

Approche distribuée

`MPI_Send(buffer, count, type, destination, tag, communicateur)`

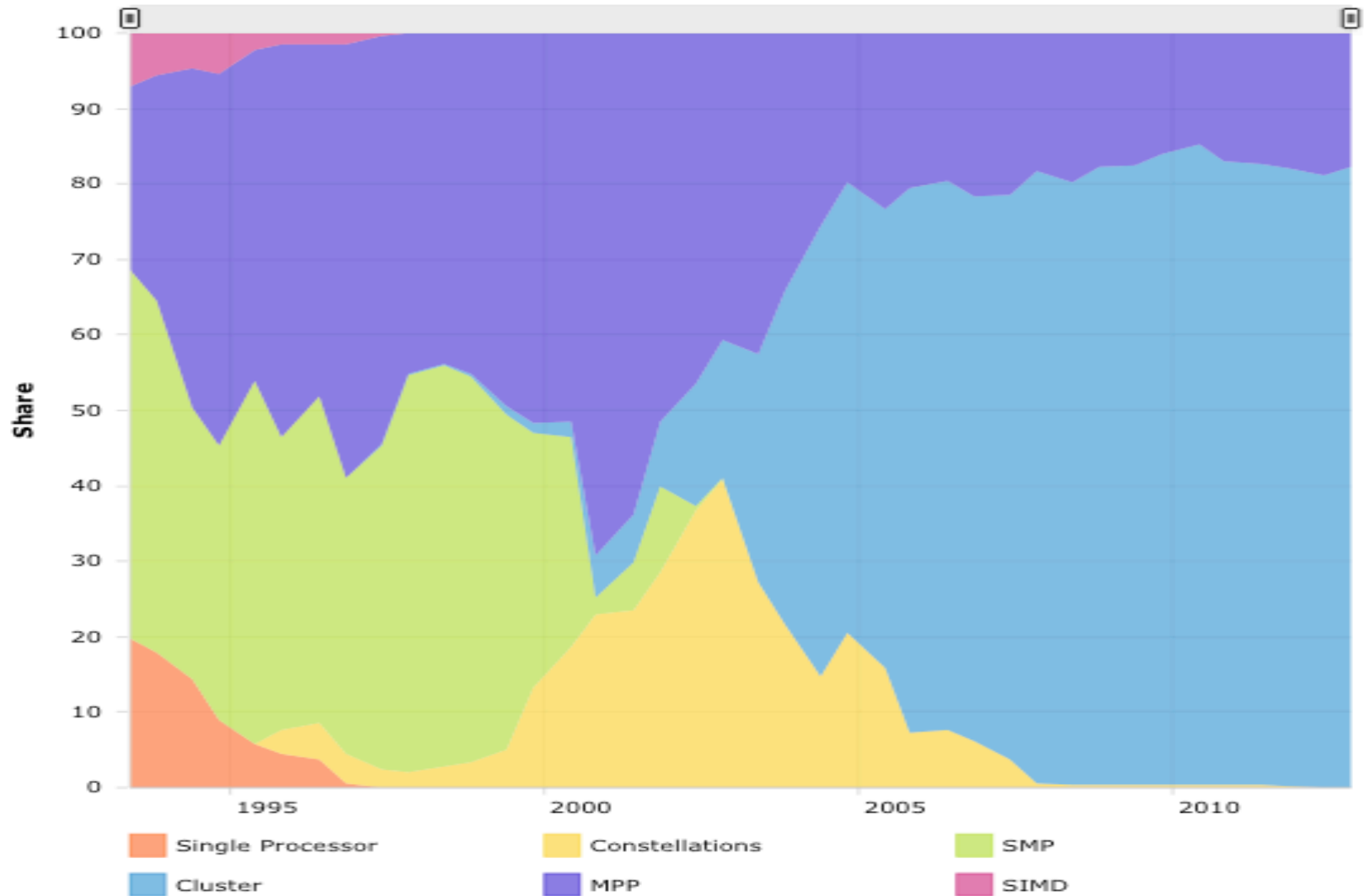
`MPI_Recv(buffer, count, type, source, tag, communicateur, &status)`

Approche distribuée

- Faire coopérer plusieurs machines à la résolution d'un même problème
- Permet de regrouper un nombre conséquent de processeurs
- Intérêts techniques et économiques
 - Approche plateformes de calculs
 - Partager la plateforme et le savoir faire
 - Faire évoluer le matériel progressivement
 - Limiter l'impact des pannes
 - Approche disséminée
 - Augmenter l'utilisation du matériel

TOP 500

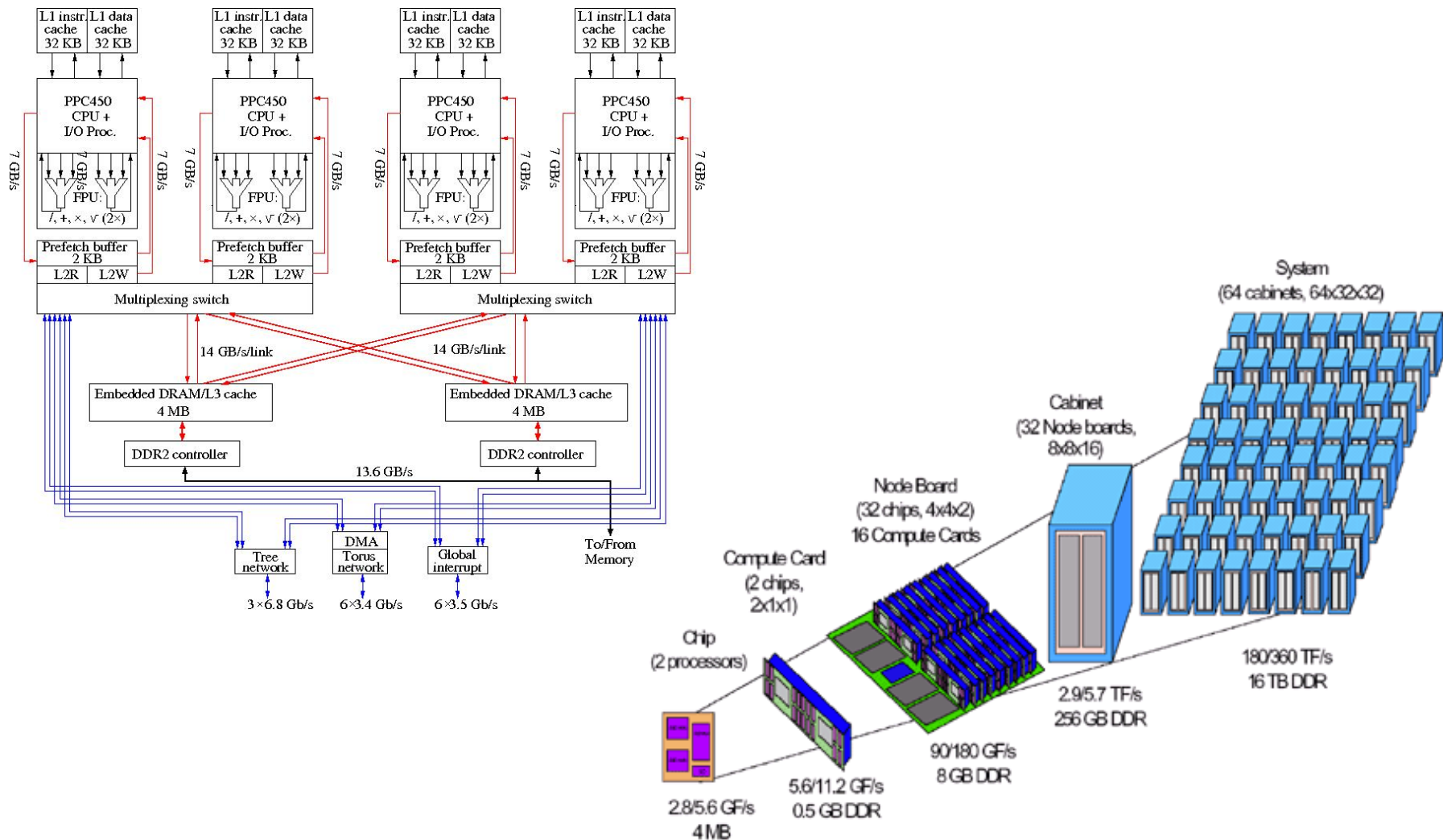
1993 - 2012



Différentes architectures

- Grappes de biprocesseurs (clusters)
- Grappes hétérogènes (GPU / xeon phi)
- Machines Massivement Parallèles (Blue gene)
- Grilles de calcul
- Cloud computing
- Peer2Peer

IBM Blue Gene



EGEE

eGEE
Enabling Grids
for E-science

Scheduled = 21539
Running = 25374

Application areas include:

Archeology
Astronomy
Astrophysics
Civil Protection
Comp. Chemistry
Earth Sciences
Finance
Fusion
Geophysics
High Energy Physics
Life Sciences
Multimedia
Material Sciences

...

>250 sites
48 countries
>50,000 CPUs
>20 PetaBytes
>10,000 users
>150 VOs
>150,000 jobs/day

21:13:50 UTC



GridPP
UK Computing for Particle Physics

Paradigmes de la programmation des architectures distribuées

- Approches explicites
 - Passage de message
 - Programmation à base de Send / Receive
 - appel de procédure à distance
 - Modèle client / serveur
 - Programmation à base de RPC,
 - Java JEE (RMI, Corba), WebService,
 - MapReduce
- Approches implicites
 - « mémoire virtuellement partagée »
 - « système distribué à image unique »

Paradigmes de la programmation des architectures distribuées

- Approches explicites → **Maitrise des performances**
 - Passage de message
 - Programmation à base de Send / Receive
 - appel de procédure à distance
 - Modèle client / serveur
 - Programmation à base de RPC,
 - Java JEE (RMI, Corba), WebService,...
 - MapReduce
- Approches implicites → **Facilité du développement**
 - « mémoire virtuellement partagée »
 - « système distribué à image unique »

Paradigme passage de message

- Participation de l'émetteur et du récepteur
- Communication point à point :
 - Send(destinataire, buffer, taille)
 - Receive(émetteur, buffer, taille)
- Communication collective
 - Broadcast(groupe,émetteur, buffer, taille)

Message Passing Interface

MPI_Send(buffer, count, type, destination, tag, communicateur)

MPI_Recv(buffer, count, type, source, tag, communicateur, &status)

- Standard industriel incontournable
 - les plateformes sont conçues pour faire tourner MPI
- Approche processus
- Bien plus utilisé qu'OpenMP
 - Même pour programmer les machines à mémoire commune
 - Qui peut le plus peut le moins
 - Assez bonne compréhension des performances
 - Pas de false sharing, pas de problème de cohérence mémoire
 - Les programmeurs cherchent à minimiser les communications
 - Oblige à un effort d'optimisation plus important que sur OpenMP
 - On observe de plus en plus de programmes hybrides MPI/OpenMP

Voir <https://computing.llnl.gov/tutorials/MPI>

Les points forts de MPI

- Position dominante
 - Incontournable
 - Standard (MPI 1.0: 1994 - 2.0 : 1997 - 3.0 : 2012)
 - Portable (C/Fortran)
- Simple et efficace
 - Les messages ne se doublent pas (ordre fifo sur un même canal)
 - Apport du support d'exécution
 - Utilisation transparentes de technologies réseau avancées
 - Possibilité de communiquer globalement (MPI-2.0)
 - Distribuer / regrouper des données en un appel de fonction
- En revanche
 - Ne pas avoir peur des calculs d'adresses
 - La difficulté croît avec le nombre de processeurs
 - Comment débbuger / optimiser un programme tournant sur 100 000 cœurs ?
 - Interface très riche
 - Trop ?

MPI Hello world

```
#include <stdio.h>
#include <MPI.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( "Hello world from process %d of %d\n",
            rank, size );

    MPI_Finalize();
    return 0;
}
```

> MPICC -o hello hello.c

> MPIexec -machinefile les-machines -n 4 hello

Hello world from process 1 of 4

Hello world from process 3 of 4

Hello world from process 0 of 4

Hello world from process 2 of 4

Émission / réception d'un buffer

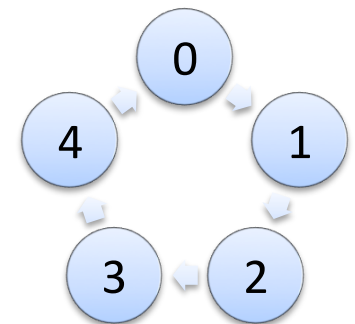
Le processus 1 réceptionne dans a le contenu d'un tableau envoyé par 0

```
int a[taille], tag = 0;
MPI_Statusq etat;
MPI_Init( &argc, &argv );
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...

If (rank == 0)
    MPI_send(&a, taille, MPI_INT, 1, tag, MPI_COMM_WORLD);
else
    MPI_recv(&a, taille, MPI_INT, 0, tag, MPI_COMM_WORLD, &etat);
```

Communication d'un jeton en anneau

```
if (rank == 0)
{
    printf( "Jeton lance par le maitre (%d participants) \n", size );
    MPI_Send(&token, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    MPI_Recv(&token, 1, MPI_CHAR, size-1, tag, MPI_COMM_WORLD, &etat);
    printf( "Jeton reçu par le maitre \n");
}
else
{
    MPI_Recv(&token, 1, MPI_CHAR, rank-1, tag, MPI_COMM_WORLD, &etat);
    printf( "Jeton chez %d \n", rank);
    MPI_Send(&token, 1, MPI_CHAR, (rank+1) % size, tag, MPI_COMM_WORLD);
}
```



Calculer $\text{out}[i] = f(\text{in}[i])$

- Le maitre envoie une partie du tableau *in* à chaque esclave
- L'esclave fait le calcul et retourne le résultat *out* au maitre.

Calculer $out[i] = f(in[i])$

- Le maitre envoie une partie du tableau *in* à chaque esclave

```
MPI_send( &in[(k-1) * tranche], tranche, MPI_INT,  
          k, TAG, MPI_COMM_WORLD)
```

```
MPI_recv( in, tranche, MPI_INT,  
          0, TAG, MPI_COMM_WORLD, &etat)
```

- L'esclave fait le calcul et retourne le résultat *out* au maitre.

```
MPI_recv( &out[(k-1) * tranche], tranche, MPI_INT,  
          k, TAG, MPI_COMM_WORLD, &etat)
```

```
MPI_send ( out, tranche, MPI_INT,  
          0, TAG, MPI_COMM_WORLD)
```


Calculer $\text{out}[i] = f(\text{in}[i])$

```
Int tranche = taille / size ;  
If (rank = 0)  
{  
    int in[taille], out[taille] ;  
    // initialiser in  
  
    for (k = 1; k < size; k++)  
        MPI_send( &in[(k-1) * tranche], tranche,  
                  MPI_INT, k, TAG, MPI_COMM_WORLD);  
  
    for (k = 1; k < size; k++)  
        MPI_recv( &out[(k-1) * tranche], tranche,  
                  MPI_INT, k, TAG, MPI_COMM_WORLD, &etat);  
}  
  
else // esclave  
{  
    int in[tranche], out[tranche];  
  
    MPI_recv( &in, tranche, MPI_INT,  
              0, TAG, MPI_COMM_WORLD, &etat);  
    // Calcul out = f(in)  
  
    MPI_send ( &out, tranche, MPI_INT,  
              0, TAG, MPI_COMM_WORLD);  
}
```

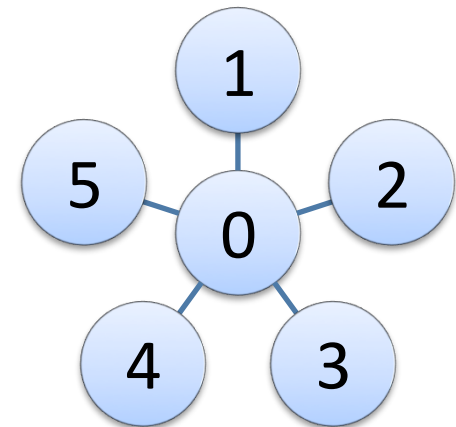
Utilisation de la variable status

- `status.MPI_SOURCE`
- `status.MPI_TAG`
- `status.MPI_ERROR`
- `int MPI_Get_count(&status, datatype, &count)`
- `int MPI_Probe(source, tag, comm, &status)`
 - Attendre un message sans en effectuer la réception
 - La variable status est renseignée
 - Ex. : dimensionner un buffer de réception

Utilisation de status : jeton centralisé

Schéma de calcul Maître / Esclave

```
if (rank == 0) {  
    for(i = 0; i < 3*(size-1); i++) {  
        MPI_Recv(&demande, 1, MPI_CHAR, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &etat);  
        MPI_Send(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD);  
        MPI_Recv(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD, &etat);  
    }  
    printf( " done \n");  
} else  
    for(i = 0; i < 3; i++){  
        MPI_Send(&demande, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
  
        MPI_Recv(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &etat);  
        printf( "Jeton chez %d \n", rank);  
        MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
    }  
}
```



Utilisation de status

Dimensionner un buffer de réception

```
if (rank == 0)
    MPI_send(&a, taille, MPI_INT, 1, tag, MPI_COMM_WORLD);
else
{
    MPI_Status status;
    int count, *buf;

    MPI_Probe(0, tag, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT, &count)

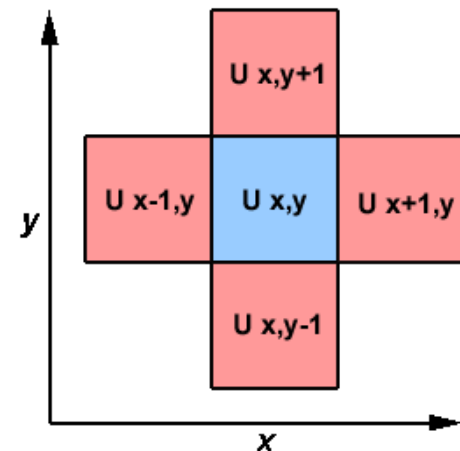
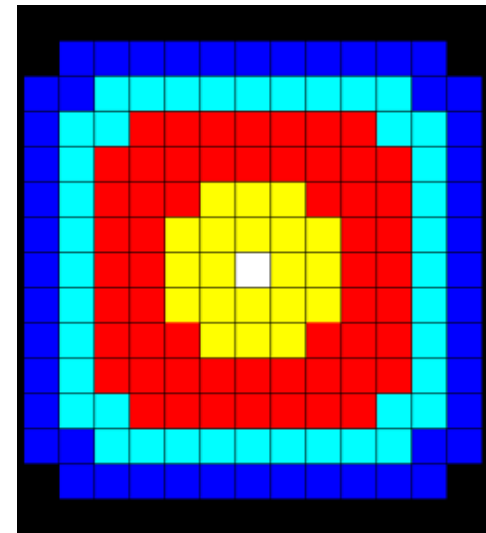
    buf = malloc(count * sizeof (int));
    MPI_Recv(buf, taille, MPI_INT, 0, tag, MPI_COMM_WORLD, &etat);
}
```

