Approche distribuée

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

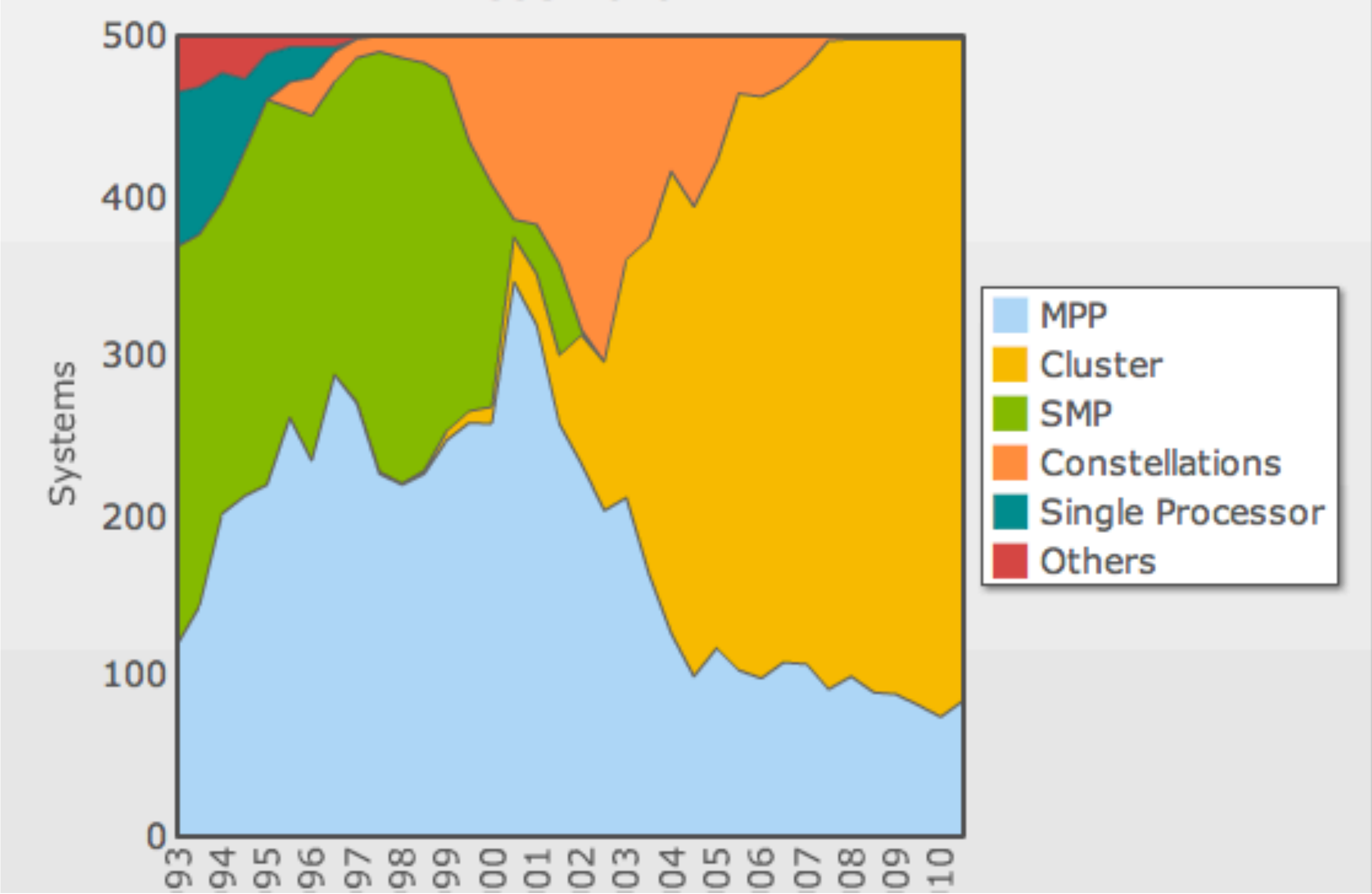
MPI_Recv(buffer, count, type, source, tag, communicateur, &status)

