

CentraleSupélec

Mineure HPC-SBD

# L'essentiel de MPI-1

Stéphane Vialle

Stephane.Vialle@centralesupelec.fr  
http://www.metz.supelec.fr/~vialle

CentraleSupélec

## L'essentiel de MPI-1

- 1 – Principes de base
- 2 – Communications Pt-à-Pt bloquantes
- 3 – Exemple de programmation MPI (Jacobi)
- 4 – Communications de groupe
- 5 – Stratégie de déploiement
- 6 – Analyse de performances en MPI
- 7 – TP : produit de matrices denses sur un anneau de processeurs

CentraleSupélec

## L'essentiel de MPI-1

### 1 – Principes de base

CentraleSupélec

### Principes de bases

#### Fait pour des archi. distribuées (ou pas)

L'utilisateur « voit » un ensemble de processus coopératifs

Déploiement et exécution

Processus 0, Processus 1, Processus 2, Processus 3

Serveur multi-cœurs  
Architecture à mémoire partagée

Cluster de serveurs  
Architecture à mémoire distribuée

CentraleSupélec

### Principes de bases

#### « Hello World »

Programme source :

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv) {
    int Me, NbPE;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NbPE);
    MPI_Comm_rank(MPI_COMM_WORLD, &Me);
    printf("Hello World from process %d/%d\n", Me, NbPE);
    fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Le « groupe » de tous les process du pgm MPI

Pour afficher tous les msgs avant la fin du pgm

Exemple d'exécution avec 3 processus :

```
Hello World from process 0/3
Hello World from process 2/3
Hello World from process 1/3
```

Pas d'hypothèse sur l'ordre d'exécution ni d'affichage !

CentraleSupélec

### Principes de bases

#### Principales instructions d'un code MPI

Instructions **incontournables** d'un programme MPI :

Inclusion du fichier d'entête MPI

```
#include "mpi.h"
```

Première instruction MPI du main (int argc, char \*\*argv)

```
MPI_Init(&argc, &argv);
```

Pour connaître le nombre de processus lancés

```
MPI_Comm_size(MPI_COMM_WORLD, &NbPE);
```

Pour connaître le numéro du processus

```
MPI_Comm_rank(MPI_COMM_WORLD, &Me);
```

Dernière instruction MPI du main

```
MPI_Finalize();
```


Instructions de **communication** entre processus :

Comms. point-à-point	Comms. de groupe	Le parallélisme est très explicite en MPI !
Ex : ... MPI_Send(...); MPI_Recv(...);	Ex : ... MPI_Bcast(...); ...	

Principes de bases  
**Difficultés de développement en MPI**

**Objectifs :**  
Concevoir un code MPI qui fonctionne (!) et qui soit efficace (!)

**Difficultés :**

- **Deux processus ne possèdent pas d'espace mémoire partagé**  
Accéder à des données stockés/générées par un autre processus impose des **échanges de messages** avec lui
- **Concevoir des schémas d'échange de données entre les processus**  
Ex : **topologie virtuelle** en anneau de processus   
topologie virtuelle en tore 2D de processus .....
- **Fonctionner avec un nombre de processus le moins contraint possible**  
Ex : topologie virtuelle en anneau (1D)  
- fonctionner pour 1, 2, 3, 4, 5 ... processus : **TB**  
- fonctionner pour 2, 4, 6, 8... processus : **inconfortable**  
Fonctionner aussi avec un seul processus (très pratique!)

Principes de bases  
**Difficultés de développement en MPI**

**Objectifs :**  
Concevoir un code qui fonctionne (!) et qui soit efficace (!)

**Difficultés :**

- **Eviter les inter-blocages**  
Ex : tous les processus bloqués en attente de réception de données  
→ **Séquencer/Planifier** les opérations **Send et Recv**
- **Diminuer l'impact des temps de latence**  
Envoyer 1 msg de 1000 données, plutôt que 1000 msg de 1 donnée.  
→ Sur chaque processus **regrouper les communications** à destination de la même cible
- **Diminuer l'impact des temps de communication**  
Réaliser les communications dans des threads en parallèle des calculs (*la carte réseau n'utilise pas le CPU*)  
→ **Recouvrir les communications par les calculs**

Principes de bases  
**Compilation d'une application MPI**

**Compilation MPI :**

- **MPI est perçu comme une simple bibliothèque**  

```
cc -I.../include -L.../libs  
-O3 -o myAppli XXX.c YYY.c ...  
-lmpi
```

  
ou bien :  

```
mpicc -O3 -o myAppli XXX.c YYY.c ...
```

  
→ On obtient **un seul fichier exécutable** (comme en séquentiel)
- **MPI est compatible avec le multithreading**  
- Notamment compatible avec OpenMP (`mpicc -O3 -fopenmp ...`)  
- Si les appels à MPI sont toujours faits par un seul thread à la fois (*pas de parallélisation des appels*)  
ALORS : aucun problème (*prêt à l'emploi*)  
SINON : installer MPI en mode multithreads/thread safe.

Principes de bases  
**Déploiement et exécution d'une appli. MPI**

**Déploiement de l'application :**  
déployer  $p$  processus sur  $m$  machines, et exécuter l'ensemble des processus :  
→ **tout se fait avec la commande « mpirun »**

<code>mpirun -np &lt;#P&gt;</code>	Nb total de processus à créer
<code>-machinefile &lt;FileName&gt;</code>	Liste des machines disponibles
<code>-map-by ... -rank-by ... -bind-to ...</code>	Contrôle du déploiement ( <b>voir plus loin - important</b> )
<code>&lt;Path/ExecName&gt; [args]</code>	Exécutable et ses arguments

Ex minimaliste : `mpirun -np 3 ./HelloWorld`  
→ lance 3 processus "HelloWorld" sur la machine courante

Principes de bases  
**Démarche complète de parallélisation MPI**

1. **Algorithmique parallèle** (optimisée)
2. **Programmation parallèle** (optimisée)  
→ code src avec : **envoi de message** + **multithreading** + **vectorisation**  
*ex : MPI + OpenMP + noyaux de calculs vectorisés*
3. **Compilation**  
→ Production d'UN binaire (`mpicc`)
4. **Stratégie de déploiement**  
→ étude des caractéristiques des machines disponibles  
→ choix d'options de déploiement (`-map-by / -rank-by / -bind-to`)
5. **Déploiement et exécution**  
→ Copie du binaire sur chaque nœud ou montage d'un répertoire partagé  
→ Lancement du programme MPI suivant le déploiement prévu  
`mpirun -np <#P> -machinefile machines.txt .....`

L'essentiel de MPI-1

## 2 – Communications point à point

- Communications disponibles
- Principes du Bsend/Recv
- Principes du Ssend/Recv
- Principes du Isend/Irecv
- Principes du Sendrecv
- Principes du Sendrecv\_replace

Communications point-à-point

## Communications disponibles

Communications point-à-point disponibles :

Mode/Type	Not specified	Buffered	Synchronous	Ready
Bloquantes	MPI_Send MPI_Recv	MPI_Bsend MPI_Recv	MPI_Ssend MPI_Recv	MPI_Rsend MPI_Recv
Non - bloquantes	MPI_Ibsend MPI_Irecv	MPI_Ibsend MPI_Irecv	MPI_Issend MPI_Irecv	MPI_Irsend MPI_Irecv

+

MPI_Sendrecv MPI_Sendrecv_replace
--------------------------------------

Communications pt-à-pt combinées et bloquantes

Communications point-à-point

## Communications disponibles

Les types des données sont spécifiés dans les communications :

Ex : MPI\_Send(..., n, MPI\_DOUBLE, ...);

Type pré-existants :

MPI_CHAR	MPI_SHORT	MPI_FLOAT	MPI_UNSIGNED_CHAR
MPI_BYTE	MPI_INT	MPI_DOUBLE	MPI_UNSIGNED_SHORT
	MPI_LONG	MPI_LONG_DOUBLE	MPI_UNSIGNED_LONG

Utilisé en TP

Définition possible de nouveaux types :

- des **tableaux** à une dimension :  
MPI\_Type\_contiguous(count, old\_type, new\_type)
- des **vecteurs extraits de tableaux** plus complexes :  
MPI\_Type\_vector(count, blocklen, stride, old\_t, new\_t)
- des **structures** de données :  
MPI\_Type\_struct(count, blocklens, indices, old\_t, new\_t)

Puis allocation du nouveau type avant usage :  
MPI\_Type\_commit(datatype)

Communications point-à-point

## Principes du Bsend/Recv

Echanges de messages en mode *bloquant* et *bufferisé* : Bsend/Recv

**Bsend(...)** :

- rend la main dès que les données à expédier sont sauvegardées,
- on peut modifier sans délai les zones de données émises,
- la transmission réelle se fera plus tard (quand le dest. le demandera)

**Recv(...)** :

- demande la transmission et la réception des données
- rend la main seulement lorsque les données sont toutes arrivées
- les réceptions de données servent aussi à resynchroniser les processus