Étude de cas

Stencil / Pochoir

- Méthode itérative de calcul de la valeur des éléments d'un tableau
- La valeur suivante est fonction des cellules voisines.
- Les cellules utiles au calcul d'une cellule forment un motif : le pochoir
- Applications :
 - Résolution EDP
 - Résolution système linéaire
 - simulation

$$\begin{aligned}U_{x,y} &= U_{x,y} \\ &+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{x,y}) \\ &+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})\end{aligned}$$

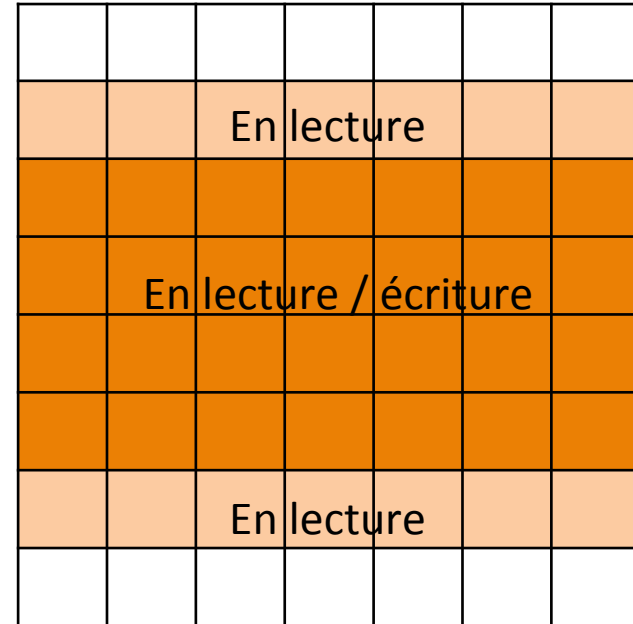
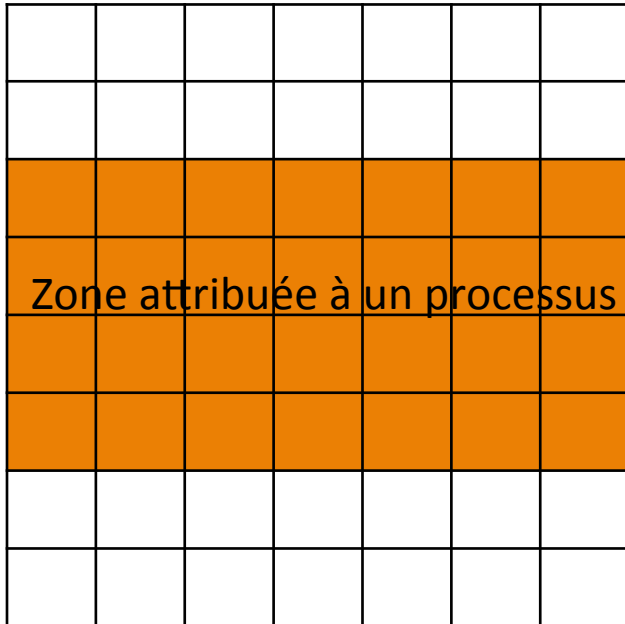


Calcul distribué d'un stencil

- Algorithme pour n étapes
 - Code du maitre :
 - Distribuer le domaine aux esclaves
 - Réceptionner les contributions
 - Code des esclaves
 - Pour n étapes :
 - Calculer l'état du domaine à l'étape suivante
 - Communiquer avec les voisins
 - » obtenir / diffuser la valeur des bords
 - Envoyer le résultat au maitre

Initialisation

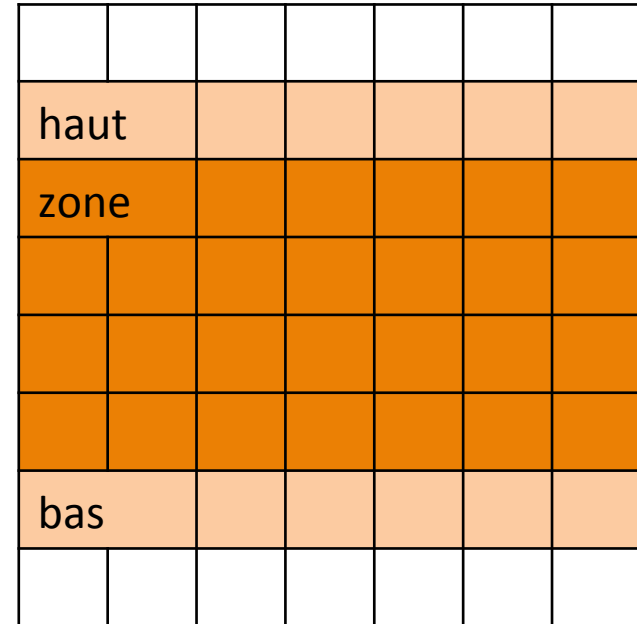
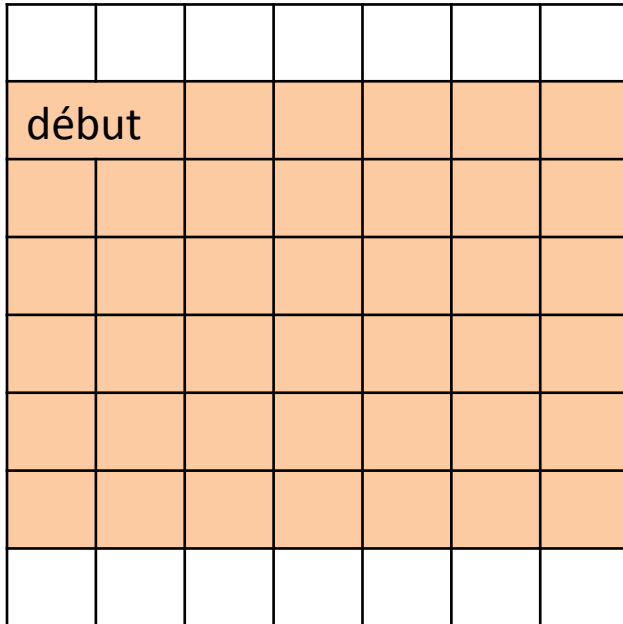
distribution du domaine



Le domaine de lecture est plus large que celui d'écriture

Initialisation

distribution du domaine



Le domaine de lecture est plus large que celui d'écriture
Adresse du début de tableau de l'esclave k :

$\text{tranche} = \text{nb_lignes} / (\text{size}-1)$

$\text{debut} : (k * \text{tranche} - 1) * \text{NBCOL}$

$\text{taille} : (\text{tranche} + 2) * \text{NBCOL}$

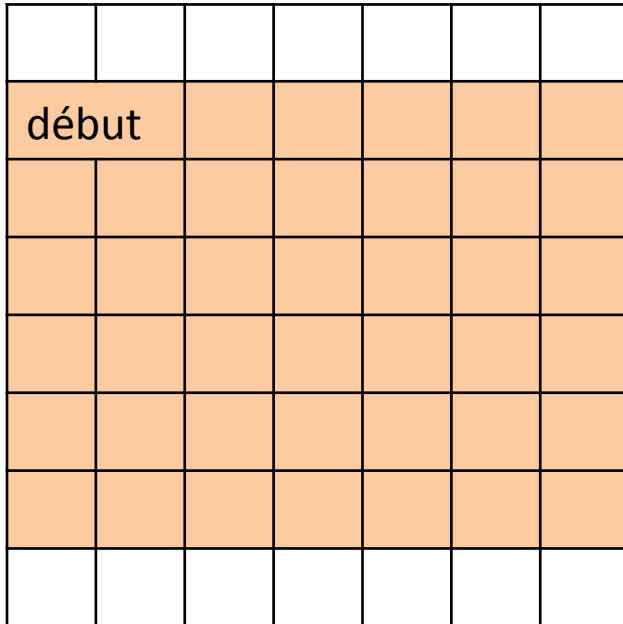
$\text{haut} : \text{in}$

$\text{zone} : \text{in} + \text{NBCOL}$

$\text{bas} : \text{in} + \text{tranche} * \text{NCOL}$

Initialisation

distribution du domaine



Le domaine de lecture est plus large que celui d'écriture

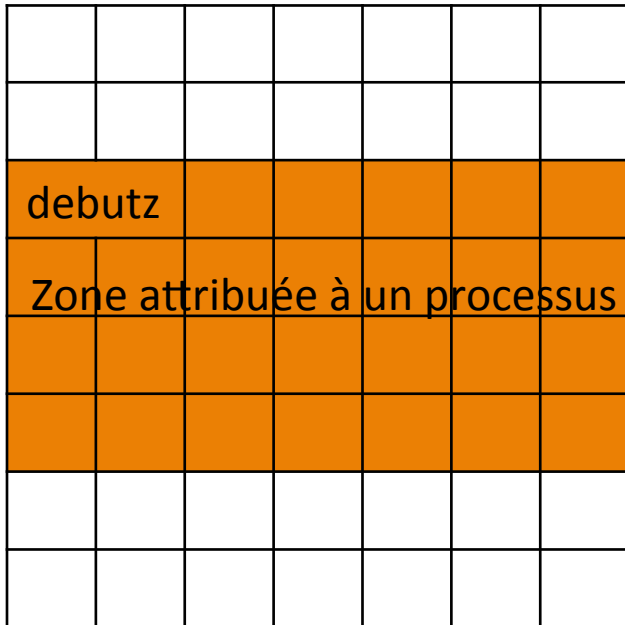
debut : $(k * tranche - 1) * NBCOL$ taille : $(tranche + 2) * NBCOL$

`MPI_Send(in+debut, taille, MPI_CHAR, k, TAG, MPI_COMM_WORLD)`

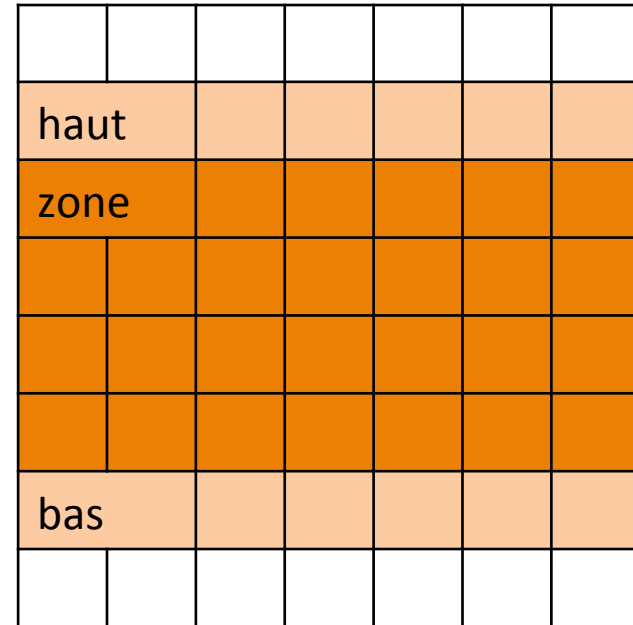
`MPI_Recv(haut, taille, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &etat`

Finalisation

Réception du domaine



debutz : $k * \text{tranche} * \text{NBCOL}$



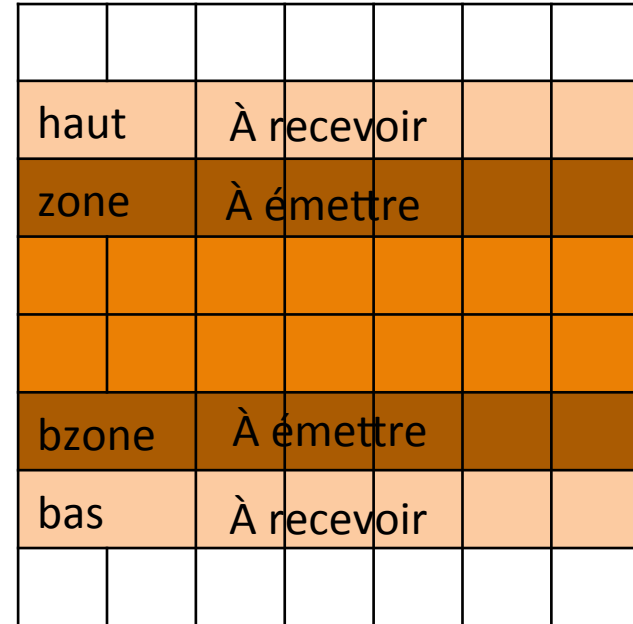
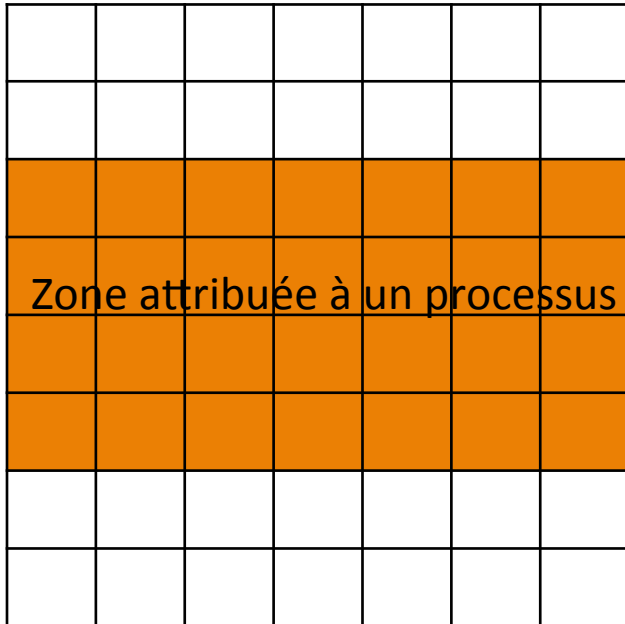
taillez : $\text{tranche} * \text{NBCOL}$

`MPI_Send(zone, taillez, MPI_CHAR, 0, TAG, MPI_COMM_WORLD)`

`MPI_Recv(in+debutz, taillez, MPI_CHAR, k, TAG, MPI_COMM_WORLD,`

Itération

émission / réception des bords



`MPI_Send(zone, NBCOL, MPI_CHAR, k-1, ...`

`MPI_Recv(haut, NBCOL, MPI_CHAR, k-1, ...`

`MPI_Send(bzone, NBCOL, MPI_CHAR, k+1,`

`MPI_Recv(bas, NBCOL, MPI_CHAR, k+1, ...`

Itération

émission / réception des bords

Processus k-1

MPI_Send(bzone, ... , k, ...

MPI_Recv(bas, ..., ... , k, ...

MPI_Send(zone, ..., ..., k-1, ...

MPI_Recv(haut, ..., ..., k-1, ...

MPI_Send(bzone, ..., ..., k+1,

MPI_Recv(bas, ..., ..., k+1,

MPI_Send(zone, , ... , ..., k, ...

MPI_Recv(haut, ... , ..., k, ...

Processus k+1

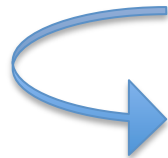

Itération

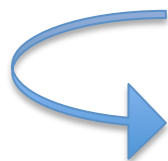
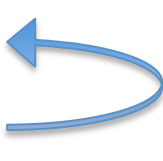
émission / réception des bords

Processus k-1

MPI_Send(bzone, ... , k, ...

MPI_Recv(bas, ..., ... , k, ...

 MPI_Send(zone, ..., ..., k-1, ...
MPI_Recv(haut, ..., ..., k-1, ... 

 MPI_Send(bzone, ..., ..., k+1,
MPI_Recv(bas, ..., ..., k+1, 

MPI_Send(zone, , ... , ..., k, ...

MPI_Recv(haut, ... , ..., k, ...

Processus k+1

Itération

émission / réception des bords

Processus k-1

MPI_Send(bzone, ... , k, ...

MPI_Recv(bas, ..., ... , k, ...

MPI_Recv(haut, ..., ..., k-1, ...

MPI_Send(zone, ..., ..., k-1, ...

MPI_Recv(bas, ..., ..., k+1,

MPI_Send(bzone, ..., ..., k+1,

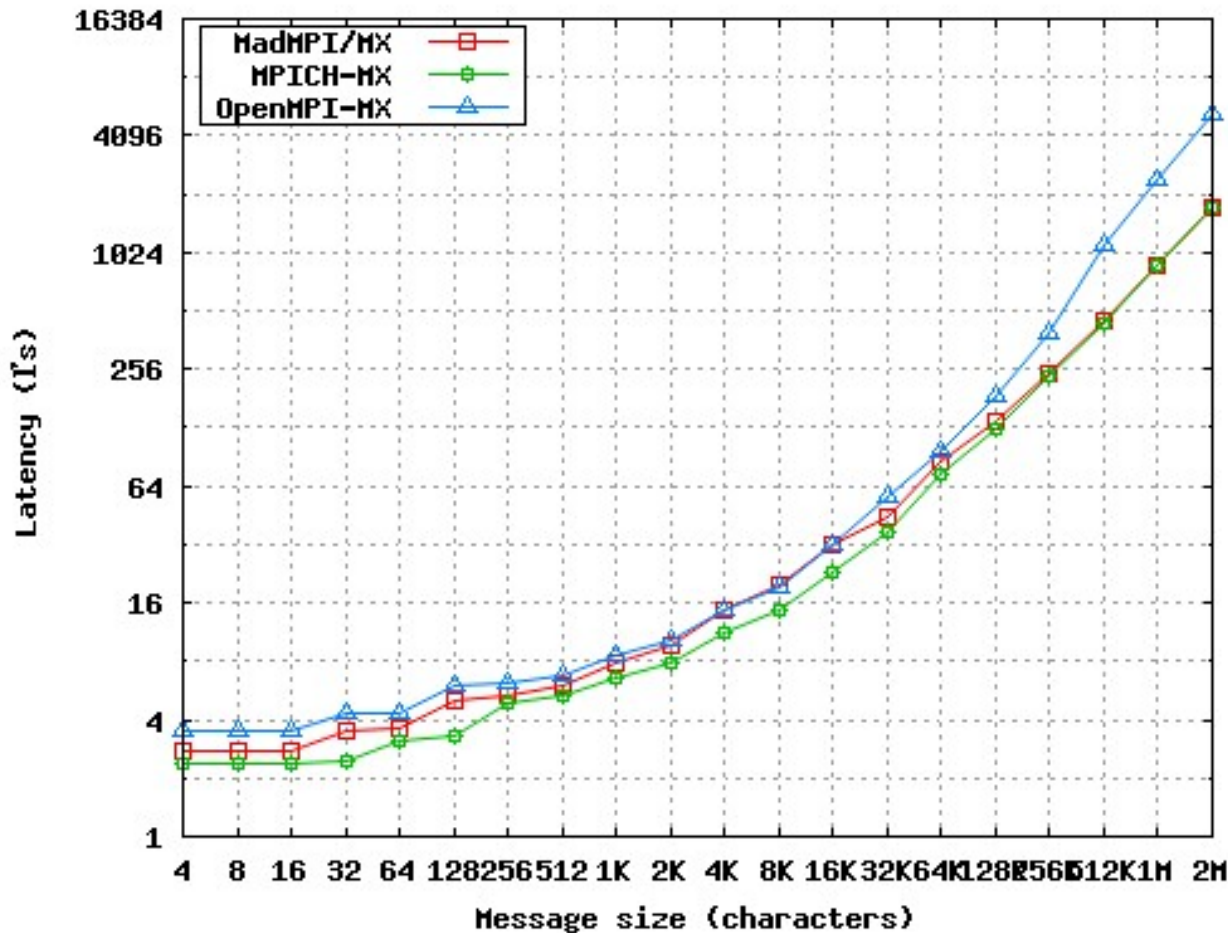
Jouer sur
la parité
de k

MPI_Send(zone, , ... , ..., k, ...