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 - 2010



Différentes architectures

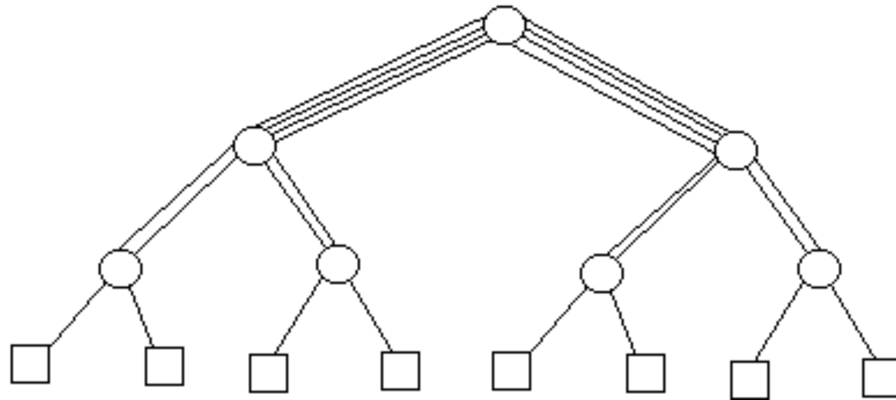
- Grappes (clusters)
  - Regroupement de serveurs de calcul
    - Processeurs standards
    - Parfois de grandes machines NUMA
  - Réseaux d'interconnexion rapide

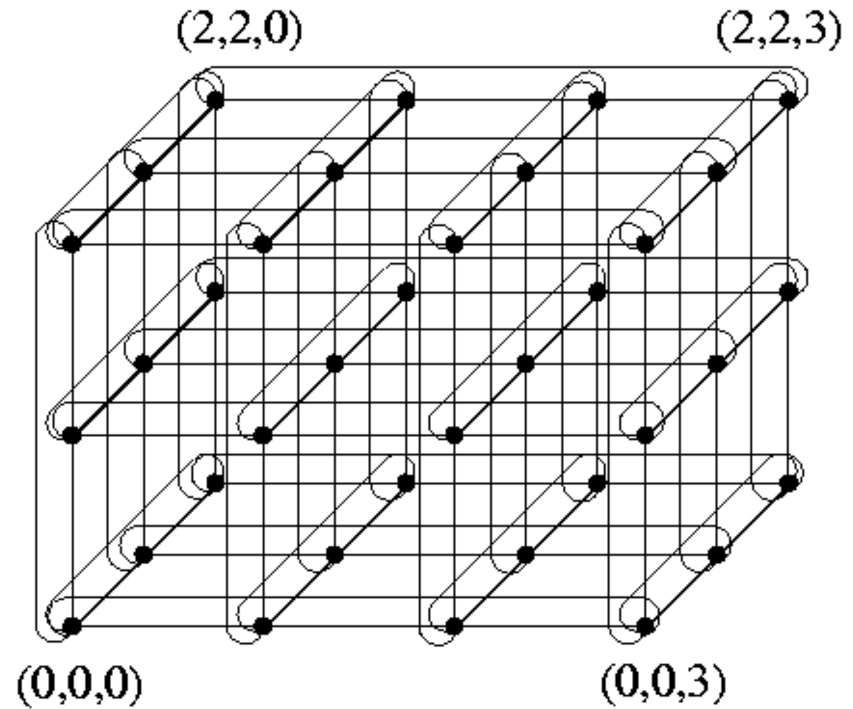
Différentes architectures

- Grappes (clusters)
  - Regroupement de serveurs de calcul
  - Réseaux d'interconnexion rapide
    - Topologie variée
    - Faible latence, gros débit