Ex. sur un anneau de processus :

Même code dans tous les processus :

```
.....
Bsend(Tab, Me+1);
Recv(Tab, Me-1);
.....
```

Communications point-à-point

## Principes du Bsend/Recv

Echanges de messages en mode *bloquant* et *bufferisé* : Bsend/Recv

**Bsend(...)/Recv(...)** :

Quelle que soit la topologie des processus (anneau, tore, hypercube...)  
→ un même code de communication pour tous les processus

- Exécuter tous les **Bsend(...)** de l'étape (dans n'importe quel ordre)
- Exécuter tous les **Recv(...)** de l'étape (dans n'importe quel ordre)

Ex. sur un anneau de processus :

Même code dans tous les processus :

```
.....
Bsend(Tab, Me+1);
Recv(Tab, Me-1);
.....
```

Communications point-à-point

## Principes du Bsend/Recv

Mode *bloquant* et *bufferisé* : MPI\_Bsend / MPI\_Recv

- Bsend** : copie les données dans le buffer et rend la main
- Recv** : se signale à l'émetteur, provoque le transfert des données, rend la main à la fin du transfert

```
MPI_Bsend(data_adr, count, datatype, destproc, tag, comm)
MPI_Recv(data_adr, count, datatype, srcproc, tag, comm, stts_adr)
```

Mode menant à des parallélisations simples à concevoir.  
Parfois les plus efficaces, malgré les temps de recopie !

Communications point-à-point

## Principes du Bsend/Recv

Mode *bloquant* et *bufferisé* : MPI\_Bsend / MPI\_Recv

Le développeur doit *allouer, attacher, détacher* et *libérer* son buffer :

```
..... // Buffer size comput
MPI_Pack_size(n, MPI_DOUBLE, MPI_COMM_WORLD, &sizeMsg);
sizeBuff = m*(sizeMsg + MPI_BSEND_OVERHEAD);
ptBuff = (double *) malloc(sizeBuff); // Buffer allocation
MPI_Buffer_attach(ptBuff, sizeBuff); // Buffer attachement
for (i=0; i<m; i++) { // Message sending
  tab = ...; // - tab update
  MPI_Bsend(tab, n, MPI_DOUBLE, suivant, ); // - msg passing
  MPI_Recv(tab, n, MPI_DOUBLE, precedent, );
}
MPI_Buffer_detach(&ptBuff, &sizeBuff); // Buffer detachment
free(ptBuff); // Buffer dealloc.
```

Bloque jusqu'à ce que tous les msg stockés dans le buffer soient délivrés

Un overhead par msg stocké dans le buffer

Communications point-à-point

## Principes du Bsend/Recv

Mode *bloquant et bufferisé* : MPI\_Bsend / MPI\_Recv

Le développeur doit *allouer, attacher, détacher* et *libérer* son buffer :

```

// Buffer size comput
MPI_Pack_size(n, MPI_DOUBLE, MPI_COMM_WORLD, &size1Msg);
sizeBuff = 1*(size1Msg + MPI_BSEND_OVERHEAD);
ptBuff = (double *) malloc(sizeBuff); // Buffer allocation
for (i=0; i<m; i++) { // Message sending
  MPI_Buffer_attach(ptBuff, sizeBuff); // - attachement
  tab = ...; // - tab update
  MPI_Bsend(tab, n, MPI_DOUBLE, suivant, ...); // - msg passing
  MPI_Recv(tab, n, MPI_DOUBLE, precedent, ...); // - detachement
  MPI_Buffer_detach(&ptBuff, &sizeBuff); // - detachement
}
free(ptBuff); // Buffer dealloc.

```

Bloque jusqu'à ce que tous les mgs stockés dans le buffer soient délivrés

Un overhead par msg stocké dans le buffer

Communications point-à-point

## Principes du Ssend/Recv

Mode *bloquant et synchrone* : Ssend/Recv

**Ssend(...)** :

- rend la main seulement lorsque les données sont toutes parties
- on ne doit PAS modifier les zones de données en cours d'émission

**Recv(...)** :

- demande la transmission et la réception des données
- rend la main seulement lorsque les données sont toutes arrivées

→ A chaque étape un Recv(...) doit correspondre à un Ssend(...)

Car les communications forment des *Rendez-Vous* entre processus →

Certains buffers sont encore nécessaires pour éviter d'écraser des données

Communications point-à-point

## Principes du Ssend/Recv

Mode *bloquant et synchrone* : Ssend/Recv

**Ssend(...)/Recv(...)** :

Quelle que soit la topologie des processus (anneau, tore, hypercube...)

→ il faut planifier précisément toute la séquence des Ssend et des Recv sur chaque processus (le *plan de communication*)

Ex. sur un anneau de processus :

```

if (processId % 2 == 0)
  1 : Ssend(Tab, Me+1);
  2 : Recv(Tab, Me-1);
else
  1 : Recv(buffer, Me-1);
  2 : Ssend(Tab, Me+1);
  3 : permut(buffer, Tab);

```

Le plan de communication sera plus long qu'avec Bsend(...)/Recv(...)

Ex : exécution de la moitié des comms (1), puis de l'autre moitié (2)

Communications point-à-point

## Principes du Ssend/Recv

Mode *bloquant et synchrone* : MPI\_Ssend / MPI\_Recv

- Ssend** : se signale au récepteur, attend son ack., puis ne rend la main qu'à la fin de l'émission des données
- Recv** : répond à l'émetteur et provoque le transfert, puis rend la main à la fin de la réception des données

```

MPI_Ssend(data_adr, count, datatype, destproc, tag, comm)
MPI_Recv(data_adr, count, datatype, srcproc, tag, comm, stts_adr)

```

Mode rappelant les rendez-vous de CSP/Occam

Pbs de *dead-lock* si tous les processus émettent puis reçoivent

Communications point-à-point

## Principes du Send/Recv

Originalité du mode *bloquant et standard* MPI\_Send/MPI\_Recv

Non spécifié en détail et optimisable par chaque constructeur pour :

- obtenir de bonnes perf. sur ses machines
- être simple d'emploi sur ses machines

**Mais fonctionnement non portable !**

Ex, par le passé :

- MPICH : *send* bufferisé (similaire Bsend)
- IBM SP : < seuil : *send* bufferisé (similaire Bsend)
- > seuil : *send* synchrone (similaire Ssend)

2 approches opposées :

- Un pgm MPI ne doit pas utiliser de comms. *bloquantes-standard* : objectif de portabilité !
- Un pgm MPI doit utiliser des comms. *bloquantes-standards* optimisées par le constructeur : objectif de performances !

Cluster de PC  
Super-calculateurs

Communications point-à-point

## Principes du Rsend/Recv

Mode *bloquant et ready* : MPI\_Rsend - MPI\_Recv

- Rsend** : recherche un récepteur signalé, si oui : débute le transfert et rend la main à la fin de l'émission, sinon ? (erreur ? recopie ?)
- Recv** : se signale à l'émetteur (*ready*), et ne rend la main qu'à la fin de la réception des données

```

MPI_Rsend(data_adr, count, datatype, destproc, tag, comm)
MPI_Recv(data_adr, count, datatype, srcproc, tag, comm, stts_adr)

```

Mode sensé être très rapide

Mais demande des « garantis » de synchro « impossibles » !

Communications point-à-point

## Communications disponibles

Communications point-à-point **fiabiles** et **portables** :

Mode/Type	Not specified	Buffered	Synchronous	Ready
Bloquantes	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend
	MPI_Recv	MPI_Recv	MPI_Recv	MPI_Recv
Non - bloquantes	MPI_Ibsend	MPI_Ibsend	MPI_Issend	MPI_Irsend
	MPI_Irecv	MPI_Irecv	MPI_Irecv	MPI_Irecv

+ MPI\_Sendrecv  
MPI\_Sendrecv\_replace

Communications pt-à-pt combinées et bloquantes

Communications point-à-point

## Principes

Echanges de messages en mode **non-bloquant (asynchrone)**

- **Isend(...)** "lance un thread d'envoi de message" et rend la main
- **Irecv(...)** "lance un thread de réception de message" et rend la main
- Un **recouvrement** des comms avec la **suite des calculs** est alors possible
  - mais on ne doit plus utiliser les zones de stockage des données avant la fin des communications en cours
- **Wait(...)** resynchronise ensuite les calculs avec la fin des comms.
  - On peut alors réutiliser/exploiter les zones de stockage

```
// Ex de code d'UN process avec comm asynchrone
..... // calcul
1 : Isend(myTab,dest,&Srq); // lance thread de comm
2 : Irecv(otherTab,src,&Rrq); // lance thread de comm
3 : suite_calcul(...) // recouvrement calcul-comms
4 : Wait(&Srq); Wait(&Rrq); // resynchro calcul-comm
..... // fin des calculs
```

La programmation avec recouvrement se révèle toujours plus complexe...