MPI_Recv(haut, ... , ..., k, ...

Processus k+1

Réduction du nombre de messages



Latence pour $8 * 1\text{ko} = 8 * 8\text{us}$
Latence pour $8\text{ko} = 16\text{ us}$

Latence = amorce + Volume / débit max

Réduction du nombre de messages

- On peut calculer l'état d'une cellule à l'étape k si on connaît l'état des cellules à distance k.

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

0	0	0	0	0
0	1	1	1	0
0	1	2	1	0
0	1	1	1	0
0	0	0	0	0

- Remplacer des synchronisations par du calcul redondant
 - Travailler avec un bord épais (shadow-zone)
 - Les cellules du bords sont calculées par les deux processus

Réduction du nombre de messages

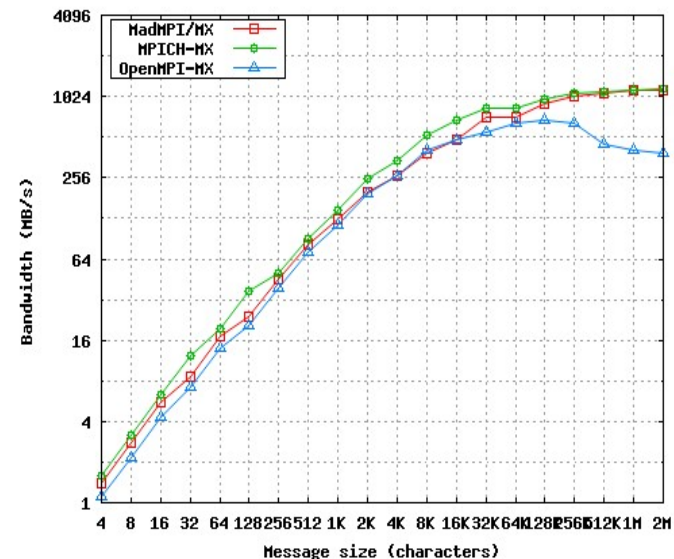
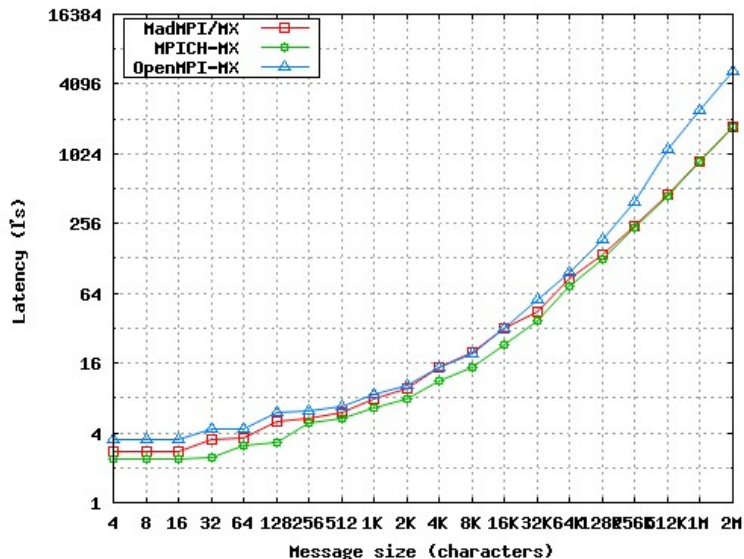
- Algorithme de l'esclave avec des cellules fantômes (*shadow zone*) de s lignes
 - Recevoir sa zone et les s lignes du *haut* et du *bas*
- Pour n/s étapes :
 - Calculer l'état du domaine pour les s étapes suivantes
 - À la sous étape x on calcule $2 * (s-x)$ lignes en plus
 - Communiquer avec les voisins
 - obtenir / diffuser la valeur des s première / dernières lignes
- Envoyer le résultat au maitre

Couches basses

- Matériel performant
 - Faible latence, gros débit
 - Grand nombre de communications en parallèle
 - Gestion de la congestion avant tout par le surdimensionnement
 - Matériel fiable
 - Correction à la volée des erreurs
 - Routage simplifié
 - Éviter la pile TCP / IP au sein des clusters
- Logiciel de communication performant
 - Permettre le calcul durant les communications
 - Grand nombre de communications en parallèle

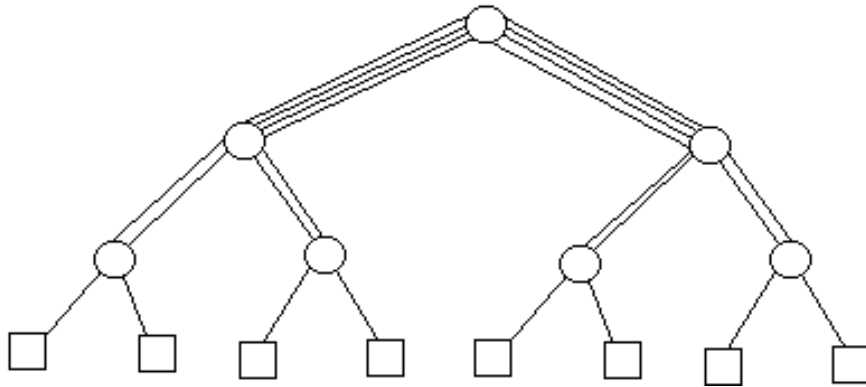
Objectifs des communications dans le cadre HPC

- Transmission performante – réseaux rapides
- Exemple carte infiniband (mellanox)
 - De l'ordre de 100Gb/s
 - Plus de 130M messages/sec
 - 1 micro seconde MPI latence

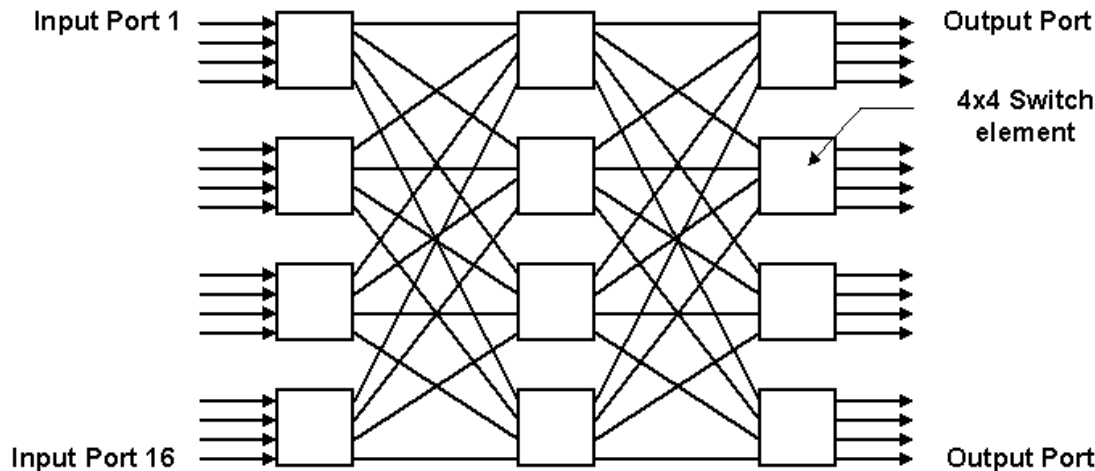
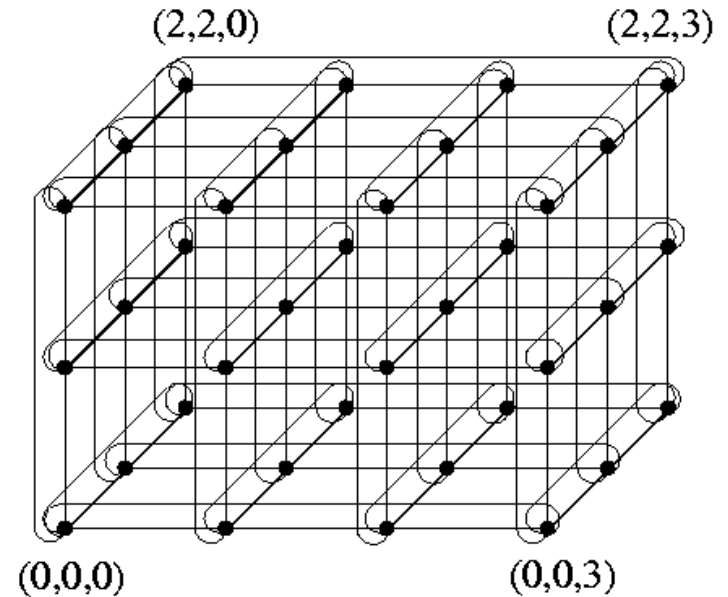


Topologies des réseaux

Fat Tree – 3D Torus - Clos



Clos
Network

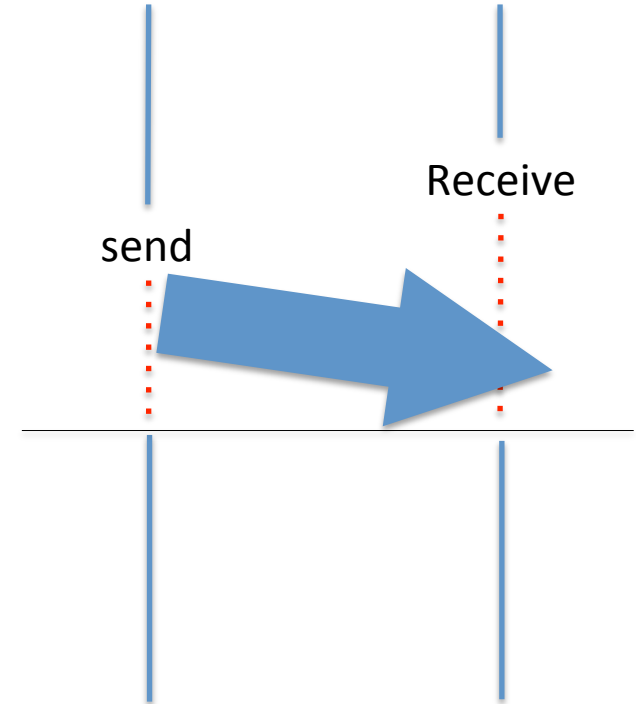


Être capable d'avoir beaucoup
de communications en parallèle

Calculer tout en communiquant

Permettre le calcul durant les communications

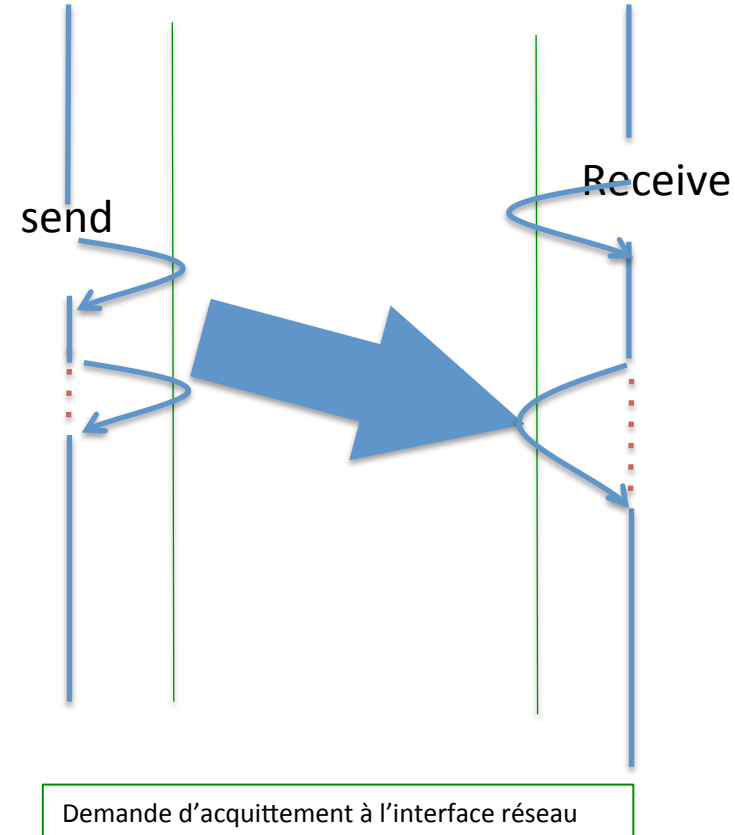
- Ne pas chercher à *synchroniser* le send et le receive
- Découpler les appels à send/receive de l'émission / réception physique des données



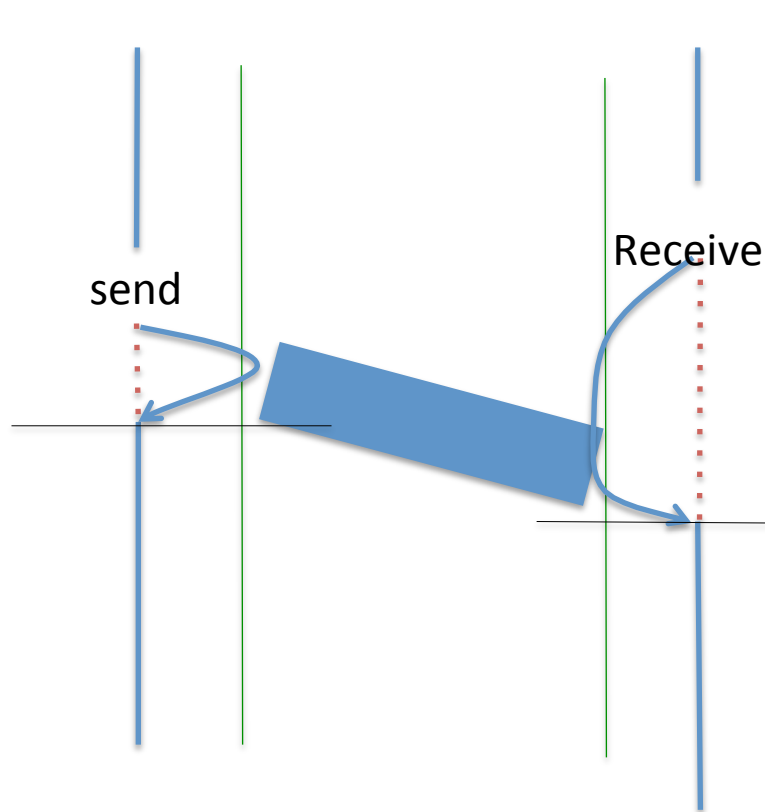
Calculer tout en communiquant

communication en arrière-plan

- Émission d'une requête
 - D'émission ou de réception
- Le processus calcule durant la communication
- Attente d'un acquittement, soit
 - Pour (ré)utiliser le buffer
 - Pour se synchroniser avec le récepteur

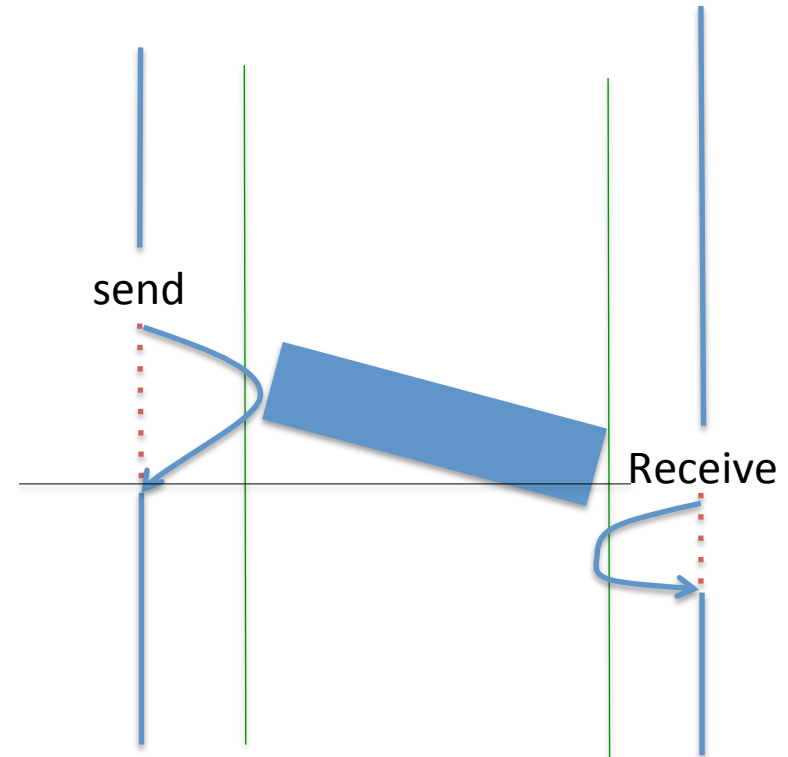


Mode bloquant vs mode asynchrone



Mode bloquant l'appel retourne
dès que le buffer est disponible

Mode immédiat = non bloquant



Mode synchrone : l'appel retourne
dès que l'on sait que le récepteur
a posté le receive
Mode asynchrone.