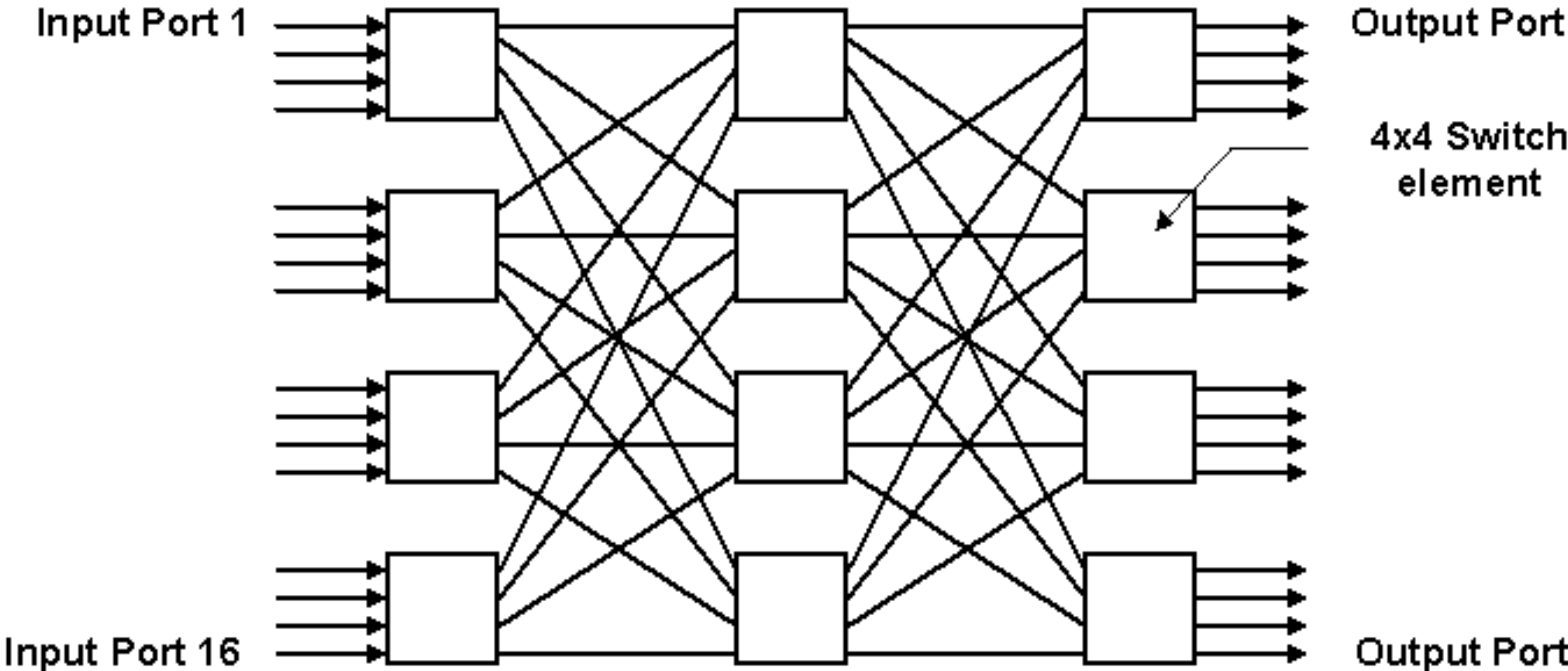
Solution la plus commune

Fat Tree – 3D Torus

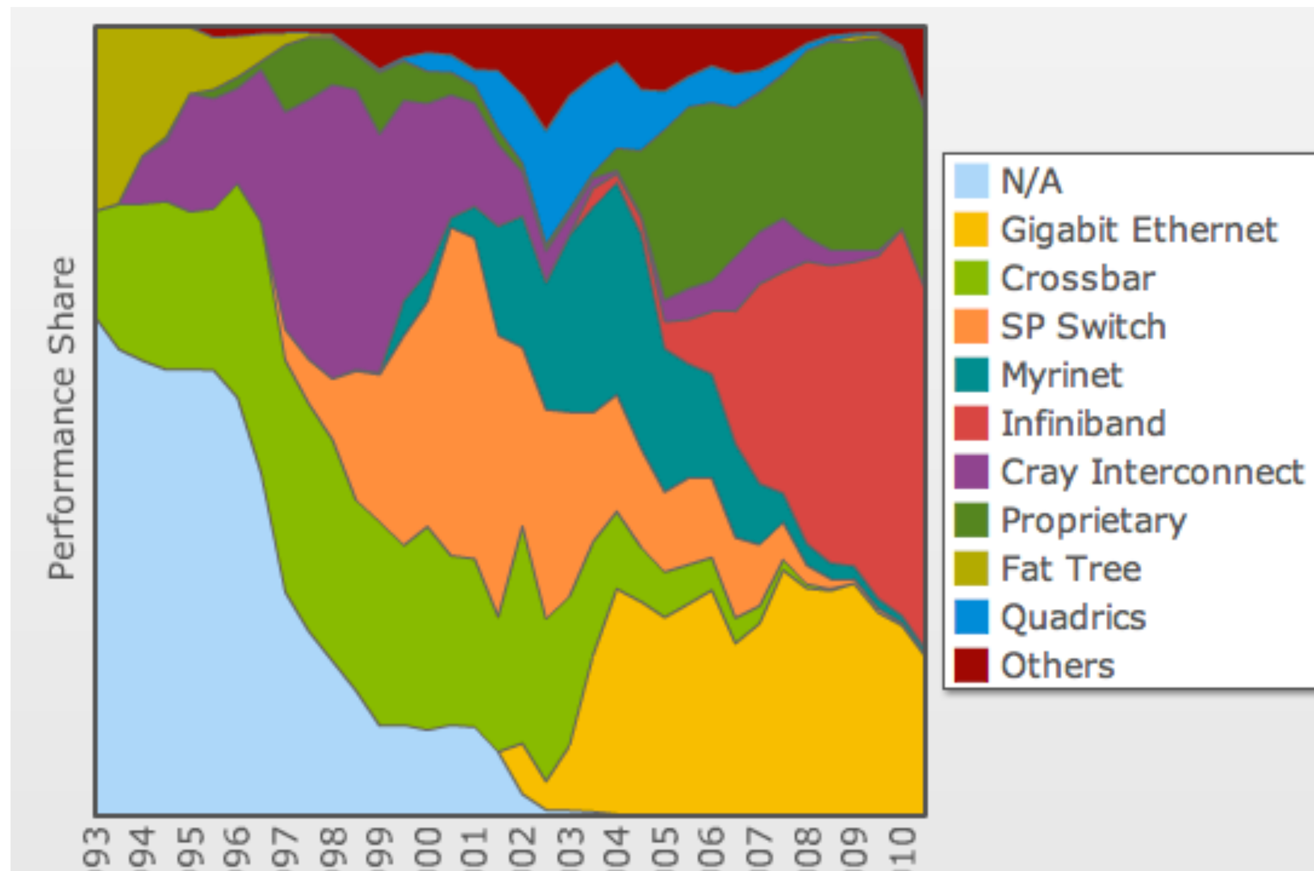




Clos



Top 500
Réseaux