Communications point-à-point

## Principes

Echanges de messages en mode **non-bloquant (asynchrone)**

- Parfois **Isend(...)** / **Irecv(...)** créent des threads qui restent inactifs
  - le recouvrement calcul-comm ne se produit pas !

Autre solution :

- Créer des threads traditionnels (Posix, OpenMP...)
- Leur faire exécuter des comms bloquantes → comms non bloquantes
- Resynchroniser les calculs et les comms avec une barrière sur la mort des threads de comm.

```
// Ex de code d'UN process avec comm asynchrone
..... // calcul
1 : tidS = thread(Send(myTab,dest)); // Thread de comm
2 : tidR = thread(Recv(otherTab,src)); // Thread de comm
3 : suite_calcul(...) // recouvrement calcul-comms
4 : threadJoin(tidS, tidR); // resynchro calcul-comm
..... // fin des calculs
```

La programmation avec recouvrement se révèle toujours plus complexe...

Communications point-à-point

## Communications disponibles

Communications point-à-point **fiabiles** et **portables** :

Mode/Type	Not specified	Buffered	Synchronous	Ready
Bloquantes	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend
	MPI_Recv	MPI_Recv	MPI_Recv	MPI_Recv
Non - bloquantes	MPI_Ibsend	MPI_Ibsend	MPI_Issend	MPI_Irsend
	MPI_Irecv	MPI_Irecv	MPI_Irecv	MPI_Irecv

+ Communications bloquantes dans des threads explicites, pour réaliser des comms. non-bloquantes

+ MPI\_Sendrecv  
MPI\_Sendrecv\_replace

Communications pt-à-pt combinées et bloquantes

Communications point-à-point

## Sendrecv

Communications **bloquantes combinées** : MPI\_Sendrecv

- Adapté à des **échanges de frontières** dans une topologie
- Communications bloquantes
- Pas de Dead-Lock possible (gérés par le système)
- Compatible avec Send et Recv aux extrémités

```
MPI_Sendrecv(send_adr, sendcount, sendtype, destproc, sendtag,
recv_adr, recvcount, recvtype, srcproc, recvtag,
comm, status_adr)
```

Ex : échange de frontières avec MPI\_Sendrecv

Sendrecv (me-1) Sendrecv (me+1) Sendrecv (me-1) Sendrecv (me+1) Sendrecv (me-1) Sendrecv (me+1)

→ Très pratique et très efficace ! Mais besoin de planifier le plan de comm. sur des topologies complexes

Communications point-à-point

## Sendrecv\_replace

Communications **bloquantes combinées** : MPI\_Sendrecv\_replace

- Adapté à des **circulation de données** dans une topologie
- Communications bloquantes
- Pas de Dead-Lock possible (gérés par le système)
- Pas d'écrasement de données (géré par le système)

```
MPI_Sendrecv_replace(data_adr, count, datatype,
destproc, sendtag, srcproc, recvtag,
comm, status_adr)
```

Ex : circulation avec MPI\_Sendrecv\_replace et 1 (seul) buffer par proc

Sendrecv\_replace (... , (me-1+P) %P, ..., (me+1) %P, ...)

Pb avec opérateur modulo

→ Très pratique et très efficace ! Mais besoin de planifier le plan de comm. sur des topologies complexes

Communications point-à-point

## Communications disponibles

Communications point-à-point fiables et portables :

Mode/Type	Not specified	Buffered	Synchronous	Ready
Bloquantes	MPI_Send MPI_Recv	MPI_Bsend MPI_Recv	MPI_Ssend MPI_Recv	MPI_Rsend MPI_Recv
	MPI_Ibsend MPI_Irecv	MPI_Ibsend MPI_Irecv	MPI_Issend MPI_Irecv	MPI_Irsend MPI_Irecv

+

Communications bloquantes dans des threads explicites, pour réaliser des comms. non-bloquantes

+

MPI\_Sendrecv  
MPI\_Sendrecv\_replace

Communications pt-à-pt combinées et bloquantes

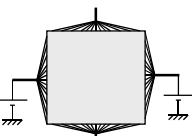
L'essentiel de MPI-1

## 3 – Exemple de programmation MPI

- Communications Send/Recv
- Communications Bsend/Recv
- Communications Ssend/Recv
- Communications Sendrecv

Exemple de programmation MPI

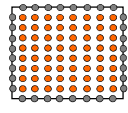
## Relaxation de Jacobi parallèle



Calcul des lignes de potentiel dans une plaque diélectrique :

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

- Discrétisation et équation aux différences
- Itération jusqu'à la convergence ( $V^{n+1}_{ij} - V^n_{ij} < \epsilon$ )



- Condition aux limites : V fixé
- $V^{n+1}_{ij} = \frac{V^n_{i-1,j} + V^n_{i+1,j} + V^n_{i,j-1} + V^n_{i,j+1}}{4}$

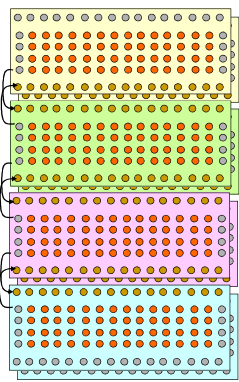
Exemple de programmation MPI

## Relaxation de Jacobi parallèle

Parallélisation par envoi de messages:

- **Partitionnement spatial des données**
- Ex : avec 4 processus de calcul

- 2 extraits des tables  $V^n$  et  $V^{n+1}$  dans chaque mémoire locale + les frontières
- **Parallélisation de la boucle principale**
- 1 barrière (implicite sur comm bloquantes)



Exemple de programmation MPI

## Implantation des initialisations

```

// Static parallelization: size and partitioning fixed
#define side      112 // Side of the Jacobi grid
#define small    28  // Need side = NbPE*small

int Me; // Processor number
int NbPE; // Number of processors

double V[2][small+2][side+2]; // Jacobi grids
int NI, OI; // Indexes of new & old grids

main(int argc, char **argv) {

    ..... // Jacobi grid initialization and relaxation loop

}

```

Exemple de programmation MPI

## Implantation des initialisations

```

// Static parallelization: size and partitioning fixed
#define side      112 // Side of the Jacobi grid
#define small    28  // Need side = NbPE*small

int Me; // Processor number
int NbPE; // Number of processors

double V[2][small+2][side+2]; // Jacobi grids
int NI, OI; // Indexes of new & old grids

main(int argc, char **argv) {
    MPI_Init(&argc, &argv); // MPI initializations
    MPI_Comm_rank(MPI_COMM_WORLD, &Me);
    MPI_Comm_size(MPI_COMM_WORLD, &NbPE);
    printf("Hello from PE%d/%d\n", Me, NbPE);

    ..... // Jacobi grid initialization and relaxation loop
    MPI_Finalize(); // End of MPI calls
}

```



Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  for (i = 1; i <= small; i++) {
    for (j = 1; j <= side; j++) {
      V[NI][i][j] =
        (V[OI][i-1][j]+
         V[OI][i+1][j]+
         V[OI][i][j-1]+
         V[OI][i][j+1])/4.0;
    }
  }
  // - Frontier exchange
  if (NbPE > 1) {
    .....
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  .....
  // - Frontier exchange
  if (NbPE > 1) {
    .....
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  .....
  // - Frontier exchange
  if (NbPE > 1) {
    if (Me == 0) {
      MPI_Send(&V[NI][small][1], side, MPI_DOUBLE,
              Me+1, cycle, MPI_COMM_WORLD);
      MPI_Recv(&V[NI][small+1][1], side, MPI_DOUBLE,
              Me+1, cycle, MPI_COMM_WORLD, &Status);
    } else if (Me == NbPE - 1) {
      .....
    } else {
      .....
    }
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  .....
  // - Frontier exchange
  if (NbPE > 1) {
    if (Me == 0) {
      MPI_Send(&V[NI][small][1], ..., Me+1, ...);
      MPI_Recv(&V[NI][small+1][1], ..., Me+1, ...);
    } else if (Me == NbPE - 1) {
      MPI_Send(&V[NI][1][1], ..., Me-1, ...);
      MPI_Recv(&V[NI][0][1], ..., Me-1, ...);
    } else {
      .....
    }
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  .....
  // - Frontier exchange
  if (NbPE > 1) {
    if (Me == 0) {
      MPI_Send(&V[NI][small][1], ..., Me+1, ...);
      MPI_Recv(&V[NI][small+1][1], ..., Me+1, ...);
    } else if (Me == NbPE - 1) {
      MPI_Send(&V[NI][1][1], ..., Me-1, ...);
      MPI_Recv(&V[NI][0][1], ..., Me-1, ...);
    } else {
      MPI_Send(&V[NI][small][1], ..., Me+1, ...);
      MPI_Send(&V[NI][1][1], ..., Me-1, ...);
      MPI_Recv(&V[NI][small+1][1], ..., Me+1, ...);
      MPI_Recv(&V[NI][0][1], ..., Me-1, ...);
    }
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

Exemple de programmation MPI  
**Implantation en Send-Recv**

```

OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
  // - Computation loop
  .....
  // - Frontier exchange
  if (NbPE > 1) {
    if (Me == 0) {
      MPI_Send(...);
      MPI_Recv(...);
    } else if (Me == NbPE - 1) {
      MPI_Send(...);
      MPI_Recv(...);
    } else {
      MPI_Send(...);
      MPI_Send(...);
      MPI_Recv(...);
      MPI_Recv(...);
    }
  }
  // - Index switching
  OI = 1-OI; NI = 1-NI;
}

```

**MPI\_Send / MPI\_Recv : simple mais peu portable !**

- Optimisé sur chaque supercalculateur
- Mais peut fonctionner différemment !