Les modes de communication de MPI

- Bloquant

`MPI_Send(buffer,count,type,dest,tag,comm)`

`MPI_Recv(buffer,count,type,source,tag,comm,status)`

- Immédiat (Non bloquant)

`MPI_Isend(buffer,count,type,dest,tag,comm,request)`

`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Il faut faire attention à ce que le buffer ne soit pas être (ré-)utilisé trop tôt en émission comme en réception

`MPI_Wait (&request,&status)`

`MPI_Test (&request,&resultat,&status)`

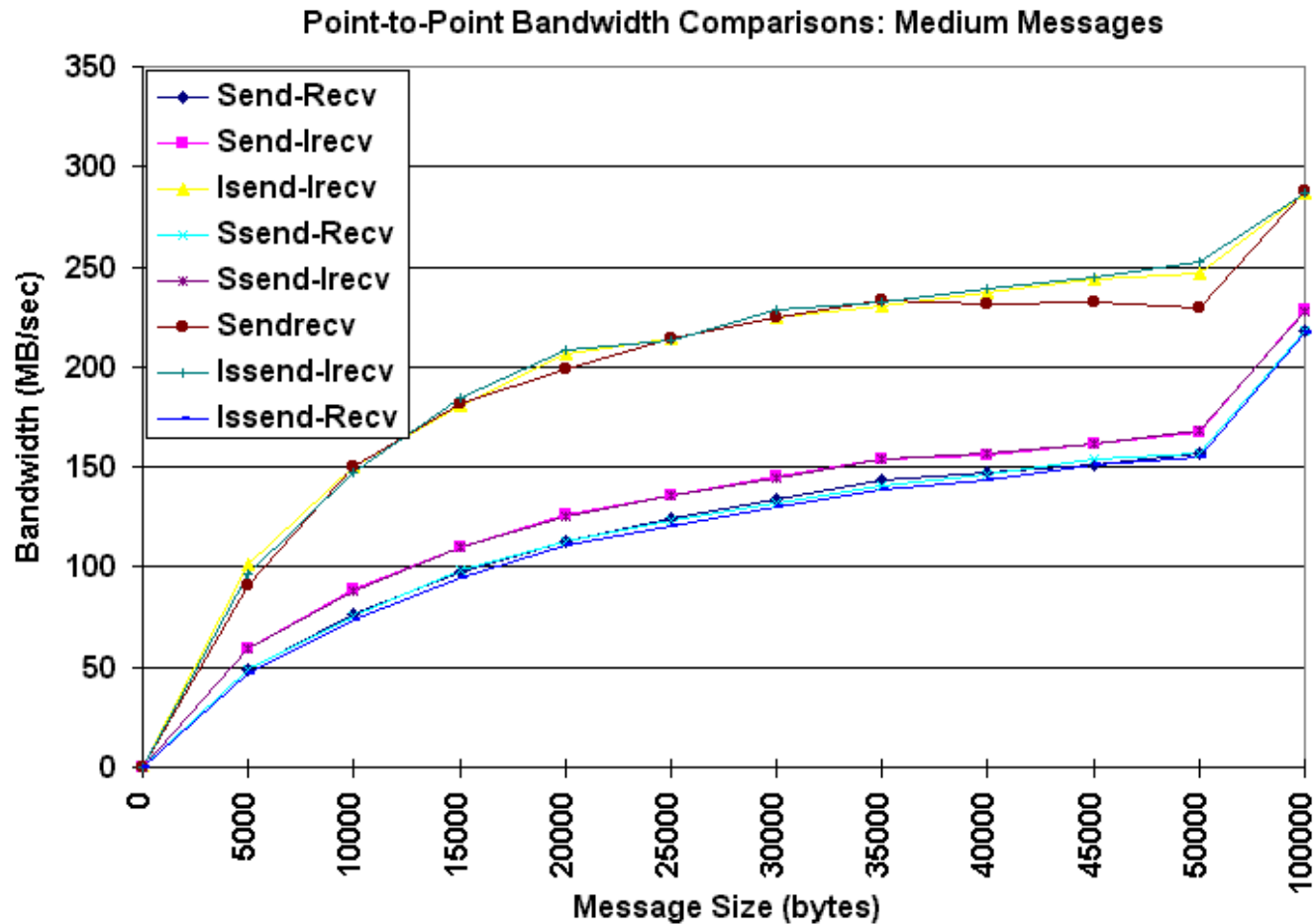
`MPI_Waitall (count,&array_of_requests,&array_of_statuses)`

Autres primitives de communication

- Ssend (synchronous send) termine lorsque l'émetteur sait que la requête de réception a été postée et que le buffer peut être réutilisé
- Rsend (Ready send) termine lorsque le buffer peut être réutilisé
 - Le recv correspondant est supposé préalablement posté
 - Évite la demande de rendez-vous
- Bsend termine après recopie des données dans un buffer
 - Buffer_attach(void*,size_t) / Buffer_detach pour allouer *le* buffer du processus
- Irsend, lbsend, lssend
 - influent sur l'interprétation du test/wait qui indique la fin de la réception

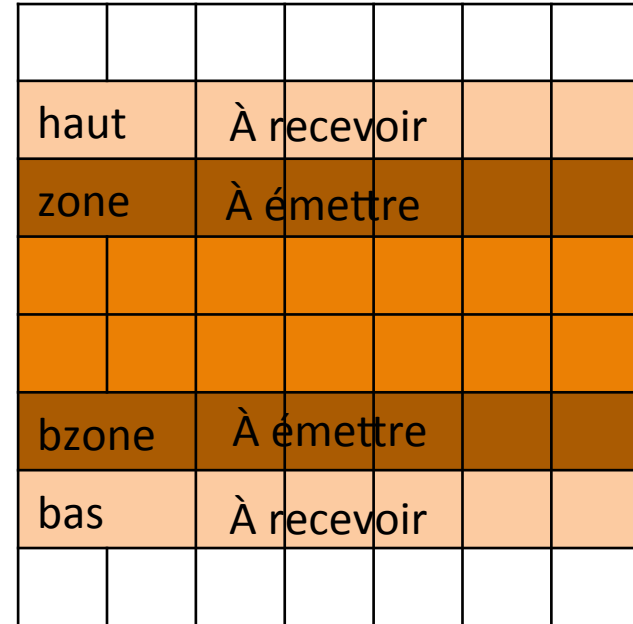
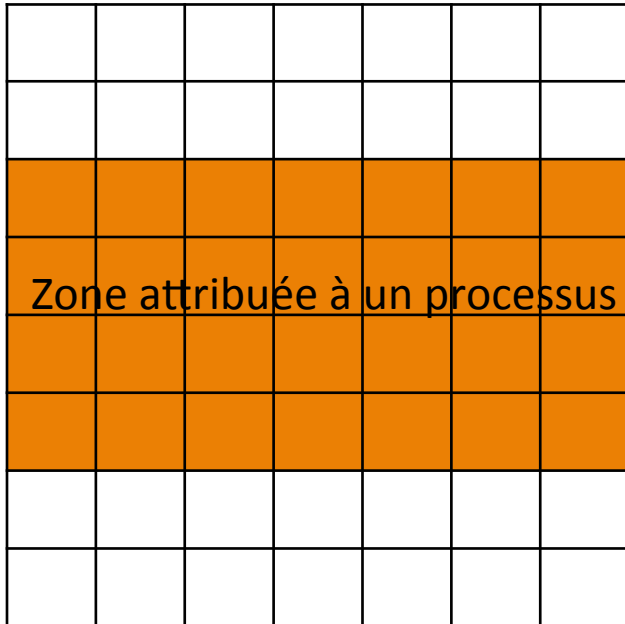
Performances (LLNL)

https://computing.llnl.gov/tutorials/MPI_performance/



Retour sur le calcul itératif

émission / réception des bords



`MPI_Send(zone, NBCOL, MPI_CHAR, k-1, ...`

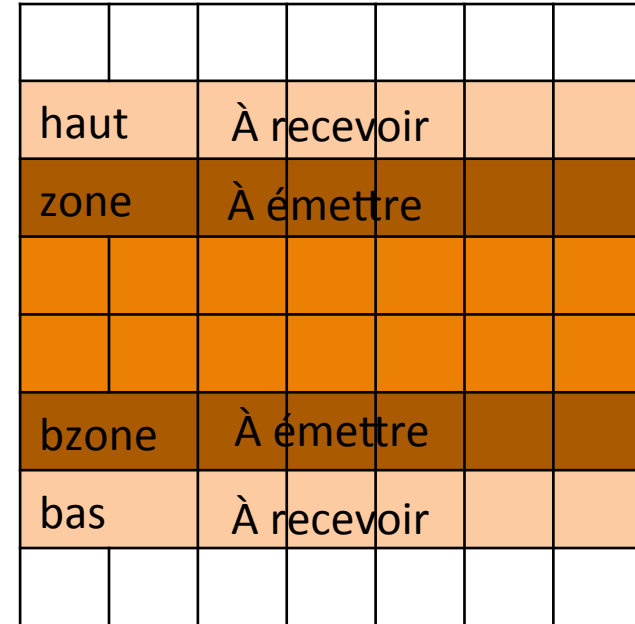
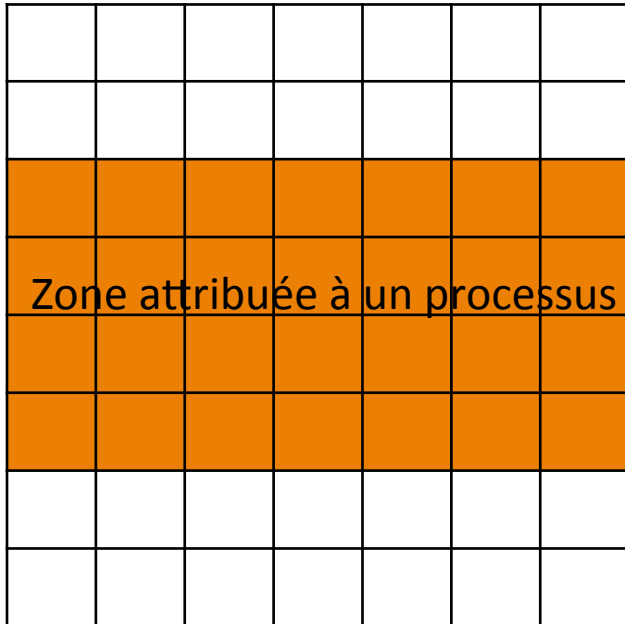
`MPI_Recv(haut, NBCOL, MPI_CHAR, k-1, ...`

`MPI_Send(bzone, NBCOL, MPI_CHAR, k+1,`

`MPI_Recv(bas, NBCOL, MPI_CHAR, k+1, ...`

Calcul itératif

émission / réception des bords



```

MPI_Isend( zone, NBCOL, MPI_CHAR, k-1, ... ,    &req1)
MPI_Irecv( haut, NBCOL, MPI_CHAR, k-1, ...,    &req2)
MPI_Isend(bzone, NBCOL, MPI_CHAR, k+1, ...,    &req3)
MPI_Irecv(bas, NBCOL, MPI_CHAR, k+1, ...,    &req4)
    
```

Recouvrement des communications par le calcul

```
While(...){  
  // Calcul des bords  
  ...  
  MPI_Isend( zone, NBCOL, MPI_CHAR, k-1, ... ,    &req1)  
  MPI_Irecv( haut, NBCOL, MPI_CHAR, k-1, ...,    &req2)  
  MPI_Isend(bzone, NBCOL, MPI_CHAR, k+1, ...,    &req3)  
  MPI_Irecv(bas, NBCOL, MPI_CHAR, k+1, ...,    &req4)  
  
  // calcul des cellules du centre  
  ...  
  MPI_Wait(&req1, & status1)  
  MPI_Wait(&req2, & status2)  
  MPI_Wait(&req3, & status3)  
  MPI_Wait(&req4, & status4)  
  
}
```

Recouvrement des communications par le calcul

```
MPI_Request req[4];  
MPI_Status stats[4];
```

```
While(...){  
    // Calcul des bords  
    ...  
    MPI_Isend( zone, NBCOL, MPI_CHAR, k-1, ... ,    req)  
    MPI_Irecv( haut, NBCOL, MPI_CHAR, k-1, ...,    req+1)  
    MPI_Isend(bzone, NBCOL, MPI_CHAR, k+1, ...,    req+3)  
    MPI_Irecv(bas, NBCOL, MPI_CHAR, k+1, ...,    req+4)  
  
    // calcul des cellules du centre  
    ...  
    MPI_Waitall(req1, stats)  
  
}
```

Optimisations

Communications persistantes

Enregistrer des requêtes pour les rejouer plusieurs fois

- Économise du temps sur la création de la requête
- Factorise du code
- `int MPI_Send_init(buf, count, datatype, dest, tag, comm, &request)`
- `int MPI_Recv_init(buf, count, datatype, source, tag, comm, &request)`
- `int MPI_Start(&request)`
- `int MPI_Startall(count, array_of_requests)`
- `int MPI_Waitall(count, array_of_requests, array_of_statuses)`
- `int MPI_Waitany(count, array_of_requests, &index, &status)`
- `int MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`

Communications persisentes

```
MPI_Request req[4];
MPI_Status stats[4];
MPI_Send_Init( zone, NBCOL, MPI_CHAR, k-1, ... , req)
MPI_Recv_Init( haut, NBCOL, MPI_CHAR, k-1, ..., req+1)
MPI_Send_Init(bzone, NBCOL, MPI_CHAR, k+1, ..., req+3)
MPI_Recv_Init(bas, NBCOL, MPI_CHAR, k+1, ..., req+4)

While(...){

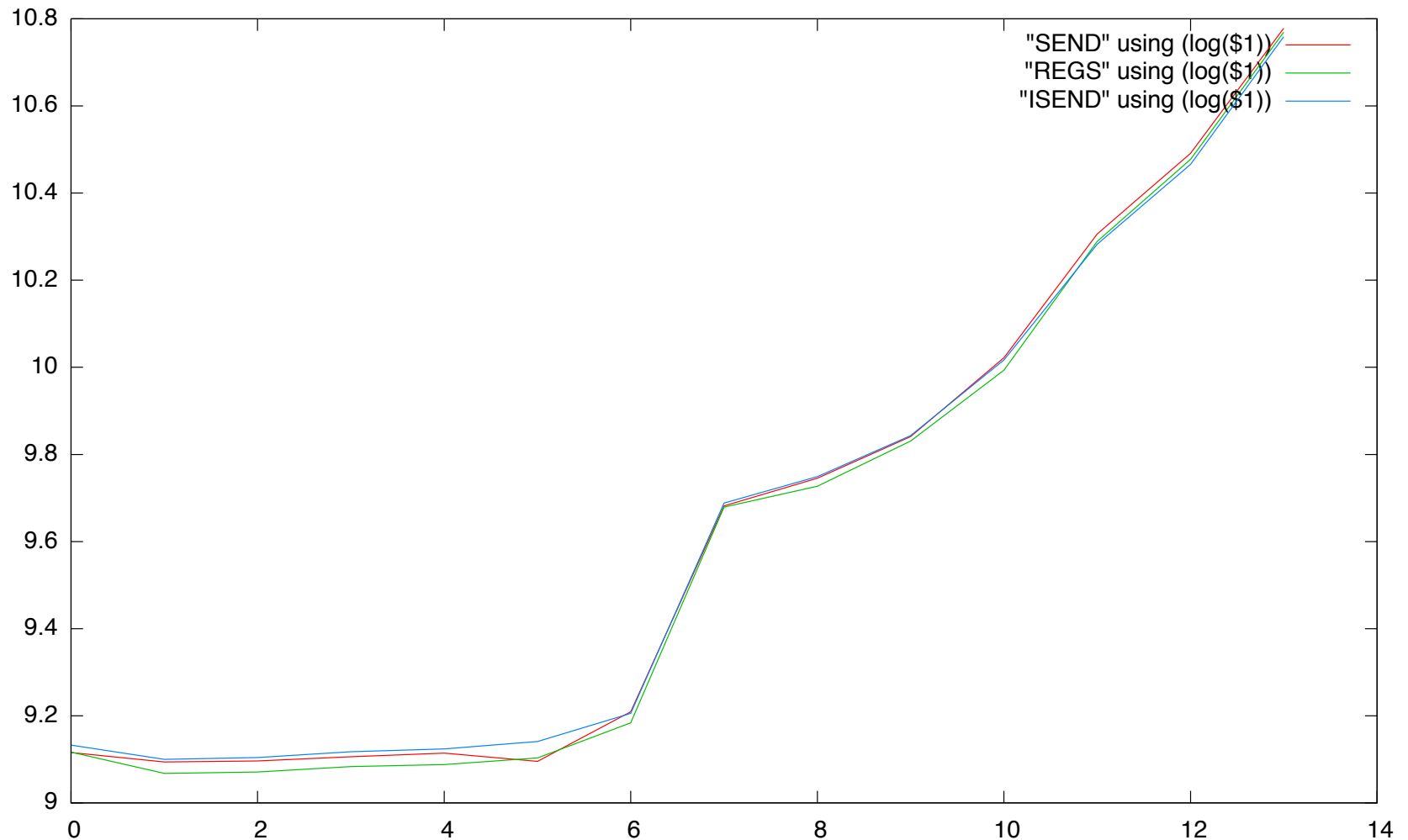
    // Calcul des bords
    ...
    MPI_Startall(4,req);

    // calcul des cellules du centre
    ...
    MPI_Waitall(req1, stats)

}
```


Expérience sur infini

~200 cycles de gain sur un ping/pong

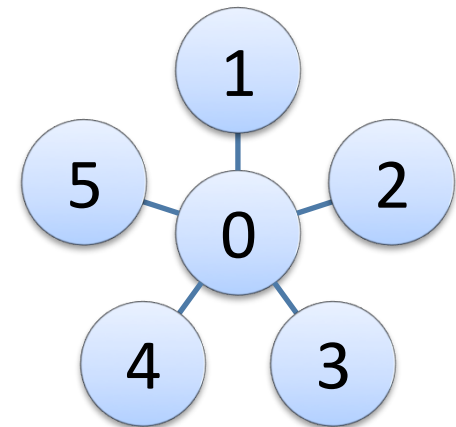


On ne gagne rien mais c'est jolie

Retour sur le jeton centralisé

Schéma de calcul Maître / Esclave

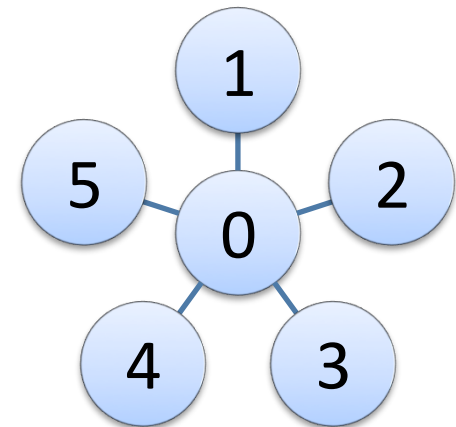
```
if (rank == 0) {  
    for(i = 0; i < 3*(size-1); i++) {  
        MPI_Recv(&demande, 1, MPI_CHAR, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &etat);  
        MPI_Send(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD);  
        MPI_Recv(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD, &etat);  
    }  
    printf( " done \n");  
} else  
    for(i = 0; i < 3; i++){  
        MPI_Send(&demande, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
  
        MPI_Recv(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &etat);  
        printf( "Jeton chez %d \n", rank);  
        MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
    }  
}
```