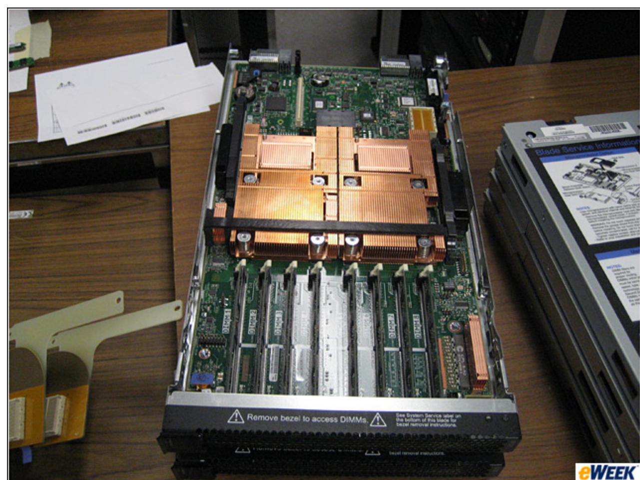


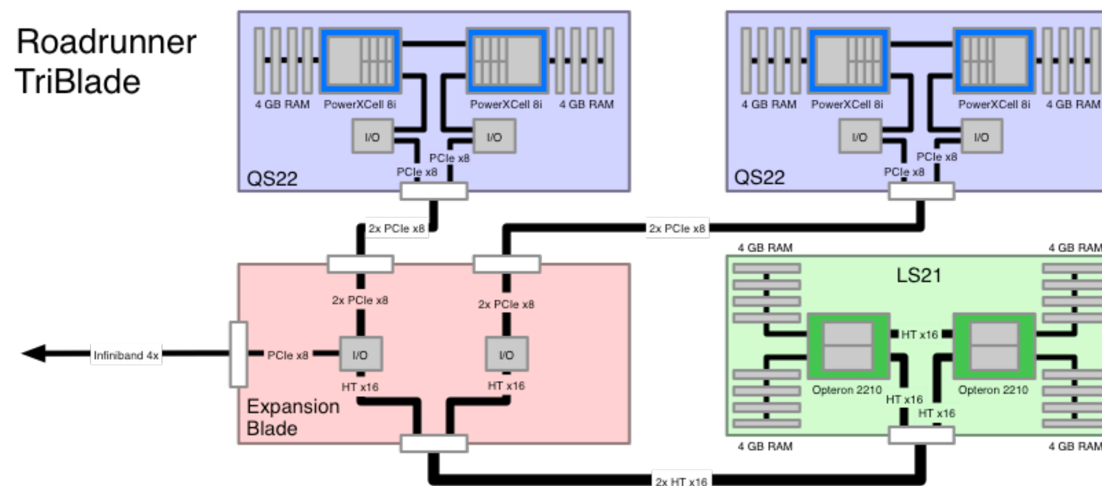
Différentes architectures

- Grappes (clusters)
- Machines hétérogènes