Rappel : 2 stratégies

- Pour des codes génériques sur « clusters de PC » : ne pas utiliser
- Sur super-calculateur propriétaire : utiliser + portage du code à chaque changement de machine

Exemple de programmation MPI  
**Implantation en Bsend-Recv**

```

// Buffer size and pointer declaration
int sizeMsg, sizeBuff;
void *PtBuff;

// Buffer size computation and allocation
MPI_Pack_size(side, MPI_DOUBLE, MPI_COMM_WORLD, &sizeMsg);
sizeBuff = 2*(sizeMsg + MPI_BSEND_OVERHEAD);
PtBuff = (void *) malloc(sizeBuff);
if (PtBuff == NULL) .....
// Relaxation loop (with comms)
OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
    // - Computation loop
    .....
    // - Frontier exchange
    .....
    // - Index switching
    OI = 1-OI; NI = 1-NI;
}
// Buffer free
free (PtBuff);

```

Exemple de programmation MPI  
**Implantation en Bsend-Recv**

```

// Buffer size computation and allocation
.....
// Relaxation loop (with comms)
OI = 0; NI = 1;
for (c = 0; c < NbCycle; c++) {
    // - Computation loop
    .....
    // - Frontier exchange
    if (NbPE > 1) {
        MPI_Buffer_attach(PtBuff, sizeBuff);
        if (Me == 0) {
            MPI_Bsend(...); MPI_Recv(...);
        } else if (Me == NbPE - 1) {
            MPI_Bsend(...); MPI_Recv(...);
        } else {
            MPI_Bsend(...); MPI_Bsend(...);
            MPI_Recv(...); MPI_Recv(...);
        }
        MPI_Buffer_detach(&PtBuff, &sizeBuff);
    }
    // - Index switching
    OI = 1-OI; NI = 1-NI;
}
// Buffer free
.....

```

Exemple de programmation MPI  
**Implantation en Ssend-Recv**

**Avantage :**  
On évite une copie mémoire

**Inconvénient :**  
Il faut planifier/séquencer les comms, sinon il y aura interblocage!

```

main(in argc, char ** argv) {
    ..... // MPI initialisations
    ..... // Non-MPI initialisations
    // RELAXATION LOOP
    for (cycle = 0; cycle < NbCycle; cycle++) {
        // - Computation loop
        // - Frontier exchange
        // - Index switching
    }
    // Print results and end of MPI calls
}

```

Exemple de programmation MPI  
**Implantation en Ssend-Recv**

```

// - Frontier exchange
if (NbPE > 1) {
    if (Me == 0) {
        MPI_Ssend(&V[NI][small][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
    } else if (Me == NbPE - 1) {
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
        MPI_Ssend(&V[NI][1][1][1], ..., Me-1, ...);
    } else if (Me%2 == 0) {
        MPI_Ssend(&V[NI][small][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
        MPI_Ssend(&V[NI][1][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
    } else /* if (Me%2 == 1) */ {
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
        MPI_Ssend(&V[NI][small][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
        MPI_Ssend(&V[NI][1][1][1], ..., Me-1, ...);
    }
}

```

Exemple de programmation MPI  
**Implantation en Ssend-Recv**

**Autre séquençement possible :**

```

main(in argc, char ** argv) {
    ..... // MPI initialisations
    ..... // Non-MPI initialisations
    // RELAXATION LOOP
    for (cycle = 0; cycle < NbCycle; cycle++) {
        // - Computation loop
        // - Frontier exchange
        // - Index switching
    }
    // Print results and end of MPI calls
}

```

Exemple de programmation MPI  
**Implantation en Ssend-Recv**

```

// - Frontier exchange
if (NbPE > 1) {
    if (Me == 0) {
        MPI_Ssend(&V[NI][small][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
    } else if (Me == NbPE - 1) {
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
        MPI_Ssend(&V[NI][1][1][1], ..., Me-1, ...);
    } else if (Me%2 == 0) {
        MPI_Ssend(&V[NI][1][1][1], ..., Me-1, ...);
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
    } else {
        MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
        MPI_Ssend(&V[NI][small][1][1], ..., Me+1, ...);
        MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
        MPI_Ssend(&V[NI][1][1][1], ..., Me-1, ...);
    }
}

```



Exemple de programmation MPI  
**Implantation en Rsend-Recv**

**Avantage :**  
Pas de recopie mémoire  
Emission rapide (suppression d'overhead)

**Inconvénient :**  
*Théoriquement* : nécessite que le recv soit fait avant !  
*sinon* : comportement non spécifié  
Ne pas modifier la zone transmise avant fin de transmission

```

main(in argc, char ** argv) {
  ..... // MPI initialisations
  ..... // Non-MPI initialisations
  // RELAXATION LOOP
  for (cycle = 0; cycle < NbCycle; cycle++) {
    ..... // - Computation loop
    ..... // - Frontier exchange
    ..... // - Index switching
  }
  // Print results and end of MPI calls
}

```

Exemple de programmation MPI  
**Implantation en Rsend-Recv**

```

// - Frontier exchange
if (NbPE > 1) {
  if (Me == 0) {
    MPI_Rsend(&V[NI][small][1][1], ..., Me+1, ...);
    MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
  } else if (Me == NbPE - 1) {
    MPI_Rsend(&V[NI][1][1][1], ..., Me-1, ...);
    MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
  } else {
    MPI_Rsend(&V[NI][1][1][1], ..., Me-1, ...);
    MPI_Rsend(&V[NI][small][1][1], ..., Me+1, ...);
    MPI_Recv(&V[NI][0][1][1], ..., Me-1, ...);
    MPI_Recv(&V[NI][small+1][1][1], ..., Me+1, ...);
  }
}

```

**Implantation « douteuse » :**  
fonctionne souvent correctement ...  
... mais sans garanti sur un simple cluster de PC !

Exemple de programmation MPI  
**Implantation en Sendrecv**

**Avantage :**  
Pas de gestion de buffer mémoire  
Deux opérations en une !

**Inconvénient :**  
Planifier/séquencez les Sendrecv

```

main(in argc, char ** argv) {
  ..... // MPI initialisations
  ..... // Non-MPI initialisations
  // RELAXATION LOOP
  for (cycle = 0; cycle < NbCycle; cycle++) {
    ..... // - Computation loop
    ..... // - Frontier exchange
    ..... // - Index switching
  }
  // Print results and end of MPI calls
}

```

Exemple de programmation MPI  
**Implantation en Sendrecv**

```

// - Frontier exchange
if (NbPE > 1) {
  if (Me == 0) {
    MPI_Sendrecv(&V[NI][small][1][1], ..., Me+1, ...
                &V[NI][small+1][1][1], ..., Me+1, ...);
  } else if (Me == NbPE - 1) {
    MPI_Sendrecv(&V[NI][1][1][1], ..., Me-1, ...
                &V[NI][0][1][1], ..., Me-1, ...);
  } else if (Me % 2 == 0) {
    MPI_Sendrecv(&V[NI][small][1][1], ..., Me+1, ...
                &V[NI][small+1][1][1], ..., Me+1, ...);
  } else /* if (Me % 2 == 1) */ {
    MPI_Sendrecv(&V[NI][1][1][1], ..., Me-1, ...
                &V[NI][0][1][1], ..., Me-1, ...);
  }
  MPI_Sendrecv(&V[NI][small][1][1], ..., Me+1, ...
              &V[NI][small+1][1][1], ..., Me+1, ...);
}

```

Exemple de programmation MPI  
**Implantation en Sendrecv**

**Autre séquençement possible :**

```

main(in argc, char ** argv) {
  ..... // MPI initialisations
  ..... // Non-MPI initialisations
  // RELAXATION LOOP
  for (cycle = 0; cycle < NbCycle; cycle++) {
    ..... // - Computation loop
    ..... // - Frontier exchange
    ..... // - Index switching
  }
  // Print results and end of MPI calls
}

```

Exemple de programmation MPI  
**Implantation en Sendrecv**

```

// - Frontier exchange
if (NbPE > 1) {
  if (Me == 0) {
    MPI_Sendrecv(&V[NI][small][1][1], ..., Me+1, ...
                &V[NI][small+1][1][1], ..., Me+1, ...);
  } else if (Me == NbPE - 1) {
    MPI_Sendrecv(&V[NI][1][1][1], ..., Me-1, ...
                &V[NI][0][1][1], ..., Me-1, ...);
  } else {
    MPI_Sendrecv(&V[NI][small][1][1], ..., Me+1, ...
                &V[NI][0][1][1], ..., Me-1, ...);
    MPI_Sendrecv(&V[NI][1][1][1], ..., Me-1, ...
                &V[NI][small+1][1][1], ..., Me+1, ...);
  }
}

```

- Le séquençement des MPI\_Sendrecv est plus simple (moins de cas à différencier)
- Le MPI\_Sendrecv de chaque extrémité se conjugue avec deux autres MPI\_Sendrecv

CentralesSupélec

## L'essentiel de MPI-1

# 4 – Communications de groupe

- Principes des communications de groupe
- Broadcast
- Scatter
- Gather
- Reduce

CentralesSupélec

## Communications de groupe

# Principes des comm. de groupe

5 types principaux :

**Broadcast**

**Scatter**

**Gather**

**Reduce(op)**

■ =  $op(op(\square), op(\square), op(\square), op(\square))$

+ les barrières !