Retour sur le jeton centralisé

Schéma de calcul Maître / Esclave

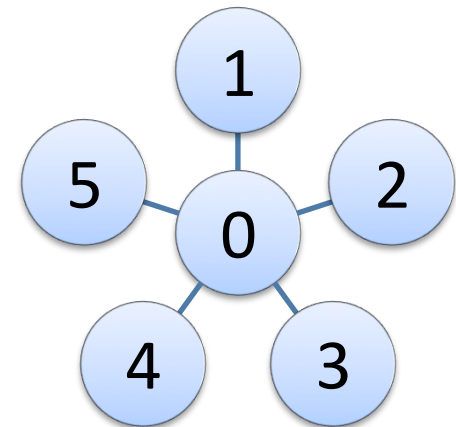
```
if (rank == 0) {  
    for(i = 0; i < 3*(size-1); i++) {  
        MPI_Recv(&demande, 1, MPI_CHAR, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &etat);  
        MPI_Send(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD);  
        MPI_Recv(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD, &etat);  
    }  
    printf( " done \n");  
} else  
    for(i = 0; i < 3; i++){  
        MPI_Send(&demande, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
  
        MPI_Recv(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &etat);  
        printf( "Jeton chez %d \n", rank);  
        MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
    }  
}
```



Jeton centralisé

Mode immédiat et ready send

```
if (rank == 0) {  
    for(i = 0; i < 3*(size-1); i++) {  
        MPI_Recv(&demande, 1, MPI_CHAR, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &etat);  
        MPI_Irecv(&token, T, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD, &etat,&req);  
        MPI_Rsend(&token, T, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD);  
        MPI_Wait(&req,&etat);  
    }  
    printf( " done \n");  
} else  
    for(i = 0; i < 3; i++){  
        MPI_Irecv(&token, T, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &etat,&req);  
        MPI_Send(&demande, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
        MPI_Wait(&req,&etat);  
        printf( "Jeton chez %d \n", rank);  
        work();  
        MPI_Rsend(&token, T, MPI_CHAR, 0, 2, MPI_COMM_WORLD);  
    }  
}
```



Communications collectives

- Barrière, diffusion à tous, ...
 - Schémas de communication classiques
 - Propices à optimisations puisque réalisables en parallèle
- Opérations réalisées par tous les processus d'un communicateur
 - Les processus doivent poster les requêtes dans le même ordre
 - Non synchronisantes
 - sauf MPI_Barrier(communicator)

Multiplication de matrices

version point à point

```
// envoyer la matrice b à tous
for(i=1;i<numprocs;i++)
    MPI_Send(b,N*N,MPI_CHAR,i,0,MPI_COMM_WORLD);

// envoyer sa tranche à chacun
for(i=1;i<numprocs;i++)
    MPI_Send(&a[i*tranche][0],tranche*N,
             MPI_CHAR,i,0,MPI_COMM_WORLD);

calculer(a,b,c,tranche);

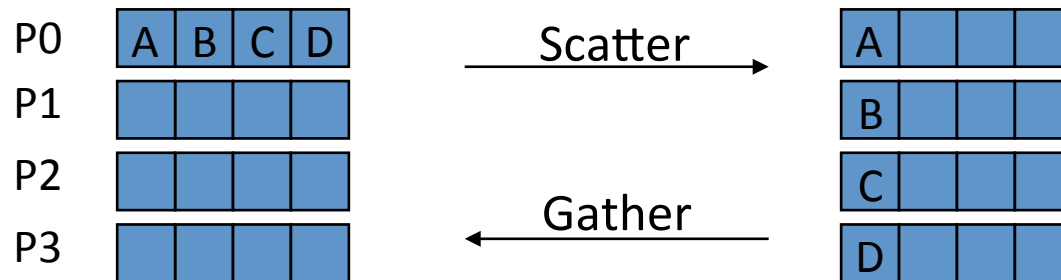
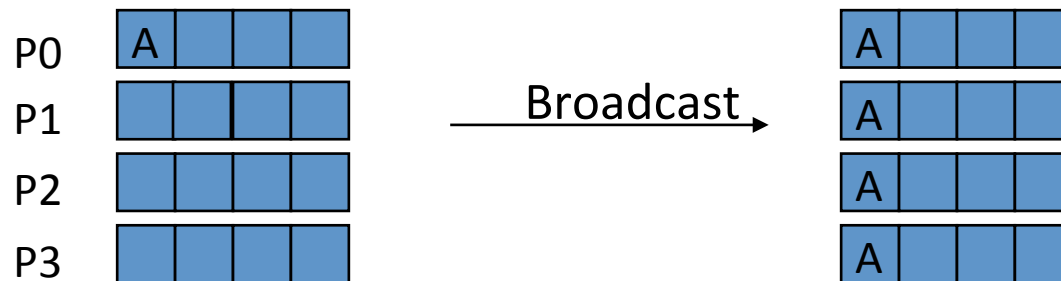
// recevoir la tranche de chacun
for(i=1;i<numprocs;i++)
    MPI_Recv(&c[i*tranche][0],tranche*N,
             MPI_CHAR,i,0,MPI_COMM_WORLD, &status);

    check(a,c);
}
```

Communications collectives

`MPI_Bcast(&buffer, count, datatype, root, comm)`

Bloquant mais non synchronisant



`MPI_Scatter(&sendbuf, sendcnt, sendtype,
&recvbuf, recvcnt, par_processus, recvtype, root, comm)`

Produit de matrices

code du maitre

```
// envoyer la matrice b à tous
```

```
for(i=1;i<numprocs;i++)  
    MPI_Send(b,N*N,MPI_CHAR,  
             i,0,MPI_COMM_WORLD);
```

```
// envoyer sa tranche à chacun
```

```
for(i=1;i<numprocs;i++)  
    MPI_Send(&a[i*tranche][0],tranche*N,  
            MPI_CHAR,i,0,MPI_COMM_WORLD);
```

```
calculer(a,b,c,tranche);
```

```
// recevoir la tranche de chacun
```

```
for(i=1;i<numprocs;i++)  
    MPI_Recv(&c[i*tranche][0],tranche*N,  
            MPI_CHAR,i,0,MPI_COMM_WORLD, &status);
```

```
    check(a,c);
```

```
}
```

```
// envoyer la matrice b à tous
```

```
MPI_Bcast(b,N*N,MPI_CHAR,0,MPI_COMM_WORLD);
```

```
// envoyer sa tranche à chacun
```

```
MPI_Scatter(    a, tranche*N,    MPI_CHAR,  
             MPI_IN_PLACE,  tranche*N, MPI_CHAR,  
             ROOT=0, MPI_COMM_WORLD);
```

```
calculer(a,b,c,tranche);
```

```
// recevoir la tranche de chacun
```

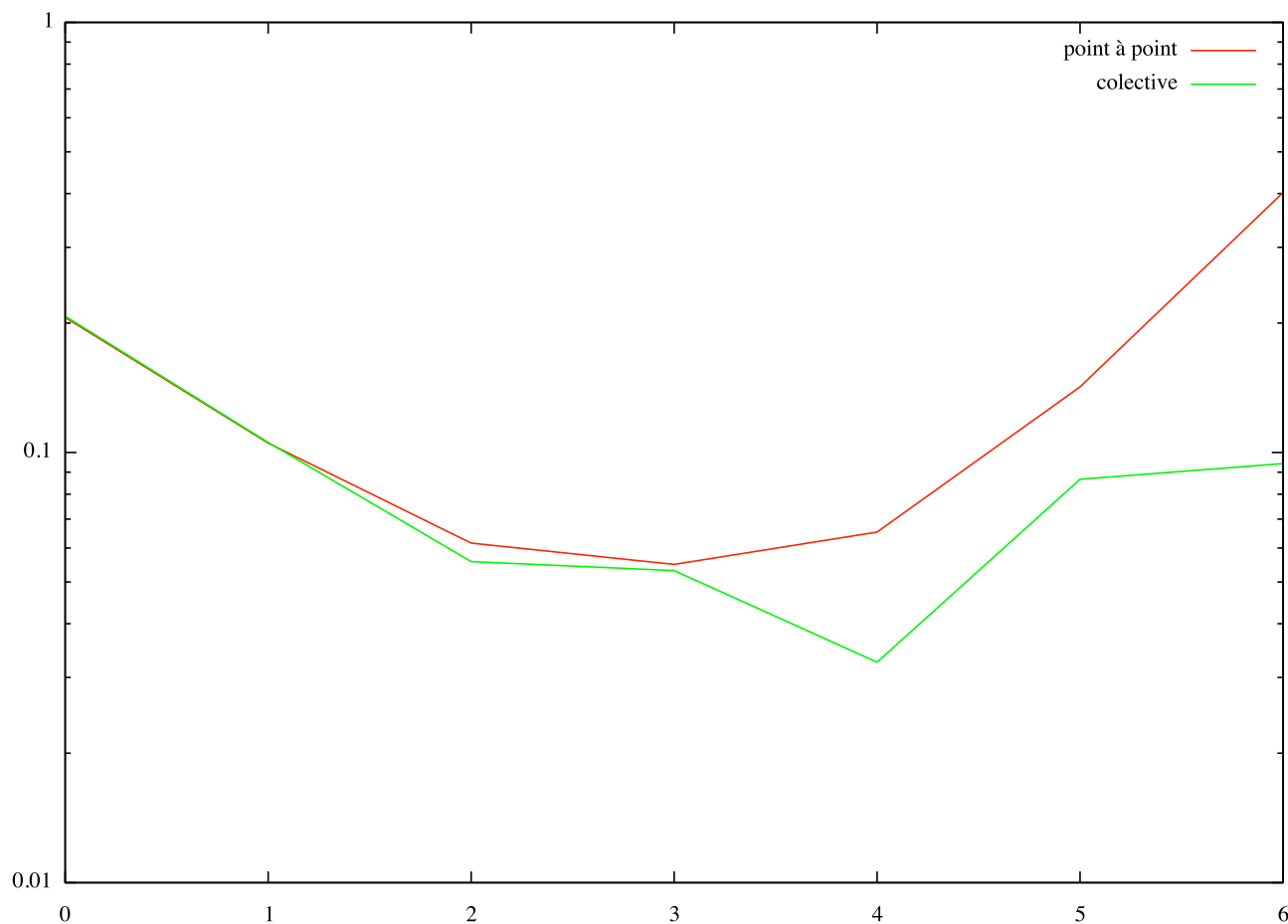
```
MPI_Gather(MPI_IN_PLACE, tranche*N,    MPI_CHAR,  
           c, tranche*N, MPI_CHAR,  
           ROOT=0, MPI_COMM_WORLD);
```

```
    check(a,c);
```

```
}
```


Produit de matrices (1024 char)

abscisse: $\log_2(\text{processus})$



Protocole de diffusion dans un hypercube

Il faut au moins $\log_2(n)$ échanges si on considère des communications point à point : le nombre de sommets informés double au plus à chaque étape.

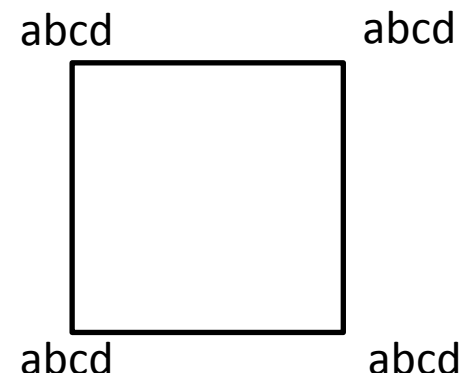
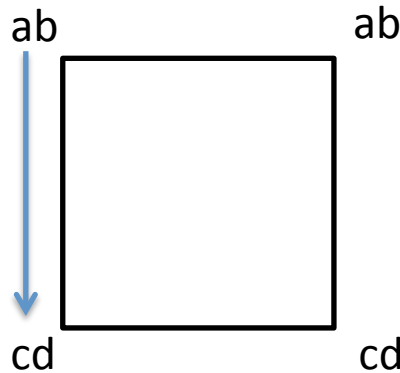
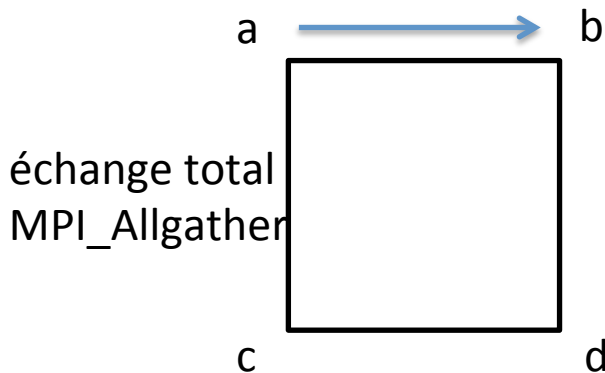
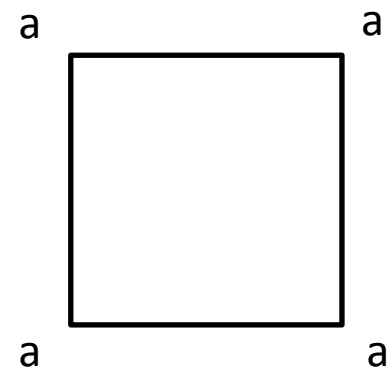
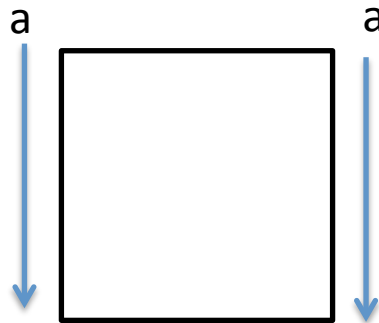
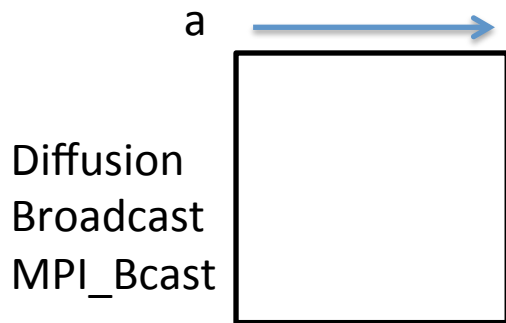


Illustration des opérations collectives

Calculer $\text{out}[i] = f(\text{in}, i)$

```
Int tranche = taille / size ;  
If (rank = 0)  
{  
    int in[taille], out[taille] ;  
    // initialiser in  
  
    for (k = 1; k < size; k++)  
        MPI_Send( in, taille,  
                  MPI_INT, k, TAG, MPI_COMM_WORLD);  
  
    for (k = 1; k < size; k++)  
        MPI_Recv( &out[(k-1) * tranche], tranche,  
                  MPI_INT, k, TAG, MPI_COMM_WORLD, &etat);  
}  
  
else  
{  
    int in[taille], out[tranche];  
  
    MPI_Recv( &in, taille, MPI_INT,  
              0, TAG, MPI_COMM_WORLD, &etat);  
    // Calcul out[i] = f(in,i)  
  
    MPI_Send ( &out, tranche, MPI_INT,  
               0, TAG, MPI_COMM_WORLD);  
}
```

Illustration des opérations collectives

Calculer $\text{out}[i] = f(\text{in}, i)$

```
Int tranche = taille / size ;
```

```
ROOT = 0;
```

```
If (rank = 0)
```

```
{
```

```
int in[taille+tranche], out[taille + tranche] ;
```

```
// initialiser in
```

```
MPI_Bcast(in+tranche,...ROOT...);
```

```
// recevoir dans in
```

```
MPI_Gather (MPI_IN_PLACE,...ROOT, out ...)
```

```
}
```

```
else
```

```
{
```

```
int in[taille], out[tranche];
```

```
// Calcul out[i] = f(in,i)
```

```
MPI_Gather(out,...ROOT...);
```

```
}
```

IN_PLACE car ROOT doit transmettre des données =>
en mode maitre / esclave il faut agrandir le tableau

Illustration des opérations collectives

Itérer x fois $\text{out}[i] = f(\text{in}, i)$

```
int tranche = taille / size ;
If (rank = 0)
{
    int in[taille], out[taille] ;
    // initialiser in
    for (int step = 0; step < x; step++)
    {
        // Envoyer in à tous
        MPI_Bcast(in, tranche, MPI_INT,
                  ROOT=0, TAG, MPI_COMM_WORLD);

        // Calcul out = f(in)

        MPI_Gather( out, tranche, MPI_INT,
                    in, tranche, MPI_INT,
                    ROOT=0, TAG, MPI_COMM_WORLD);
    }
}
```

```
else
{
    int in[taille], out[tranche];

    for (int step = 0; step < x; step++)
    {
        MPI_Bcast( in, tranche, MPI_INT,
                   ROOT=0, TAG, MPI_COMM_WORLD);

        // Calcul out = f(in)

        MPI_Gather ( out, tranche, MPI_INT,
                     NULL, tranche, MPI_INT,
                     ROOT=0, TAG, MPI_COMM_WORLD);
    }
}
```

Le maitre travaille aussi : version SPMD

Illustration des opérations collectives

Itérer x fois $\text{out}[i] = f(\text{in}, i)$

```
int tranche = taille / size ;
If (rank = 0)
{
    int in[taille], out[taille] ;
    // initialiser in
    for (int step = 0; step < x; step++)
    {
        // Envoyer in à tous
        MPI_Bcast(in, tranche, MPI_INT,
                  ROOT=0, TAG, MPI_COMM_WORLD);

        // Calcul out = f(in)

        MPI_Gather( out, tranche, MPI_INT,
                    in, tranche, MPI_INT,
                    ROOT=0, TAG, MPI_COMM_WORLD);
    }
}
```

```
else
{
    int in[taille], out[tranche];

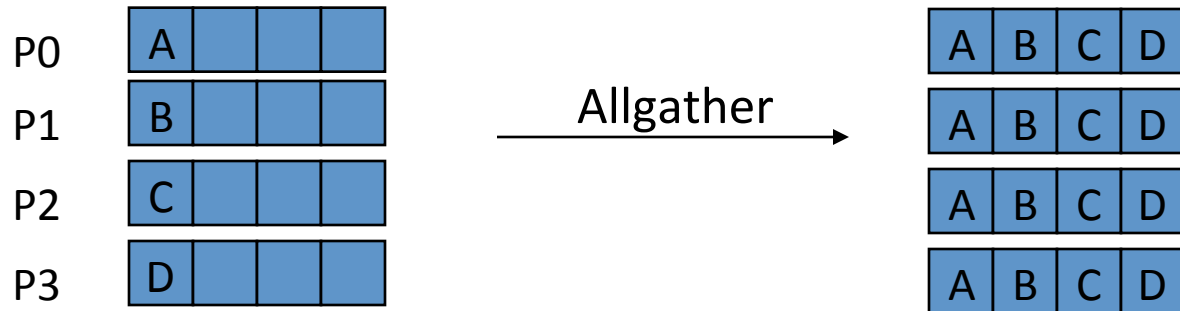
    for (int step = 0; step < x; step++)
    {
        MPI_Bcast( in, tranche, MPI_INT,
                   ROOT=0, TAG, MPI_COMM_WORLD);

        // Calcul out = f(in)

        MPI_Gather ( out, tranche, MPI_INT,
                     NULL, tranche, MPI_INT,
                     ROOT=0, TAG, MPI_COMM_WORLD);
    }
}
```