    - Accélérateurs GPU

Différentes architectures

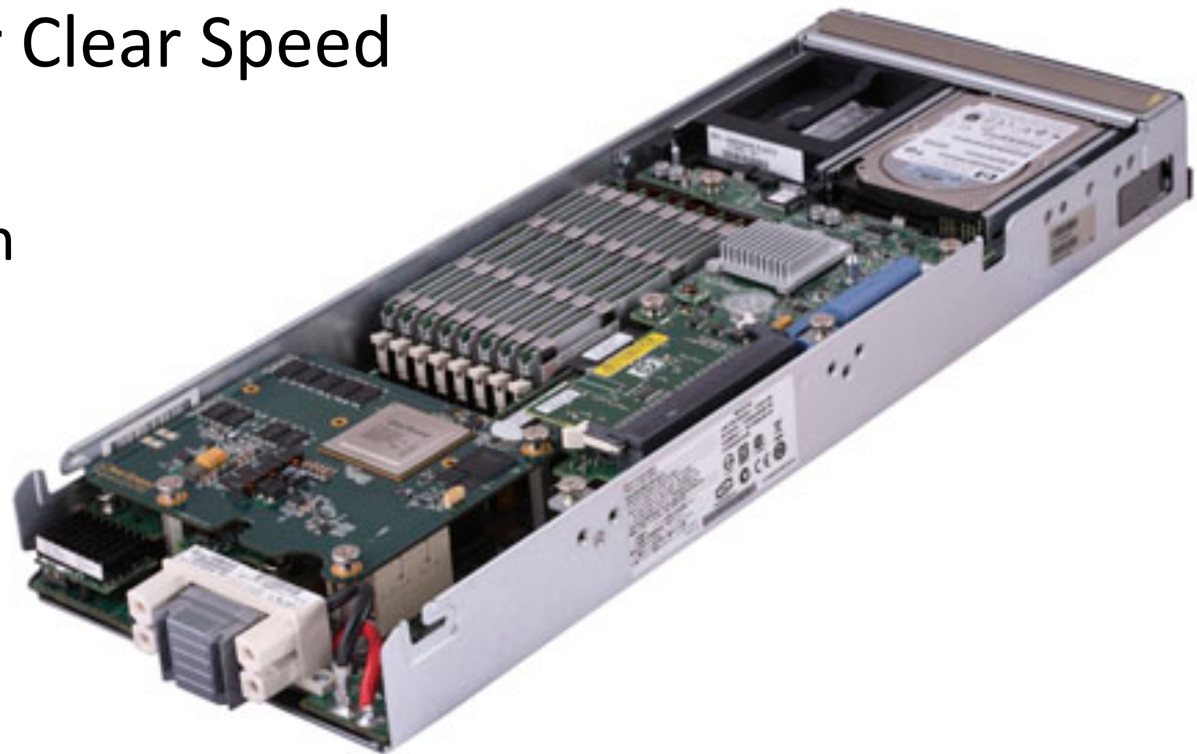
- Grappes (clusters)
- Machines hétérogènes
  - IBM PowerXCell 8i





