposition problème à N corps



Également utilisé pour les méthodes multi-pôles

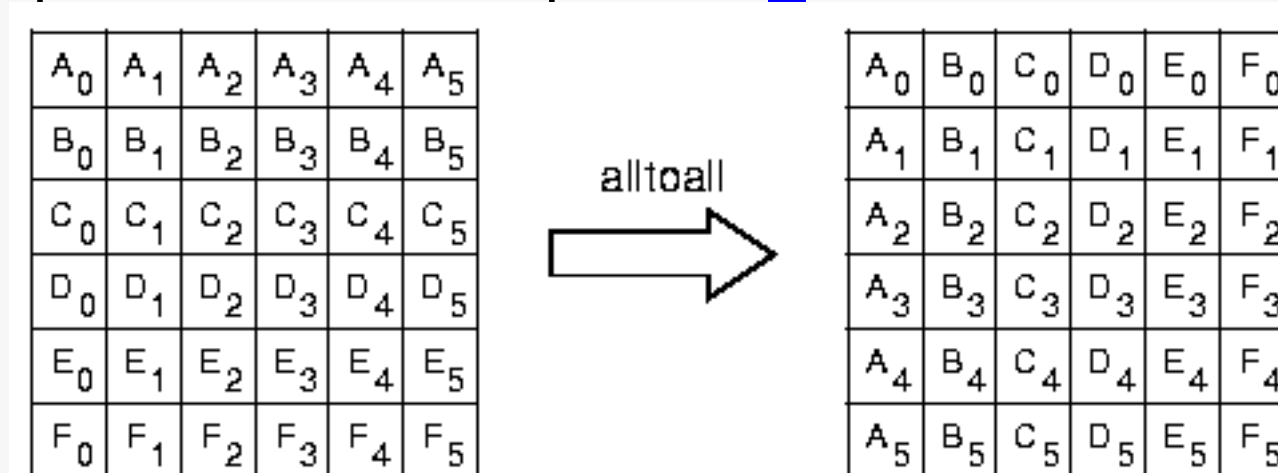
Décomposition des données

Transformées de Fourier rapide (FFT) de dimension $d > 1$

Exemple FFT 2D équivalente à une FFT 1D sur chaque ligne et chaque colonne

Chaque processus contient une série de lignes contiguës :
FFT 1D locales avec transposition entre les deux

Transposition effectuée par **`MPI_ALLTOALL`**



NB : dans la plupart des applications, la dernière transposition n'est pas nécessaire. En 3D, même principe : on se ramène à des FFTs 2D locales.

Conclusions

En guise de conclusion

Les atouts de MPI :

- MPI est un standard
- MPI est portable
- Communauté importante d'utilisateurs
- On peut paralléliser beaucoup de code avec seulement quelques fonctions MPI simples à utiliser

Néanmoins :

- Coder en parallèle n'est pas intuitif
- MPI n'est pas connu pour améliorer la lisibilité du code...
- Complexité du débogage : un ordre de grandeur supérieur
- Toujours se poser la question si cela en vaut la peine et si ça n'existe pas déjà : <http://www.mcs.anl.gov/mpi/libraries.html>