**Principes :**

- Utilisent les *communicator* et les groupes de processus
- Opérations bloquantes
- Des variantes existent : *all-reduce*, *all-to-all*, *scatterv*, ...

**Intérêt dans un supercalculateur :**  
 Le routage est optimisé selon le réseau sous-jacent (arborescent – linéaire – sur bus – ...)

CentralesSupélec

## Communications de groupe

# Broadcast

```
int MPI_Bcast(buffer, count, datatype, root, comm)
void *buffer; // Starting address of buffer
int count; // Number of elts in buffer (integer)
MPI_Datatype datatype; // Data type of buffer
int root; // Rank of broadcast root (integer)
MPI_Comm comm; // Communicator
```

datatype  
datatype  
.....  
datatype

count

Chaque processus exécute MPI\_Bcast (en émetteur ou récepteur)

**Généralisation :**  
 MPI\_Alltoall et MPI\_Alltoallv

CentralesSupélec

## Communications de groupe

# Scatter

```
int MPI_Scatter(sendbuf, sendcnt, sendtype,
               recvbuf, recvcnt, recvtype, root, comm)
void *sendbuf; // Address of send buffer
int sendcnt; // Nb of elements sent to each process
MPI_Datatype sendtype; // Data type of elt to send
void *recvbuf; // Address of receive buffer
int recvcnt; // Number of elements in receive buffer
MPI_Datatype recvtype; // Data type of elt to receive
int root; // Rank of the sending process
MPI_Comm comm; // Communicator
```

sendtype

sendcnt

- Chaque processus exécute MPI\_Scatter (en émetteur ou récepteur)
- Le buffer d'émission n'a de sens que sur le processus root

**Généralisation :**  
 MPI\_Scatterv (avec partitionnement explicite des données)

CentralesSupélec

## Communications de groupe

# Gather

```
int MPI_Gather(sendbuf, sendcnt, sendtype,
              recvbuf, recvcnt, recvtype, root, comm)
void *sendbuf; // Starting address of send buffer
int sendcnt; // Number of elements in send buffer
MPI_Datatype sendtype; // Data type of elts to send
void *recvbuf; // Address of receive buffer
int recvcnt; // Nb of elts to receive from each proc
MPI_Datatype recvtype; // Data type of elt to recv
int root; // Rank of the receiving process
MPI_Comm comm; // Communicator
```

Sendtype

sendcnt

- Chaque processus exécute MPI\_Gather (en émetteur ou récepteur)
- Le buffer de réception n'a de sens que sur le processus root

**Généralisation :**  
 MPI\_Gatherv, MPI\_Allgather, MPI\_Allgatherv

CentralesSupélec

## Communications de groupe

# Reduce

```
int MPI_Reduce(sendbuf, recvbuf, count, datatype, op,
              root, comm)
void *sendbuf; // Address of send buffer
void *recvbuf; // Address of receive buffer
int count; // Number of elts in send buffer
MPI_Datatype datatype; // Data type of elts to send
MPI_Op op; // Reduce operation
int root; // Rank of the process hosting result
MPI_Comm comm; // Communicator
```

■ =  $op(op(\square), op(\square), op(\square), op(\square))$

- Opérations de réduction disponibles :  
 MPI\_MAX, MPI\_MIN, MPI\_SUM, MPI\_PROD,  
 MPI\_LAND, MPI\_BAND, MPI\_LOR, MPI BOR,  
 MPI\_LXOR, MPI\_BXOR, MPI\_MINLOC
- Définition de nouvelles opérations avec  
 MPI\_Op\_create()

**Généralisation :**  
 MPI\_Allreduce, MPI\_Reduce\_scatter : les res sont redistribués

CentralesSupélec

## L'essentiel de MPI-1

# 5 – Stratégie de déploiement

- Placement de processus sur cluster de multi-cœurs
- Processus et threads sur clusters de multicœurs
- Meilleur déploiement ?

CentralesSupélec

## Stratégie de déploiement

### Processus sur clusters de multicœurs (1)

nodes × sockets × cores

Ressources matérielles:

Topologie virtuelle de l'algorithme (ex : anneau)

→ Quel *mapping* de la topologie virtuelle sur les rsrscs matérielles ?  
 → Quel *ranking* des processus MPI sur la topologie matérielle ?  
 → Quel *binding* d'un process sur les rsrscs matérielles locales ?

CentralesSupélec

## Stratégie de déploiement

### Processus sur clusters de multicœurs (1)

nodes × sockets × cores

Ressources matérielles:

Topologie virtuelle de l'algorithme (ex : anneau)

```

mpirun -np #P -machinefile machines.txt
-map-by <node|socket|core...>
-rank-by <node|socket|core...>
-bind-to <none|socket|core...>
--report-bindings .....
  
```

CentralesSupélec

## Stratégie de déploiement

### Processus sur clusters de multicœurs (1)

nodes × sockets × cores

Ressources matérielles:

Topologie virtuelle de l'algorithme (ex : anneau)

```

mpirun -np #P -machinefile machines.txt
-map-by ppr:N:<node|socket|core...>
-rank-by <node|socket|core...>
-bind-to <none|socket|core...>
--report-bindings .....
  
```

CentralesSupélec

## Stratégie de déploiement

### Processus sur clusters de multicœurs (2)

**Sol 1 : un processus MPI par nœud**

- Chaque machine présente une fois dans le fichier des machines ciblées
- Nbr de machines listées  $\geq P$  (P : nbr de processus a créer)
- Utile si un processus nécessite toutes le cache ou toute la RAM d'un nœud / d'un socket

→ Reste un mode « luxueux »

machines.txt

```

Node 0
Node 1
....
Node P-1
....
Node max
  
```

Attention : le binding d'un process peut être limité à un socket...

CentralesSupélec

## Stratégie de déploiement

### Processus sur clusters de multicœurs (2)

**Sol 1 : un processus MPI par nœud**

- Chaque machine présente une fois dans le fichier des machines ciblées
- Nbr de machines listées  $\geq P$  (P : nbr de processus a créer)

machines.txt

```

Node 0
Node 1
....
Node P-1
....
Node max
  
```

```

mpirun -np #P -machinefile machines.txt
-map-by ppr:1:node
-rank-by node
-bind-to none|socket .....
  
```

Stratégie de déploiement

### Processus sur clusters de multicœurs (3)

**Sol 2 : un processus MPI par cœur en « round robin »**

- Nbr de machines nécessaires =  $P / C$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)
- Simple, mais toutes les communications se font à travers le réseau.

`machines.txt`

```
Node 0
Node 1
....
Node P/C-1
....
Node max
```

Stratégie de déploiement

### Processus sur clusters de multicœurs (3)

**Sol 2 : un processus MPI par cœur en « round robin »**

- Nbr de machines nécessaires =  $P / C$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)

```
mpirun -np #P -machinefile machines.txt
-map-by ppr:#C:node
-rank-by node
-bind-to core .....
```

`machines.txt`

```
Node 0
Node 1
....
Node P/C-1
....
Node max
```

Stratégie de déploiement

### Processus sur clusters de multicœurs (4)

**Sol 3 : un processus MPI par cœur en « mode compact »**

- Nbr de machines nécessaires =  $P / C$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)
- Un maximum de communications se font au sein d'un même nœud : plus efficace!  
*Ce déploiement dépend de la topologie virtuelle et de l'architecture réelle.*

`machines.txt`

```
Node 0
Node 1
....
Node P/C-1
....
Node max
```

Stratégie de déploiement

### Processus sur clusters de multicœurs (4)

**Sol 3 : un processus MPI par cœur en « mode compact »**

- Nbr de machines nécessaires =  $P / C$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)

```
mpirun -np #P -machinefile machines.txt
-map-by ppr:#C:node
-rank-by core
-bind-to core .....
```