Solution centralisée qui implique une sérialisation des communications
alors que certaines communications pourraient se faire en parallèle

Communications collectives



Itérer x fois $\text{out}[i] = f(\text{in}[i])$

```
Int tranche = taille / size ;
If (rank = 0)
{
    int in[taille], out[taille] ;
    // initialiser in
    MPI_Bcast(in, tranche, MPI_INT,
              ROOT, TAG, MPI_COMM_WORLD);

    for (int step = 0; step < x; step++)
    {
        // Calcul out = f(in)

        MPI_Allgather(out, tranche, MPI_INT,
                      in, tranche, MPI_INT,
                      0, TAG, MPI_COMM_WORLD);
    }
}
```

```
else
{
    int in[tranche], out[tranche];

    MPI_Bcast( in, tranche, MPI_INT,
              ROOT, TAG, MPI_COMM_WORLD, &etat);

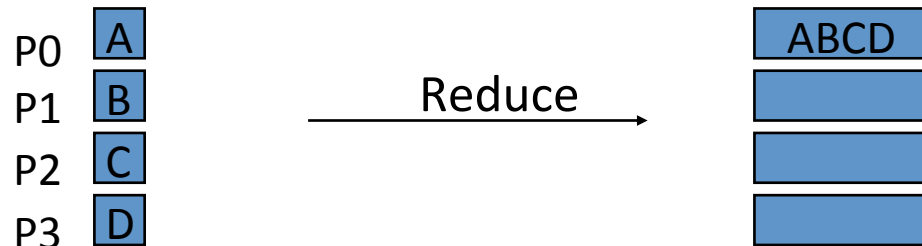
    for (int step = 0; step < x; step++)
    {

        // Calcul out = f(in)

        MPI_Allgather ( &out[(rank-1)*tranche], tranche, MPI_INT,
                      in, tranche, MPI_INT,
                      0, TAG, MPI_COMM_WORLD);
    }
}
```

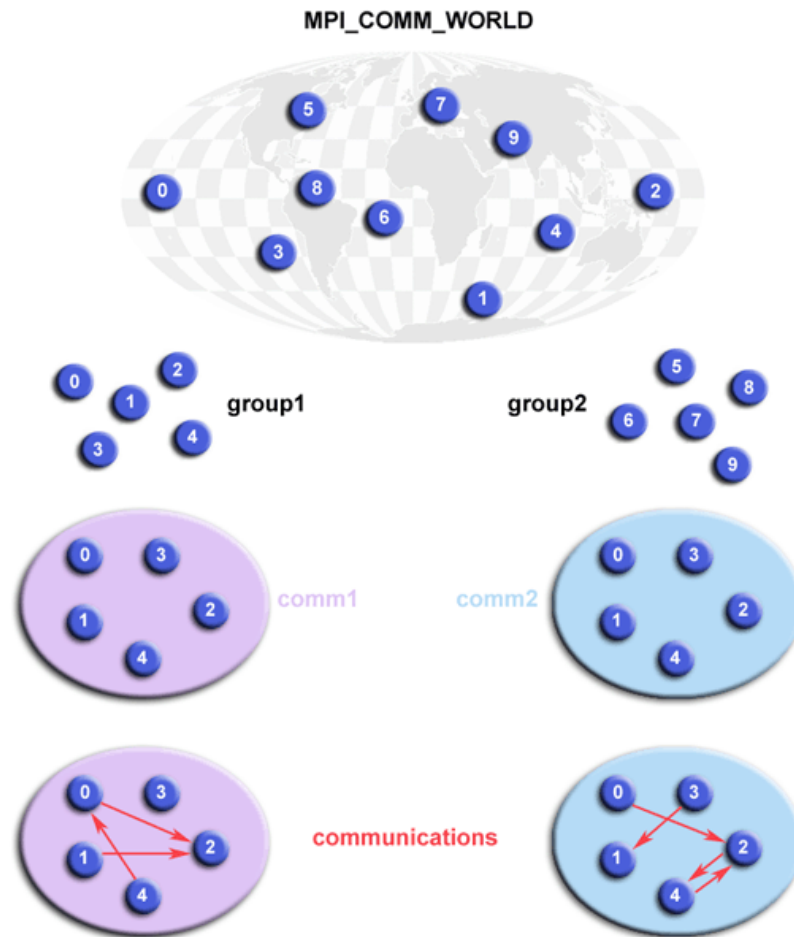

Opérations collectives

- `MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)`
 - `MPI_MAX`, `MIN`, `SUM`, `LAND`, `BAND`, `LOR`, `BOR`, `LXOR`, `BXOR`, `MAXLOC`, `MINLOC`
 - `MPI_Op_create(function, commute, &op)`
 - `MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)`
 - Non déterminisme => au final tous les nœuds n'ont pas forcément les mêmes valeurs



Opérations collectives

notions de groupe et de communicateur



Exemple (LLNL)

```
#include "MPI.h"
#include <stdio.h>
#define NPROCS 8

int main(argc,argv)
int argc;
char *argv[]; {
    int    rank, new_rank, sendbuf, recvbuf, numtasks,
           ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm  new_comm;

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

    if (numtasks != NPROCS) {
        printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
        MPI_Finalize();
        exit(0);
    }
    sendbuf = rank;
```

```
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
    MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
}
else {
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
}

/* Create new new communicator and then perform collective
communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",
rank,new_rank,recvbuf);

MPI_Finalize();
}
```

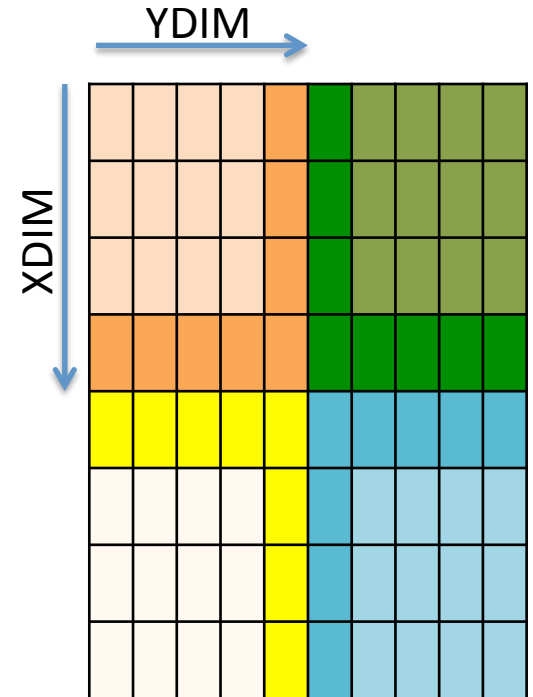
Opérations & communication collectives

Conclusion

- Facilité de codage
 - Programmation SPMD
- Gain de performance
 - Moins de requêtes, plus de parallélisme
 - Transmission d'informations à la bibliothèque
 - Utilisation d'algorithme de diffusion performant
 - Utilisation d'informations topologiques
- À savoir
 - Toutes les opérations collectives sont bloquantes (voir MPI 3)
 - Les processus doivent poster les requêtes dans le même ordre et utiliser le même communicateur
 - Les processus n'exécutent pas *simultanément* les opérations collectives
 - Les opérations collectives ne sont à priori pas synchronisantes
 - Les opérations collective ne sont pas déterministes

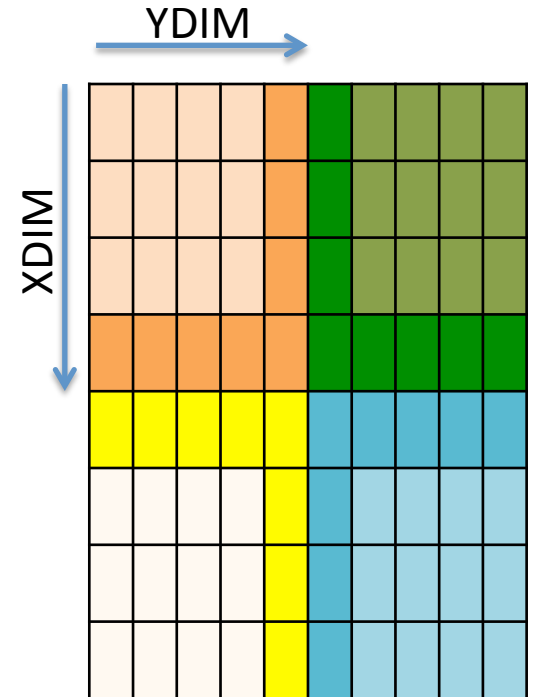
Création de datatypes optimisation / simplification

- Découpe du stencil en pavés
 - Les threads doivent communiquer au Nord Sud Est Ouest
 - Nord / Sud :
Send(zone, YDIM,...
Send(zone[XDIM-1][0],YDIM,...
 - Est / Ouest :
For (i=0; i < XDIM; i++)
Send(&zone[i][YDIM-1],1,
Send(&zone[i][0],1,



Création de datatypes optimisation / simplification

- Découpe du stencil en pavés
 - Les threads doivent communiquer au Nord Sud Est Ouest
 - Nord / Sud :
Send(zone, YDIM,...
Send(zone[XDIM-1][0],YDIM,...
 - Est / Ouest : **création de paquets**
For (i=0; i < XDIM; i++)
Est[i]=zone[i][YDIM-1]
Ouest[i]=zone[i][0]
Send(Est, XDIM,
Send(Ouest, XDIM,



Création de datatypes optimisation / simplification

- Interface pour construire des paquets de données
 - Regrouper de façon automatisée des données
 - Économiser des requêtes
- Empaqueter / dépaqueter des données (héritage de PVM)
 - Type MPI_PACKED et fonction MPI_PACK() / MPI_UNPACK()
 - Recopie dans un buffer
- Définir un type dérivé (datatype) à l'aide d'opérateurs :
 - structures (struct)
 - Juxtaposition d'objets de même datatype (contiguous)
 - Extraction régulière de données d'un datatype (vector)
 - Extraction suivant un tableau d'index certains éléments d'un tableau (indexed)
 - À la iovev

Exemple de construction d'un datatype

Objectif : envoyer les bords d'une macro cellule aux voisins

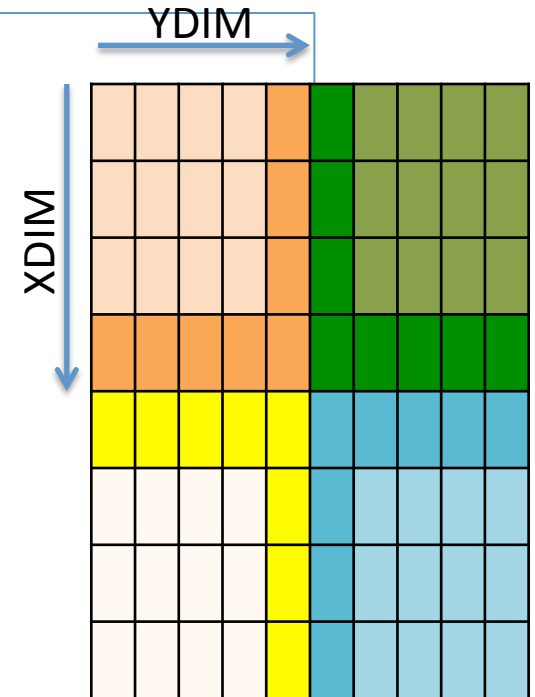
`MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`

Les débuts de blocs sont séparés de *stride* fois la taille de oldtype

```
MPI_Datatype colonne;  
MPI_Type_vector(XDIM, 1, YDIM, MPI_CHAR, &colonne);  
MPI_Type_commit(&colonne);
```

```
MPI_Send(zone, 1, colonne, ...  
MPI_Irecv(zone, 1, colonne, ...
```

- Optimisation dépendant de l'implémentation et du contexte:
 - utilisation d'*iovec* ou de copie
- On peut avoir un datatype différent à la réception
 - On peut émettre en ligne et recevoir en colonne...



MPI et les threads

- Intérêts : l'avenir ?!
 - Réduire la consommation mémoire
 - Réduire la consommation réseau
 - Adapter l'exécution à l'architecture

- Difficultés :
 - Efforts à produire
 - Ratio Processus / Threads à déterminer
 - Empreinte mémoire utilisation du réseau
 - Placement des threads sur les cœurs
 - Utilisation « Probe / malloc / Recv » difficile
 - Deux threads peuvent être en concurrence pour réceptionner un message

- En pratique :
 - `MPI_Init_thread()`

Quatre implémentations possibles

- Single
 - un unique thread
- Funneled
 - seul un thread est autorisé à faire des appels MPI
- Serialized
 - un seul thread à la fois est autorisé à faire des appels MPI
- Multiple
 - vraiment multithreadée

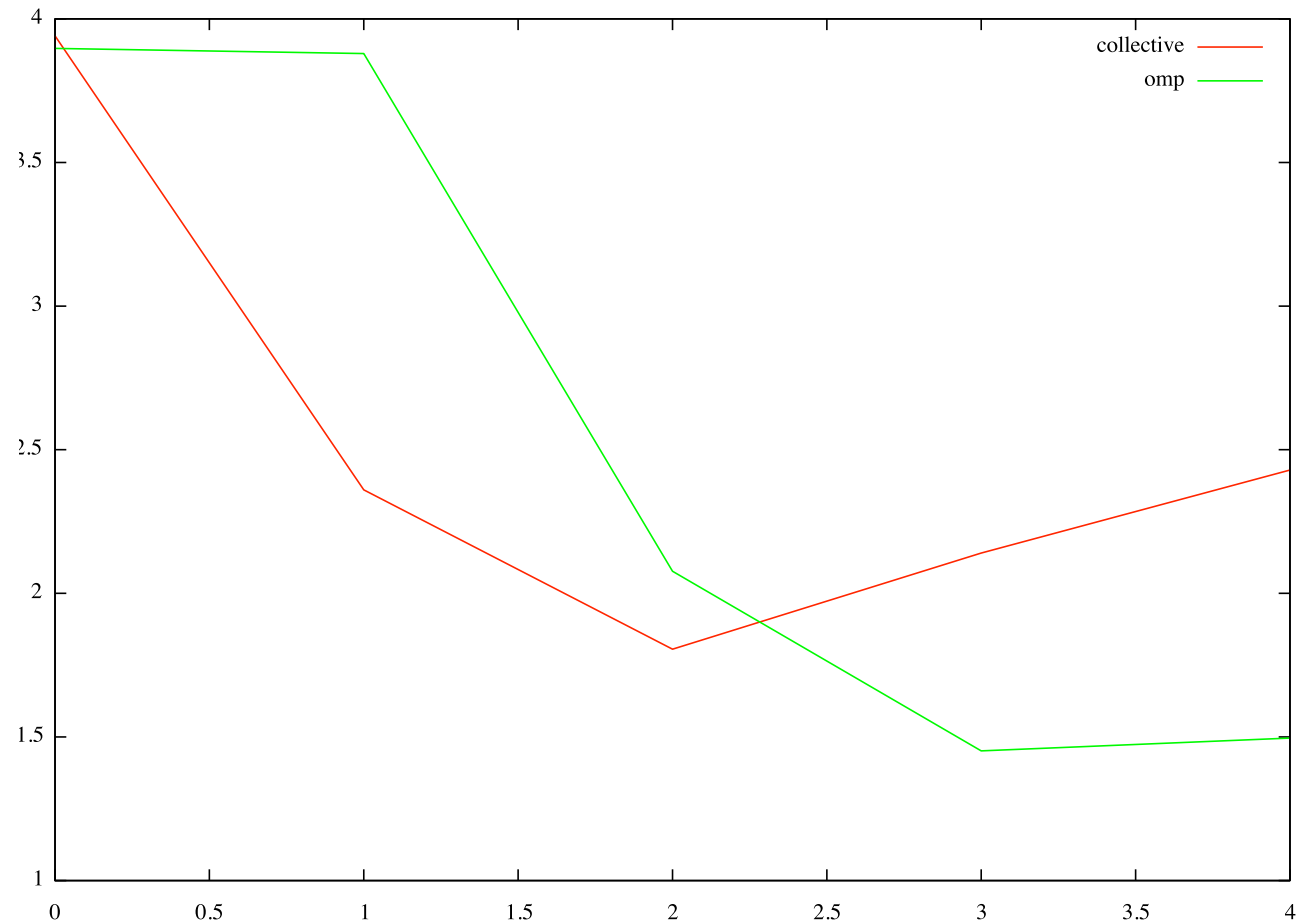
- Voir cours de Programmation hybride MPI-OpenMP IDRIS –CNRS

`MPIcc -fopenmp`
`MPIrun -bynode -np 4 -machinefile fichier
programme`

Comparaison MPI + openmp vs MPI

produit de matrices

4 threads OpenMP



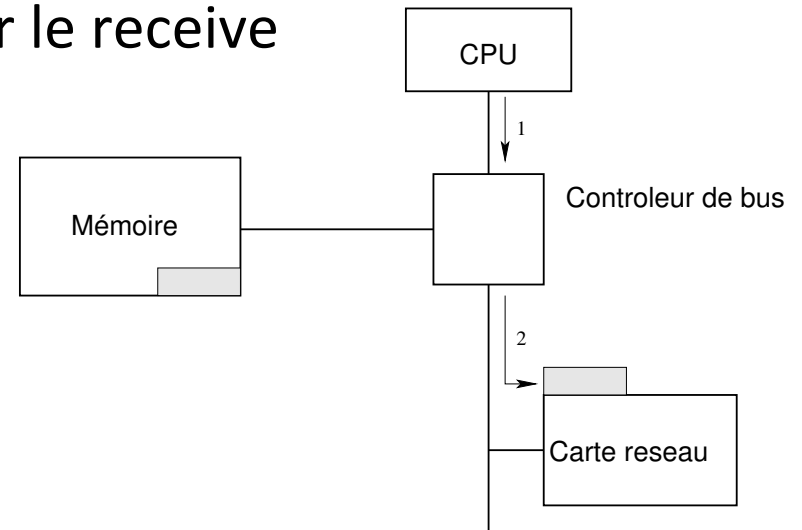
Optimisation des communications

Comment procède MPI ?