`machines.txt`

```
Node 0
Node 1
....
Node P/C-1
....
Node max
```

Stratégie de déploiement

### Processus sur clusters de multicœurs (4)

**Sol 3 : un processus MPI par cœur en « mode compact »**

- Nbr de machines nécessaires =  $P / C$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)

```
mpirun -np #P -machinefile machines.txt
-map-by ppr:1:core
-rank-by core
-bind-to core .....
```

`machines.txt`

```
Node 0
Node 1
....
Node P/C-1
....
Node max
```

Stratégie de déploiement

### Processus et threads sur clusters de multicœurs

**Sol 4 : un processus MPI par nœud et C threads/processus**

- Nbr de machines listées  $\geq P$   
( $P$  : nbr de processus à créer,  $C$  : nb de cœurs/nœud)
- Un processus/nœud, puis chaque processus créera  $C$  threads

```
mpirun -np #P -machinefile machines.txt
-map-by ppr:1:node
-rank-by node
-bind-to none .....
```

`machines.txt`

```
Node 0
Node 1
....
Node P-1
....
Node max
```

Refuse parfois de binder sur le nœud entier

Stratégie de déploiement

## Processus et threads sur clusters de multicœurs

**Sol 5 : un processus MPI par Socket et C threads/processus**

- Nbr de machines listées  $\geq P/S$   
(P : nbr de processus à créer,  
S sockets/nœud, Cs : nb de cœurs/socket)
- Un processus/socket, puis chaque processus créera Cs threads

`mpirun -np #P -machinefile machines.txt`

`-map-by ppr:1:socket`  
`-rank-by socket`  
`-bind-to socket .....`

**Solution plus efficace sur nœuds modernes/NUMA**

`machines.txt`

```
Node 0
Node 1
....
Node P/S-1
....
Node max
```

Stratégie de déploiement

## Options de binding

Des options pour contrôler les ressources allouées à un processus multithreadé :

```
mpirun --bind-to < none | hwthread | core |
socket | numa | board |
11cache | 12cache | 13cache >
```

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 1 :**

- Ressources allouées : 8 nœuds
- Stratégie de déploiement de l'application *myAppli* :
  - un processus MPI par cœur physique,
  - processus de numéros consécutifs sur un même nœud
  - un thread par cœur logique (hyp : option `-nt` de l'application)

**Question 1 :** combien de processus doit-on créer (option `-np` de *mpirun*) ?  
→  $8 \times 4 \times 16 = 512$  processus MPI

**Question 2 :** combien de *threads* doit créer chaque processus (option `-nt`) ?  
→ 2 threads

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 1 :**

- Ressources allouées : 8 nœuds
- Stratégie de déploiement de l'application *myAppli* :
  - un processus MPI par cœur physique,
  - processus de numéros consécutifs sur un même nœud
  - un thread par cœur logique (hyp : option `-nt` de l'application)

**Question 3 :** quelle commande *mpirun* doit-on entrer ?  
→ `mpirun -np 512 -machinefile machines.txt`  
`-map-by ppr:1:core -rank-by core -bind-to core`  
`myAppli -nt 2`

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 2 :**

- Ressources allouées : 16 nœuds
- Stratégie de déploiement de l'application *myAppliOptim* :
  - un processus MPI par socket,
  - processus de numéros consécutifs sur un même nœud
  - un thread par cœur physique (hyp : option `-nt` de l'application)

**Question 1 :** combien de processus doit-on créer (option `-np` de *mpirun*) ?  
→  $16 \times 4 = 64$  processus MPI

**Question 2 :** combien de *threads* doit créer chaque processus (option `-nt`) ?  
→ 16 threads

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 2 :**

- Ressources allouées : 16 nœuds
- Stratégie de déploiement de l'application *myAppliOptim* :
  - un processus MPI par socket,
  - processus de numéros consécutifs sur un même nœud
  - un thread par cœur physique (hyp : option `-nt` de l'application)

**Question 3 :** quelle commande *mpirun* doit-on entrer ?  
→ `mpirun -np 64 -machinefile machines.txt`  
`-map-by ppr:1:socket -rank-by socket -bind-to socket`  
`myAppliOptim -nt 16`

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 3 :**

- Ressources allouées : 16 nœuds
- Stratégie de déploiement de l'application *myAppliOptim* :
  - 4 processus MPI par socket,
  - processus de numéros consécutifs sur des nœuds différents
  - un thread par cœur physique (hyp : option *-nt* de l'application)

**Question 1 :** combien de processus doit-on créer (option *-np* de *mpirun*) ?  
→  $16 \times 4 \times 4 = 256$  processus MPI

**Question 2 :** combien de *threads* doit créer chaque processus (option *-nt*) ?  
→ 4 threads

Stratégie de déploiement

## Exemple de déploiement

**Cluster :** 16 nœuds ×  
4 sockets (processeurs) ×  
16 cœurs physiques ×  
2 threads (hyperthreading)

**Règles :**

- On alloue  $n$  nœuds ( $1 \leq n \leq 16$ )
- On cherche toujours à utiliser tous les cœurs des nœuds alloués

**Pb 3 :**

- Ressources allouées : 16 nœuds
- Stratégie de déploiement de l'application *myAppliOptim* :
  - 4 processus MPI par socket,
  - processus de numéros consécutifs sur des nœuds différents
  - un thread par cœur physique (hyp : option -

**Question 3 :** quelle commande *mpirun* doit-on entrer ?

→ *mpirun -np 256 -machinefile machines.txt -map-by ppr:4:socket -rank-by node -bind-to socket myAppliOptim -nt 4*

Sur chaque socket l'OS réparti 4×4 threads sur 16 cœurs

Stratégie de déploiement

## Rapport de binding

```
mpirun -np 4 -machinefile machsar.txt
-map-by ppr:1:socket -rank-by socket -bind-to socket
--report-bindings
./MatrixProduct -k 1 -klc 4 -nt 4
```

32 nodes × 2 sockets × 4 cores × hyperthreading

```
[sar01:85379] MCW rank 0 bound to
socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]],
socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]]:
[BB/BB/BB/BB][../../../../]
[sar01:85379] MCW rank 1 bound to
socket 1[core 4[hwt 0-1]], socket 1[core 5[hwt 0-1]],
socket 1[core 6[hwt 0-1]], socket 1[core 7[hwt 0-1]]:
[../../../../][BB/BB/BB/BB]
[sar02:78868] MCW rank 3 bound to
socket 1[core 4[hwt 0-1]], socket 1[core 5[hwt 0-1]],
socket 1[core 6[hwt 0-1]], socket 1[core 7[hwt 0-1]]:
[../../../../][BB/BB/BB/BB]
[sar02:78868] MCW rank 2 bound to
socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]],
socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]]:
[BB/BB/BB/BB][../../../../]
```

Stratégie de déploiement

## Meilleur déploiement ?

**Dépend de plusieurs facteurs :**

- Schéma de communication (topologie des processus MPI)  
→ *Minimiser les communications inter-nœuds*
- Architecture NUMA des nœuds  
→ *Au moins un processus par sous-nœud NUMA*
- Bibliothèque de multithreading et d'envoi de messages disponibles  
→ *Toutes les implantations des bibliothèques de multithreading et de MPI ne sont pas équivalentes*

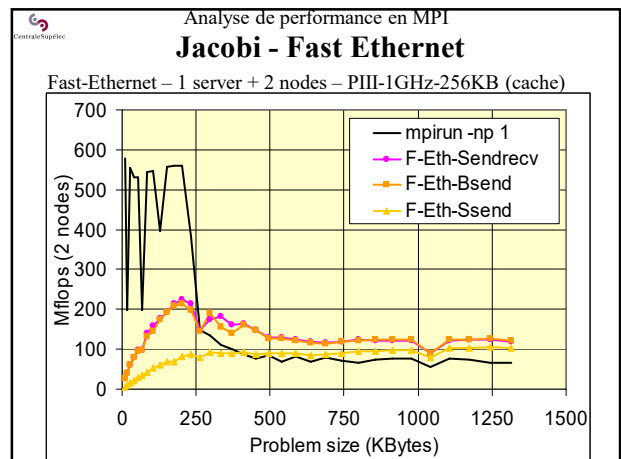
Quand on travaille toujours sur un même cluster on finit par savoir ce qui est le plus efficace sur lui !

Stratégie de déploiement

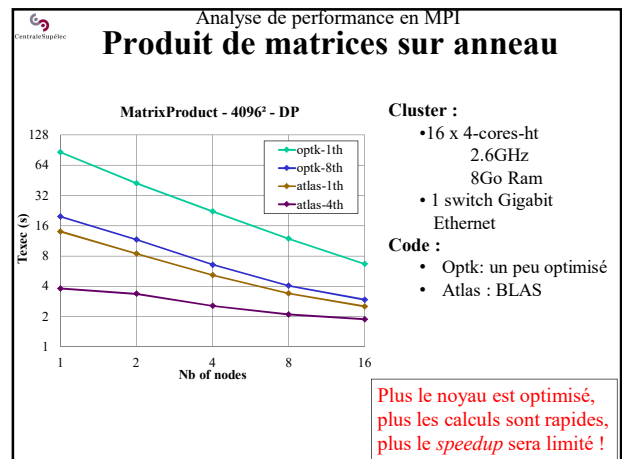
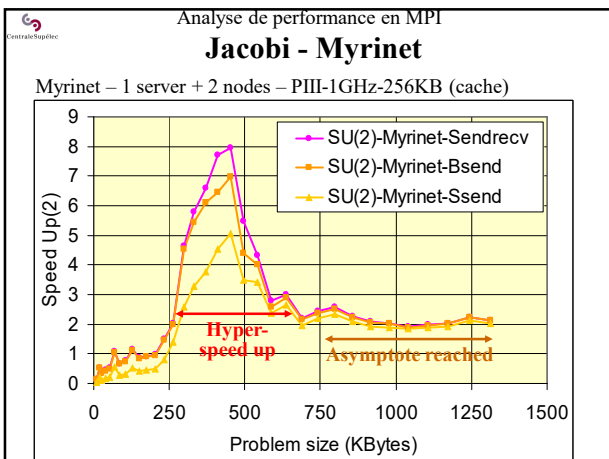
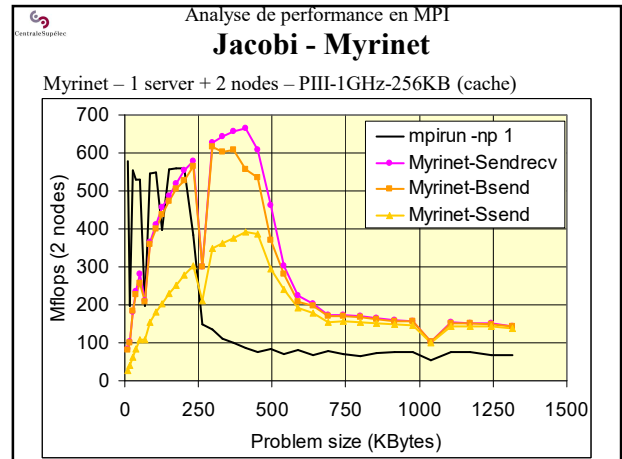
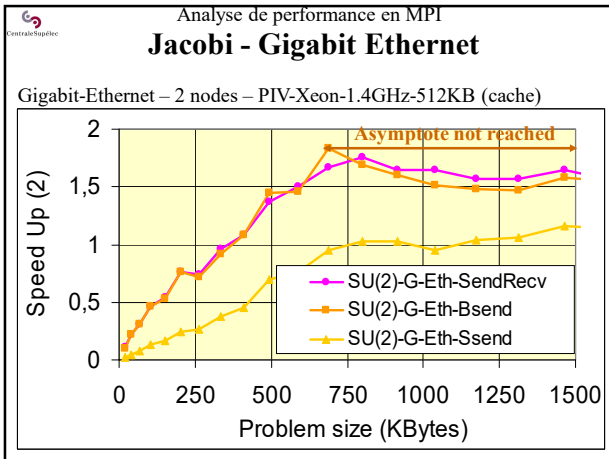
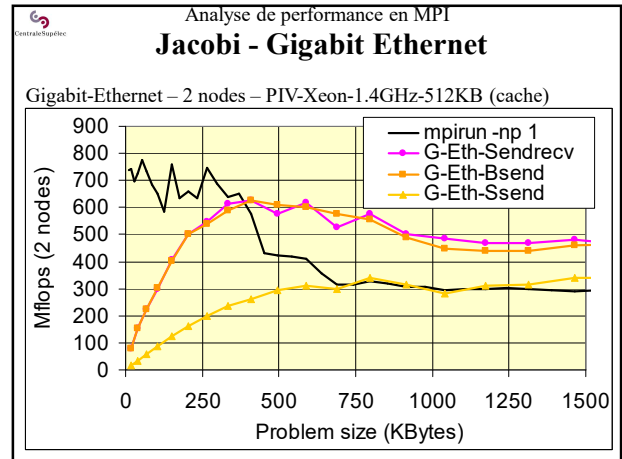
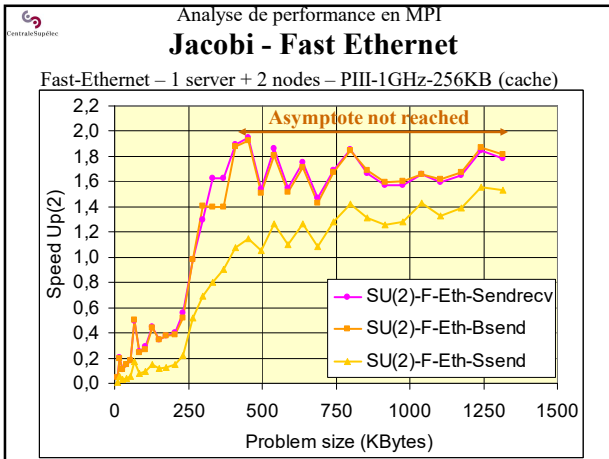
## L'essentiel de MPI-1

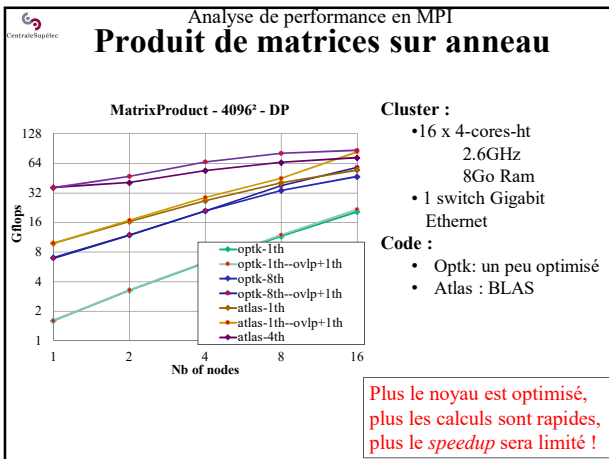
### 6 – Analyse de performances en MPI

- Jacobi sur (tout) petits clusters
- Produit de matrices
- Ce qui compte ....









### Analyse de performance en MPI

## Il n'y a pas que le nbr de processeurs qui compte lors des choix

Mémoire et mémoire cache ....

- Optimisations sérielles ! (vaste)
- Stratégie de blocage en cache
- Recherche d'hyper-accélération.

Processeurs

- Nombre de processeurs
- Puissance des procs.
- **Consommation électrique**

Réseau d'interconnexion

- **Bw ?**
- **Latence ?**
- Capacité d'extensibilité ?

Équilibrage des ressources :

- **Mémoire vs processeur**
- **Nbr de nœuds vs réseau**
- **Performances calculatoires vs énergétiques**

### L'essentiel de MPI-1

## 7 – TP : Produit de matrices denses sur un anneau de processeurs

- Topologies virtuelles
- Algorithme distribué sur un anneau
- Modélisation des performances sans recouvrement

### Produit de matrices denses sur anneau

## Topologies virtuelles

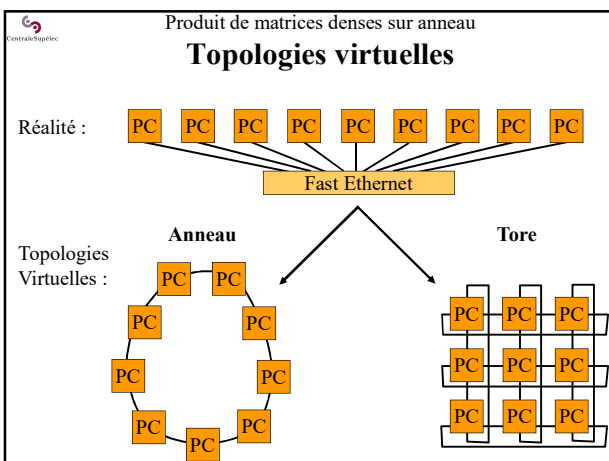
Principe du produit de matrices denses sur cluster :

- Partitionnement initial des matrices A et B
- Boucle de calcul et de circulation des données :
  - Sans recouvrement des calculs et des communications  
**Loop {calcul ; circulation}**
  - Avec recouvrement des calculs et des communications  
**Loop {par{calcul; circulation}}**

Différents algorithmes testés :

- Circulation des données sur un anneau
- Circulation des données sur un tore

Quelle topologie virtuelle est supportable par le cluster réel ?



### Produit de matrices denses sur anneau

## Algorithme distribué

Problème à résoudre :

$$C = A \cdot B \quad c_{ij} = \sum_{k=1}^n (a_{ik} \cdot b_{kj}) \quad O(\text{Nbr d'opérations}) = O(N^3/2)$$

Comment répartir les données ?