- Se passer de TCP/IP dès que possible
- Trois principales techniques de communication :
 - PIO
 - Copie à l'émission comme à la réception
 - DMA + copie
 - Copie à la réception
 - DMA zéro-copie (réseaux rapides)
 - Synchronisation des cartes de communication via un rendez-vous

PIO + copie

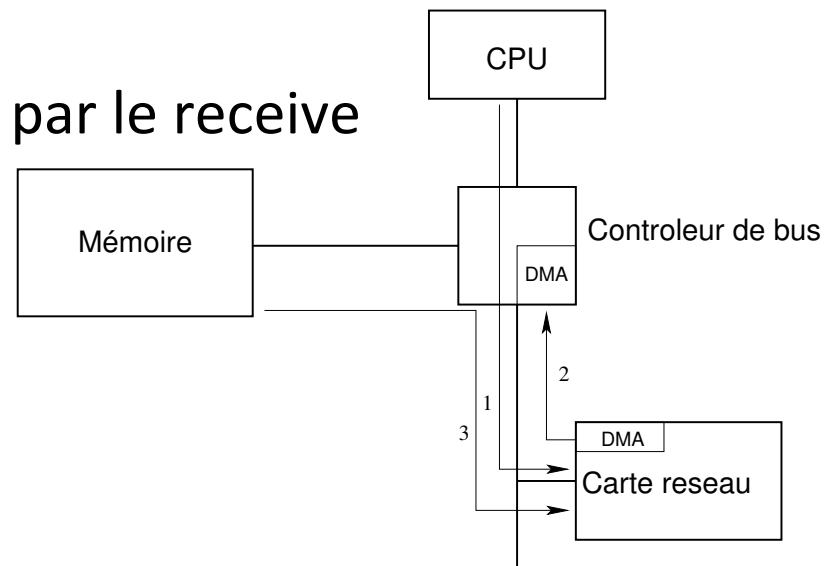
- Programmed input output
 - ① Le processeur émetteur *pousse* les données en copiant directement celles-ci dans la carte
 - ② La carte réceptrice copie le résultat dans un buffer en mémoire
 - ③ Le message sera recopié par le receive



DMA + copie

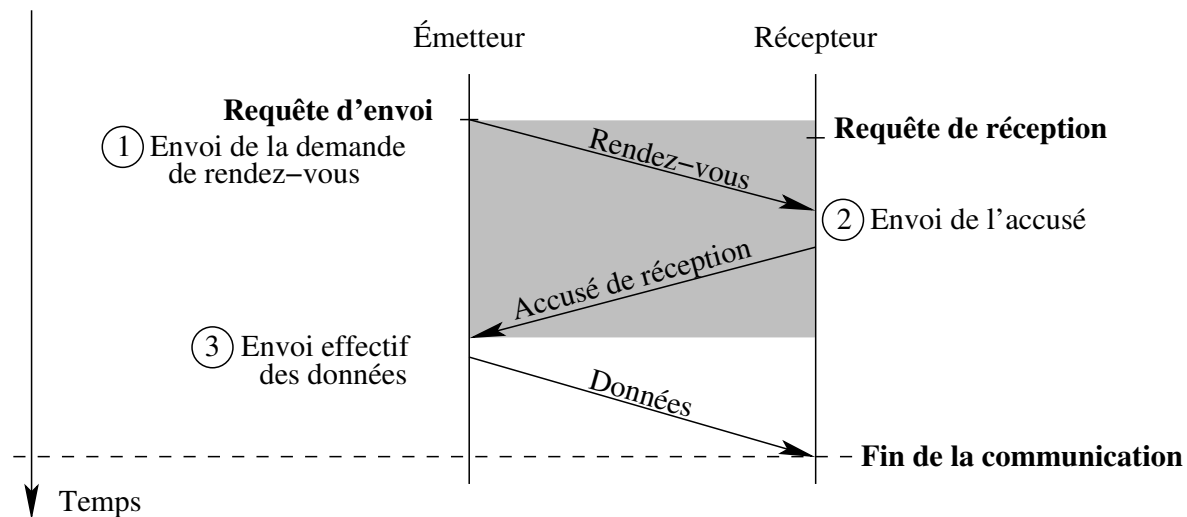
- Direct Memory Access

- ① Le processeur émetteur informe la carte du transfert à réaliser
- ② La carte demande le transfert au contrôleur DMA
- ③ La carte réceptrice copie le résultat dans un buffer en mémoire
- ④ Le message sera recopié par le receive



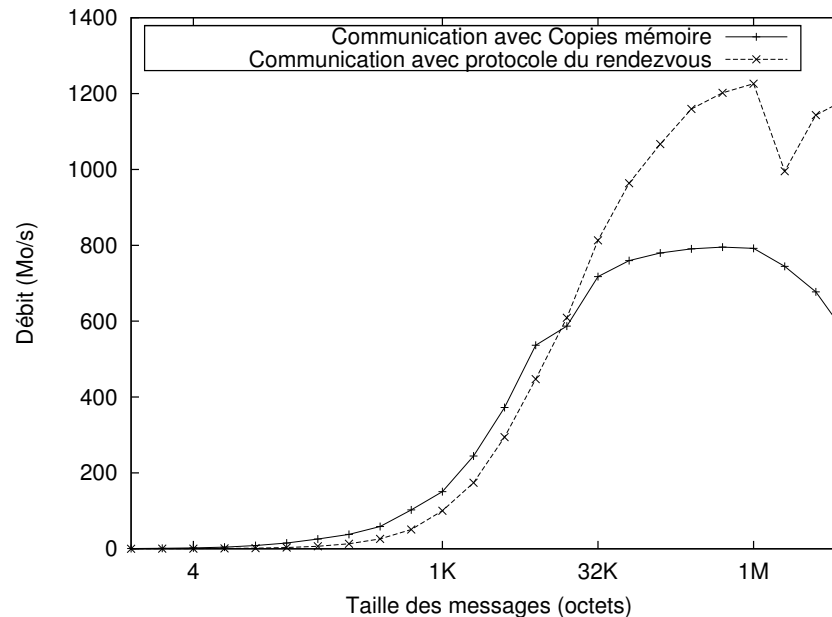
DMA zéro copie réseaux rapides

- Le processeur émetteur/récepteur informe la carte du transfert à réaliser
 - ① La carte émettrice demande un rendez-vous à la carte réceptrice
 - ② La carte réceptrice prépare son DMA puis accorde le rendez-vous
 - ③ La carte émettrice demande le transfert au contrôleur DMA
- Le transfert est réalisé via les buffers internes aux cartes

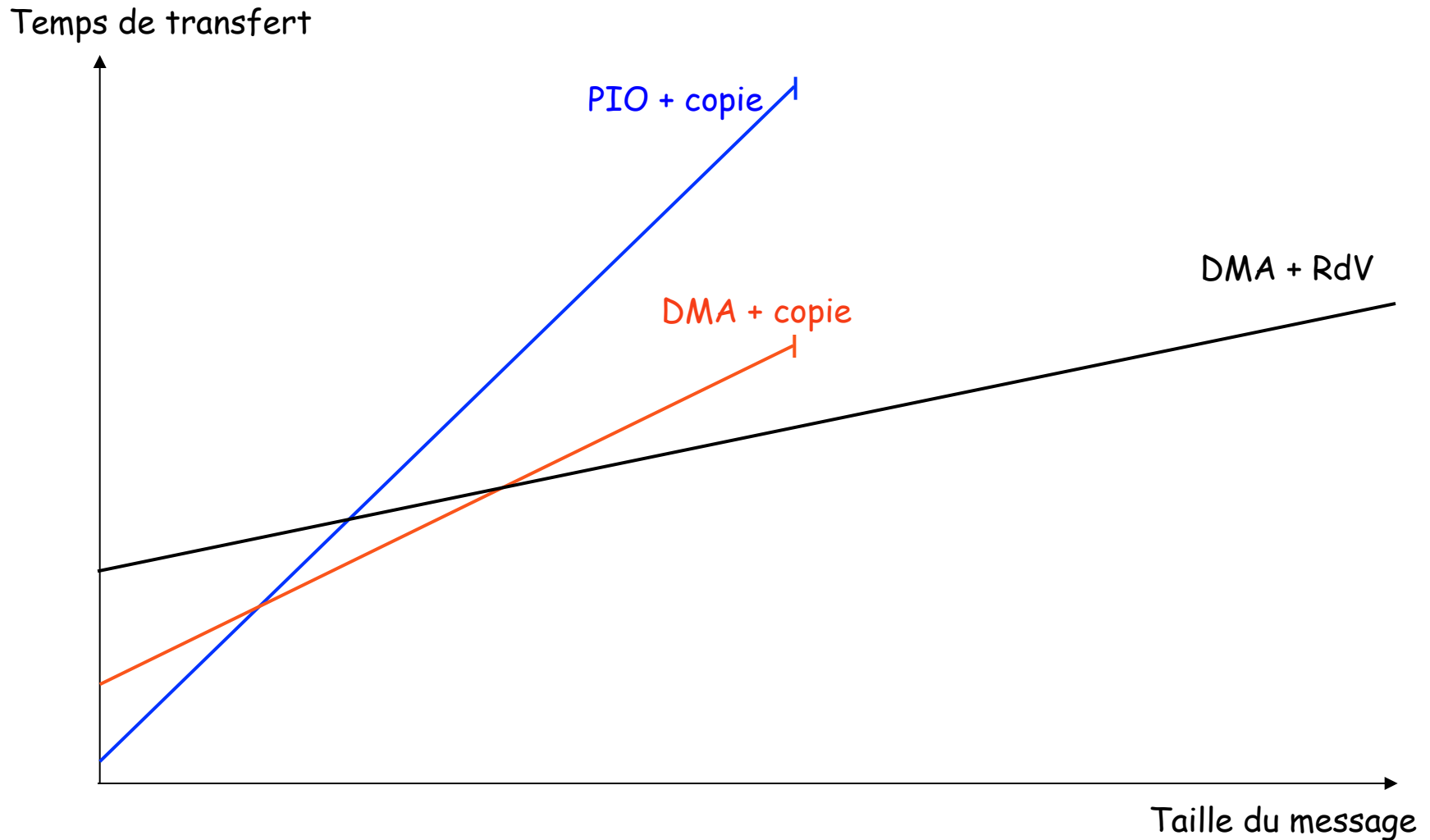


DMA zéro copie

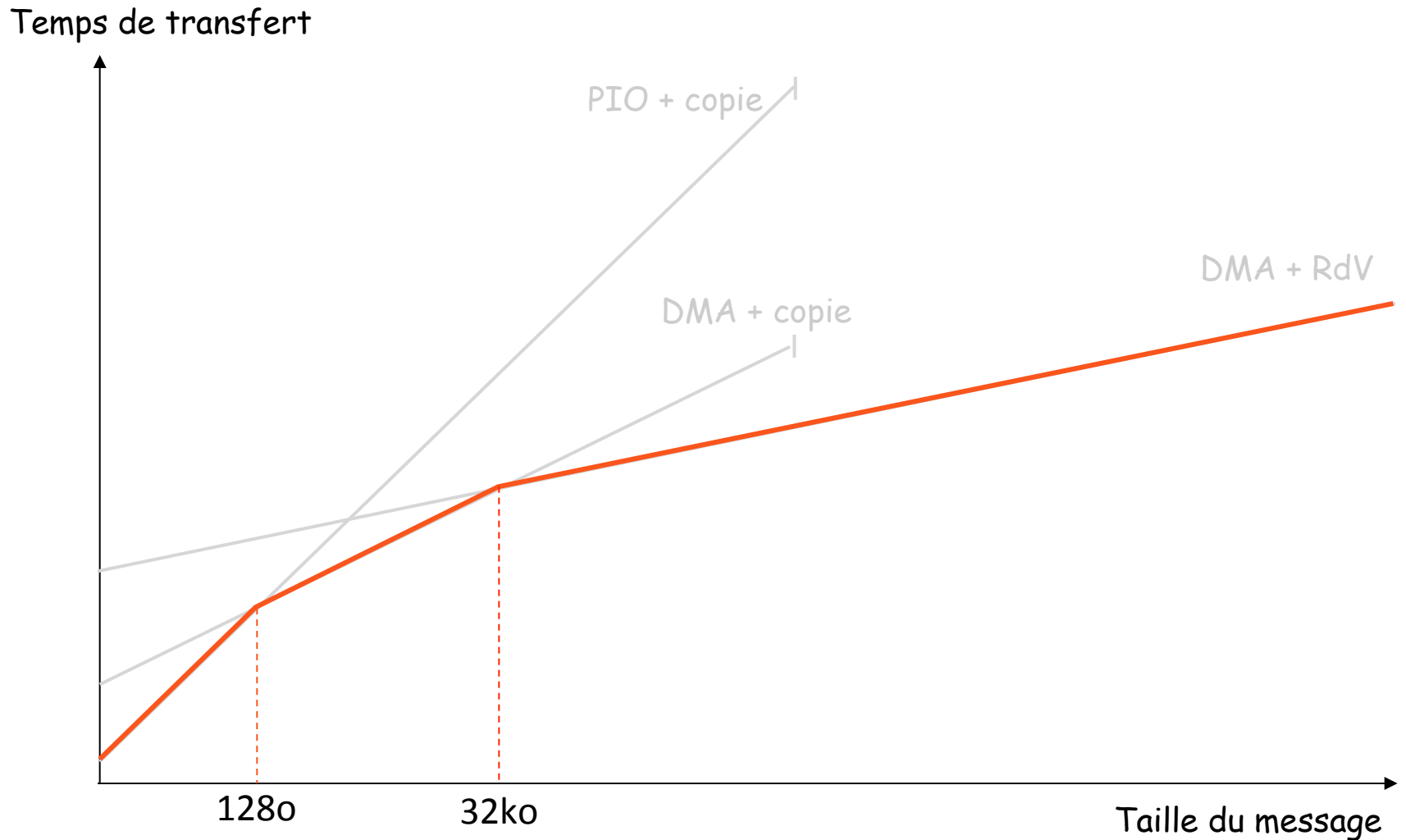
- Le processeur émetteur/récepteur informe la carte du transfert à réaliser
 - ① La carte émettrice demande un rendez-vous à la carte réceptrice
 - ② La carte réceptrice prépare son DMA puis accorde le rendez-vous
 - ③ La carte émettrice demande le transfert au contrôleur DMA
- Le transfert est réalisé via les buffers internes aux cartes



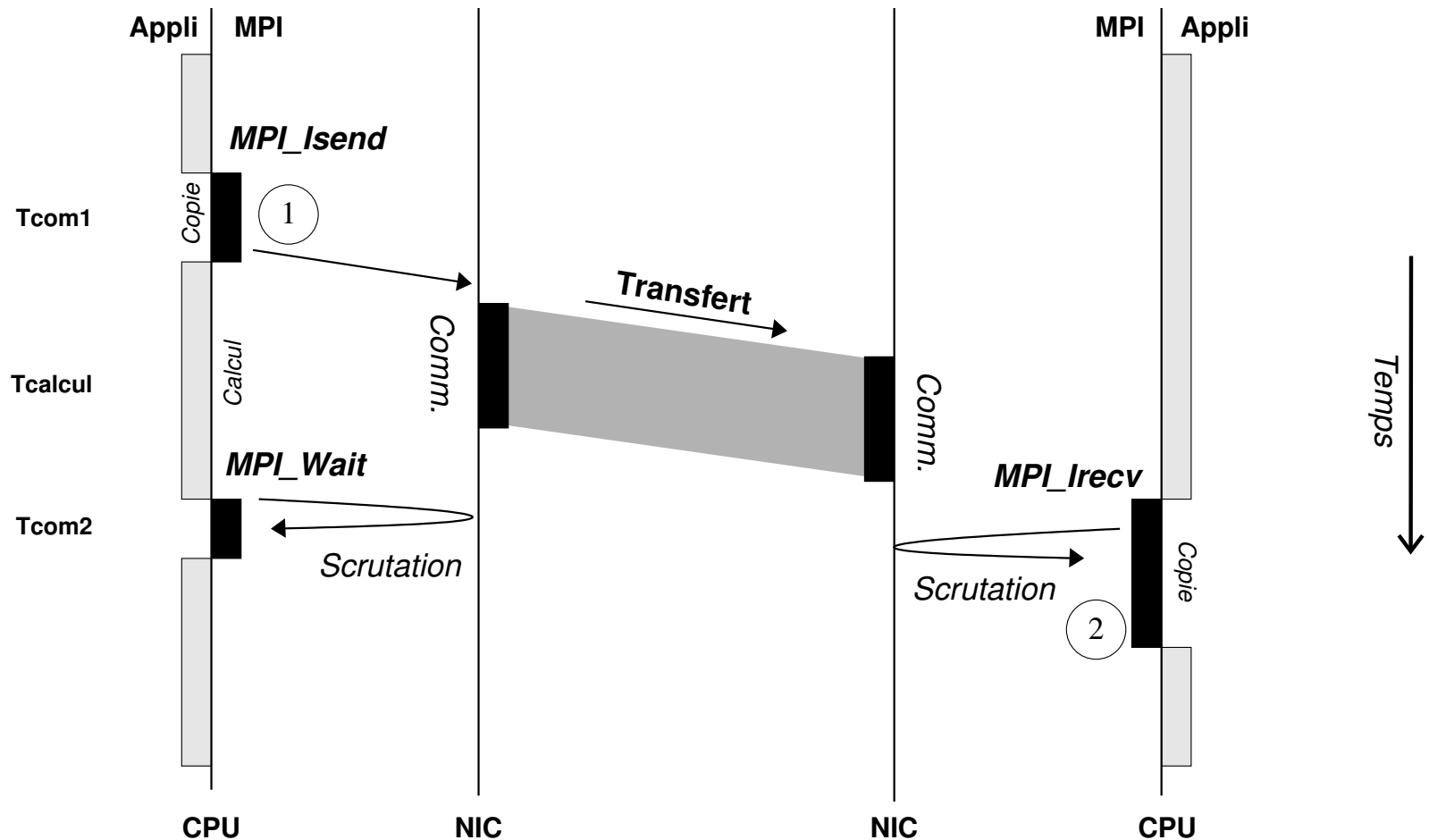
Performances



Performances



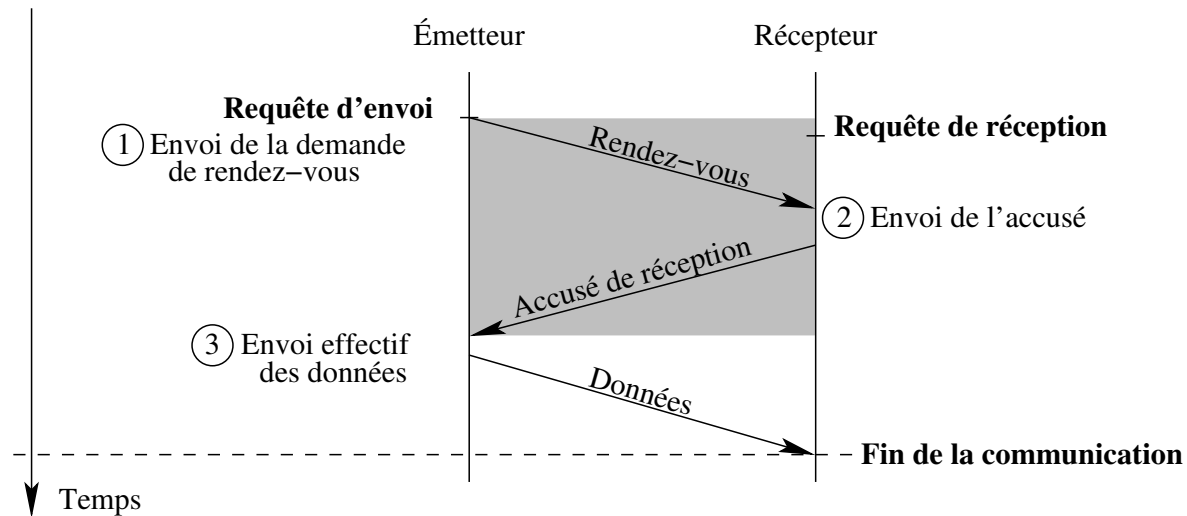
Recouvrement calcul communication comportement avec copie