Pb du **partitionnement**

Calculs déterministes localisés :

- Partitionnement statique

Calculs déterministes NON localisés :

- Duplication ?
- **Circulation ?**

Produit de matrices denses sur anneau  
**Algorithme distribué**

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 0**  
(état initial)

Produit de matrices denses sur anneau  
**Algorithme distribué**

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 1**

Produit de matrices denses sur anneau  
**Algorithme distribué**

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 2**

Produit de matrices denses sur anneau  
**Algorithme distribué**

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Déroulement de l'algorithme sur **PE-2**, avec  $P = 4$  :

Etape 0 C

Etape 1 C

Etape 2 C

Etape 3 C

Produit de matrices denses sur anneau  
**Algorithme distribué**

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Résultats à la fin des  $P$  étapes :

Topologie

Partitionnement statique de C

**Bilan :**

- Chaque PC a calculé un bloc de colonnes de C
- Les  $P$  PC ont travaillé en parallèle

→ **Calcul de tous les blocs de colonnes en parallèle**, en  $P$  étapes

Produit de matrices denses sur anneau  
**Algorithme distribué**

Stratégies d'implantation sur un anneau de  $P$  processeurs :

```

// Sans recouvrement
for (step=0; step<P; step++)
  calcul();
  barrier(); // si besoin
  circulation();
  barrier(); // si besoin
}

// Avec recouvrement
for (step=0; step<P; step++)
  thread{ calcul(); }
  thread{ circulation(); }
  barrier();
  permutBuff();
}

```

↓

- Concevoir l'algorithme avec des barrières de (re)synchronisation potentielles
- Selon le mécanisme de communication utilisé :
  - Implanter des barrières explicites : synchronisation forte ou bien
  - Se contenter de la synchro des comm : synchronisation relaxée

Produit de matrices denses sur anneau

## Algorithme distribué

Stratégies d'implantation sur un anneau de P processeurs :

<pre>// Sans recouvrement for (step=0; step&lt;P; step++) calcul(); barrier(); // si besoin circulation(); barrier(); // si besoin }</pre>	<pre>// Avec recouvrement for (step=0; step&lt;P; step++) thread{ calcul(); } thread{ circulation(); } barrier(); permutBuff(); }</pre>
--	---

↓

- Concevoir l'algorithme avec une barrière de (re)synchronisation
- Implanter la barrière de (re)synchronisation
- Permuter les buffers de calcul et de communication (quasi-obligatoire)

Produit de matrices denses sur anneau

## Modélisation de performances

Performances sans recouvrement :

- Temps séquentiel :  $T_{seq} = N \cdot (2 \cdot \sqrt{N} - 1) t_{flop}$   
 $T_{seq} \approx 2 \cdot N \cdot \sqrt{N} t_{flop}$
- Temps parallèle :  $t_{1-calc} = \frac{\sqrt{N}}{P} \cdot \frac{\sqrt{N}}{P} \cdot (2 \cdot \sqrt{N} - 1) t_{flop}$   
 $t_{1-calc} \approx 2 \cdot \frac{N \cdot \sqrt{N}}{P^2} t_{flop}$   
 $t_{1-circ} = t_s + \sqrt{N} \cdot \frac{\sqrt{N}}{P} t_w$   
 $t_{1-circ} \approx \frac{N}{P} t_w$

Modèles & hypothèses !

Speed up :  $S_{no} = \frac{T_{seq}}{T_{par}^{no}} \approx P \cdot \frac{1}{1 + \frac{P t_w}{2 \cdot \sqrt{N} t_{flop}}}$

$T_{par}^{no} \approx P \cdot (t_{1-calc} + t_{1-circ})$   
 $T_{par}^{no} \approx 2 \cdot \frac{N \cdot \sqrt{N}}{P} t_{flop} + N t_w$

Produit de matrices denses sur anneau

## Modélisation de performances

Comparaison calculs-communications (sans recouvrement) :

- Calculs :  $T_{par}^{calc} \approx 2 \cdot \frac{N^{3/2}}{P} t_{flop}$   
 $P \text{ fixé} \Rightarrow O(T_{par}^{calc}) = O(N^{3/2})$
- Circulation :  $T_{par}^{circ} \approx N t_w$   
 $O(T_{par}^{circ}) = O(N)$

$O(T_{par}^{calc}) > O(T_{par}^{circ})$

A P fixé, quand N ↑ :  
les calculs deviennent prépondérants sur les surcoûts  
→ les performances s'améliorent (asymptotes idéales)

Bon problème !

Produit de matrices denses sur anneau

## Modélisation de performances

Volume de communications sur le réseau d'interconnexion :

Cluster :  $N_{nodes} \times N_{socket/node} \times N_{coeurs/socket} = N_n \times N_s \times N_c$   
Déploiement : 1 processus/nœud et  $N_s \times N_c$  threads par processus

- $N_n$  processus et étapes
- Volume d'une tranche de matrice A :  $Q_{slice} = Q_s = \frac{Q_A}{N_n}$

A chaque étape : - chaque processus émet  $Q_s$  data  
- chaque nœud émet  $Q_s$  data vers un autre nœud

Au final :

Volume total de données traversant le réseau durant tout le calcul :  $N_n \times (N_n \times Q_s) = N_n \times Q_A$

Taille de chaque message :  $Q_A / N_n$

Produit de matrices denses sur anneau

## Modélisation de performances

Volume de communications sur le réseau d'interconnexion :

Cluster :  $N_{nodes} \times N_{socket/node} \times N_{coeurs/socket} = N_n \times N_s \times N_c$   
Déploiement : 1 processus/socket et  $N_c$  threads par processus et numérotation selon les sockets

- $N_n \times N_s$  processus et étapes
- Volume d'une tranche de matrice A :  $Q_{slice} = Q_s = \frac{Q_A}{(N_n \times N_s)}$

A chaque étape : - chaque processus émet  $Q_s$  data  
- chaque nœud émet  $Q_s$  data vers un autre nœud

Au final :

Volume total de données traversant le réseau durant tout le calcul :  $(N_n \times N_s) \times (N_n \times Q_s) = N_n \times Q_A$

Taille de chaque message :  $Q_A / (N_n \times N_s)$

Plus de plus petits msg, mais même volume total

Produit de matrices denses sur anneau

## Modélisation de performances

Volume de communications sur le réseau d'interconnexion :

Cluster :  $N_{nodes} \times N_{socket/node} \times N_{coeurs/socket} = N_n \times N_s \times N_c$   
Déploiement : 1 processus/socket et  $N_c$  threads par processus et numérotation selon les nœuds

- $N_n \times N_s$  processus et étapes
- Volume d'une tranche de matrice A :  $Q_{slice} = Q_s = \frac{Q_A}{(N_n \times N_s)}$

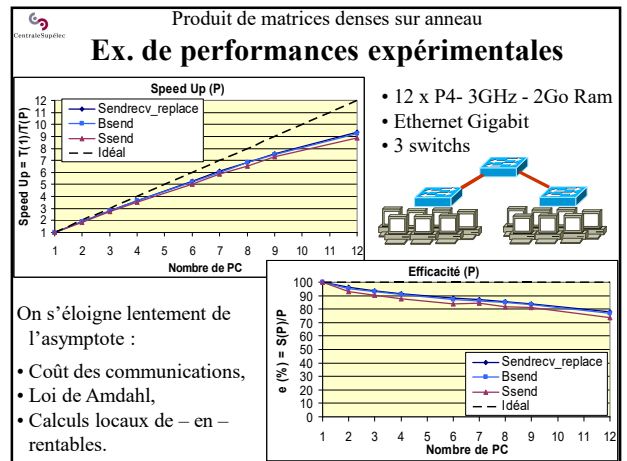
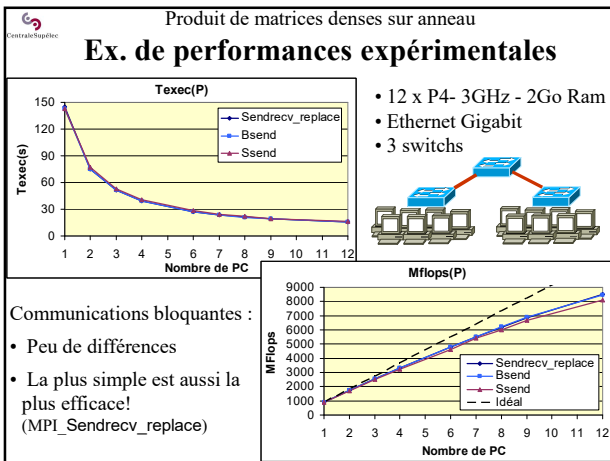
A chaque étape : - chaque processus émet  $Q_s$  data  
- chaque nœud émet  $N_s \times Q_s$  data vers d'autres nœuds

Au final :

Volume total de données traversant le réseau durant tout le calcul :  $(N_n \times N_s) \times (N_n \times (N_s \times Q_s)) = N_n \times N_s \times Q_A$

Taille de chaque message :  $Q_A / (N_n \times N_s)$

Bcp plus de plus petits msg, et volume total plus important !  
→ Plus lent !



Produit de matrices denses sur anneau  
**Développements et tests**

→ TP

Produit de matrices denses sur anneau  
**L'essentiel de MPI-1**

Questions ?