Recouvrement calcul communication

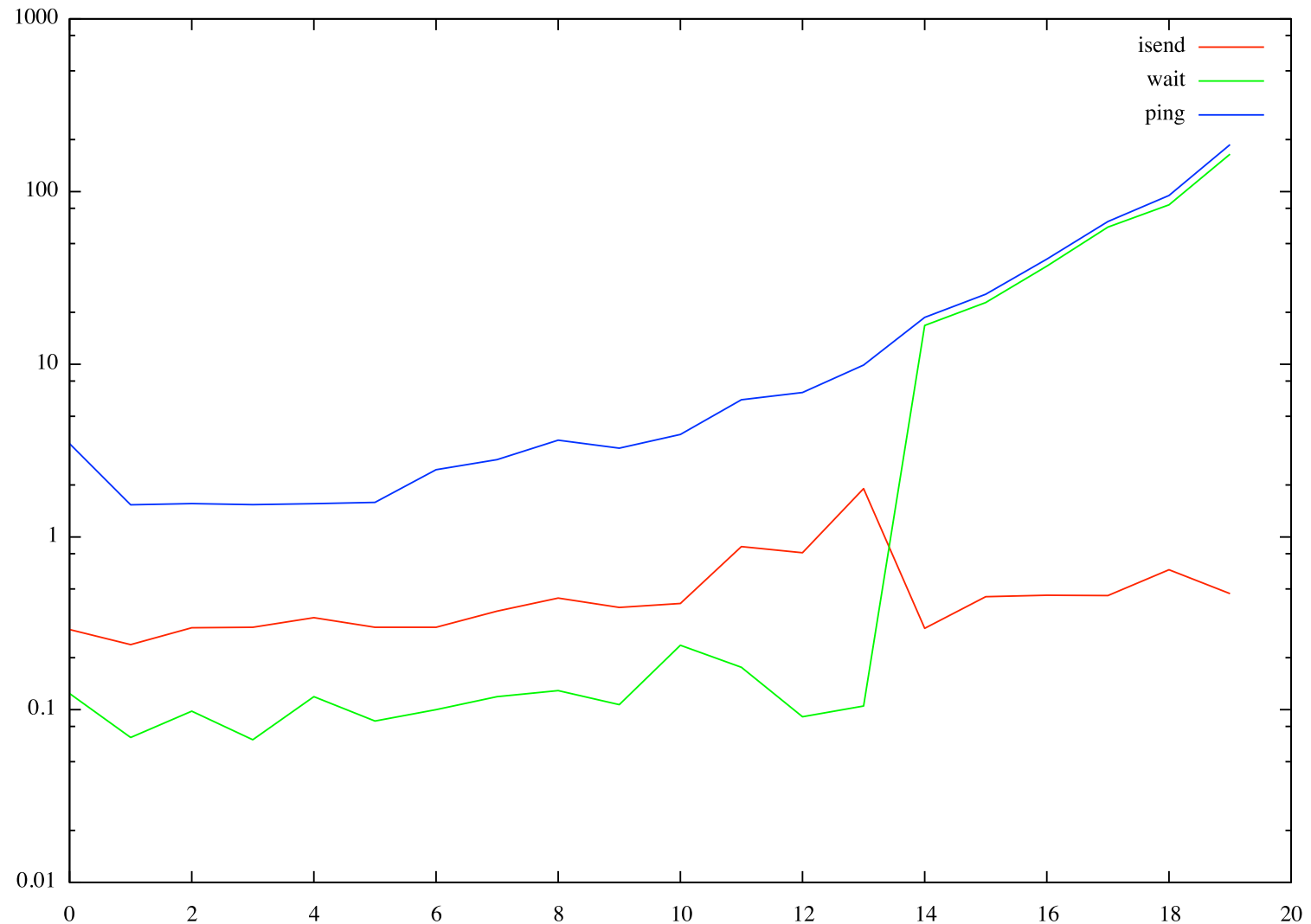
Rappel : DMA zéro copie sur réseau rapide

- Le processeur émetteur/récepteur informe la carte du transfert à réaliser
 - ① La carte émettrice demande un rendez-vous à la carte réceptrice
 - ② La carte réceptrice prépare son DMA puis accorde le rendez-vous
 - ③ La carte émettrice demande le transfert au contrôleur DMA
- Le transfert est réalisé via les buffers internes aux cartes



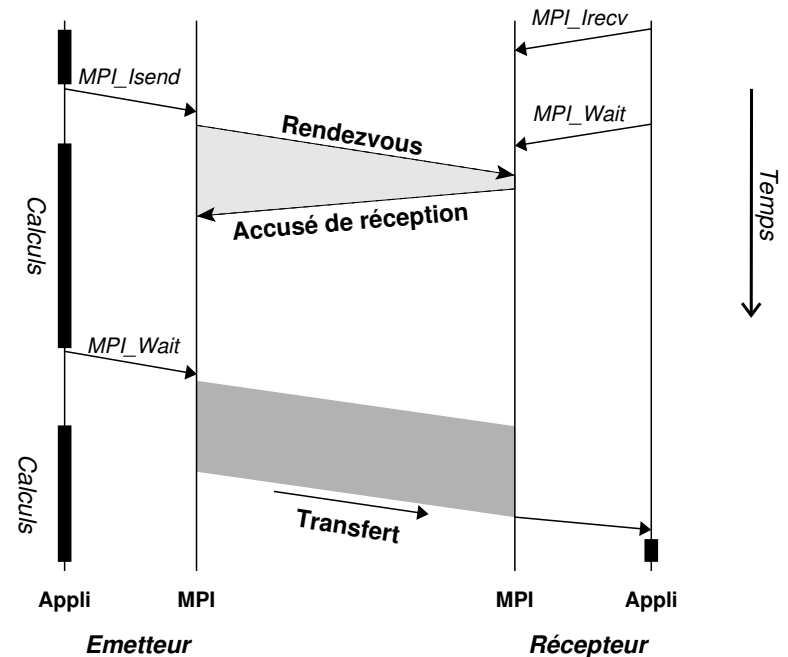
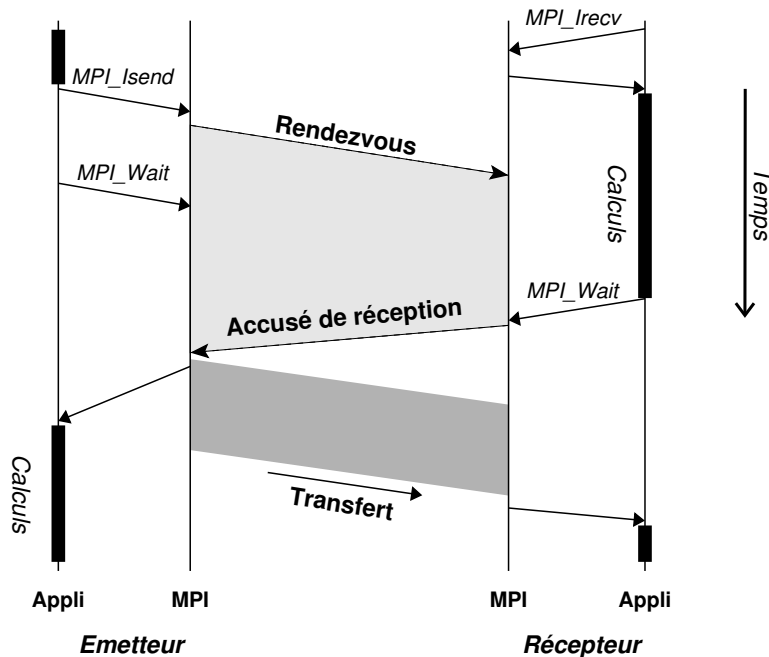
Isend / Wait

le guet-apens du rendez-vous



Recouvrement calcul communication guet-apens du rendez-vous

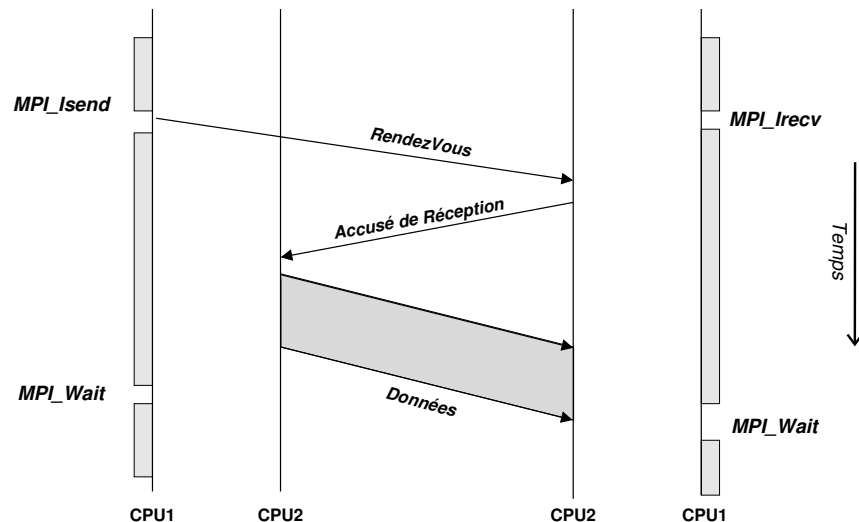
Le transfert effectif n'a lieu que lorsque l'accusé de réception est pris en compte par la couche MPI de l'émetteur.



L'émission est bloquée en attente alors qu'émetteur et récepteur sont prêts :
la bibliothèque n'a pas la main pour faire le transfert !

Solutions au problème du rendez-vous

- Scrutation : donner souvent la main à la bibliothèque
 - Truffer le code d'appel à `MPI_Test()`
 - Interruption : utiliser un thread de progression
 - Se bloque pour attendre une interruption matérielle
 - Pousse le message au réveil
 - Doit être ordonnancé très rapidement (réactivité du S.E.)
- ➔ Surcoûts (changements de contextes et interruptions)

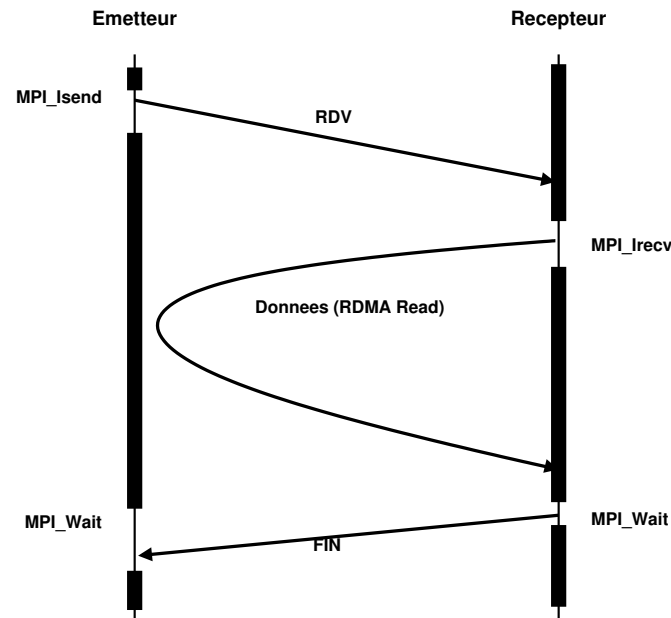


Solutions au problème du rendez-vous

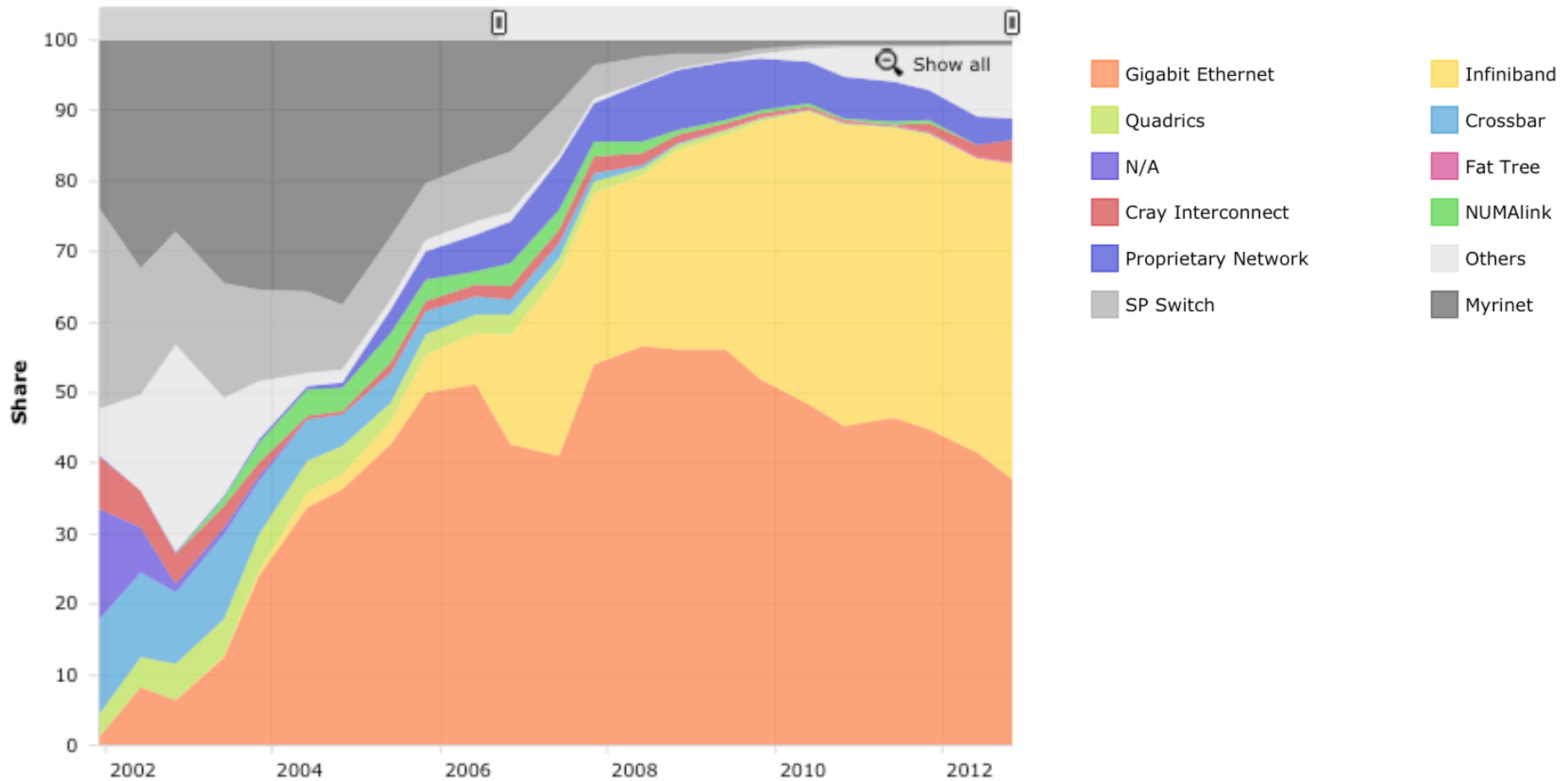
- Scrutation : donner souvent la main à la bibliothèque
- Interruption : utiliser un thread de progression
 - Réactivité (S.E.)
 - Surcoûts
- « Attente mixte »
 - Scrutation pendant un certain temps puis appel à un thread de progression
 - À la manière des *spinlocks*

Solutions au problème du rendez-vous

- Scrutation : donner souvent la main à la bibliothèque
- Interruption : utiliser un thread de progression
- Attente mixte
- Technique du *Remote DMA* (MPI 2)
 - Lecture / écriture à distance dans la mémoire
 - Mémoire commune
 - Infiniband



Top 500 Réseaux



Et bien encore plus...

MPI 2

- MPI 2 permet de
 - Mettre en œuvre une topologie virtuelle
 - Ex. cartesian
 - Permet au support d'exécution de projeter efficacement l'application sur la plateforme
 - Faire des entrées/sorties parallèles
 - Accéder à des mémoires distantes
 - *One sided communication* : get, put, accumulate
 - Efficace sur architecture à mémoire partagée et sur un réseau infiniband
 - Créer dynamiquement des processus
 - Intercommunicateur : grappe de grappes, grilles

Et bien encore plus...

MPI 3

- Non blocking and Neighborhood Collectives
- Matched Probe
- MPI Tool interface
- New One Sided Functions and Semantics
- New Communicator Creation Functions
- Improvements in Language Bindings
- Fault Tolerance/Resiliency