Différentes architectures

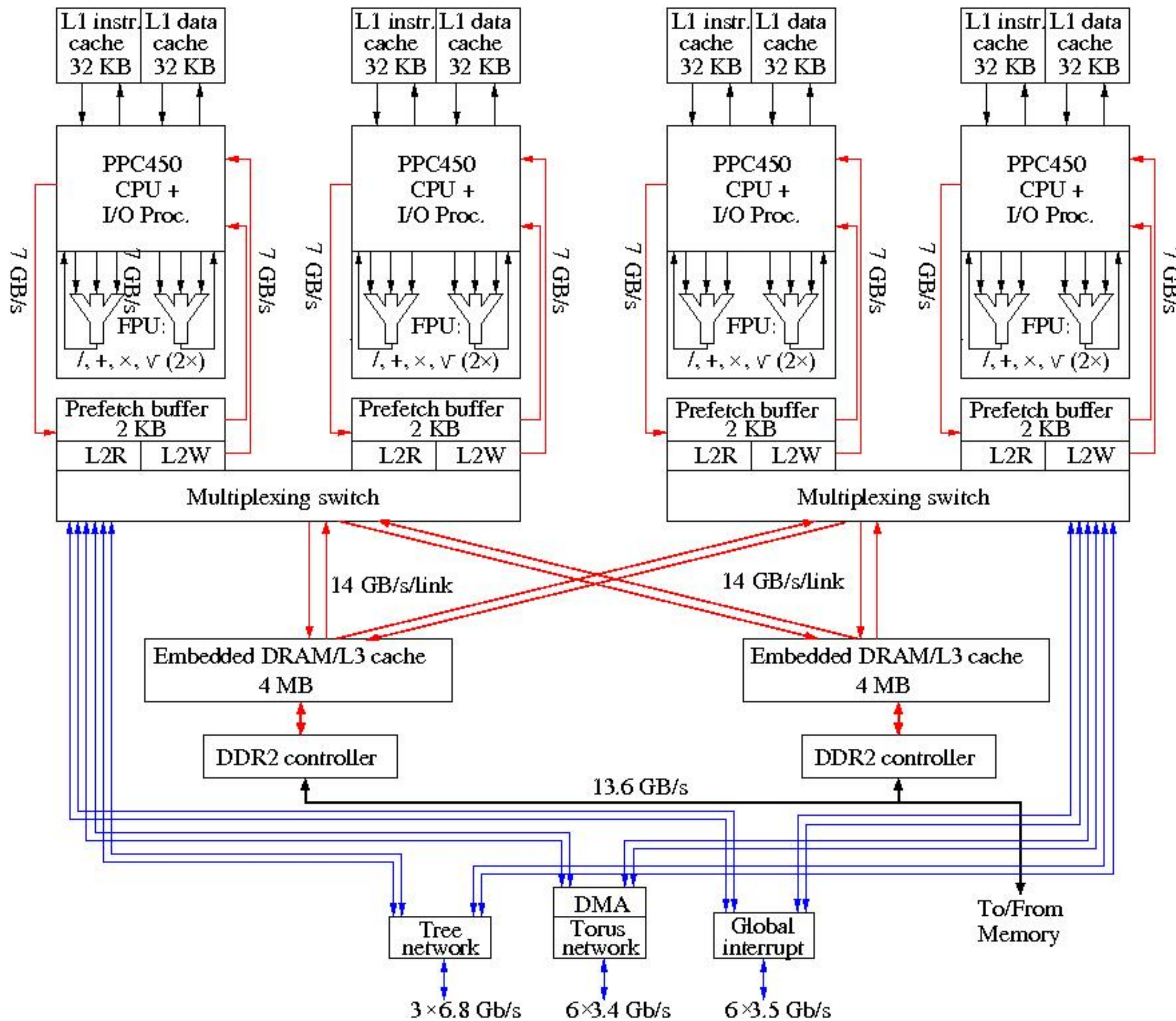
- Grappes (clusters)
- Machines hétérogènes
  - Accélérateur Clear Speed
    - FFT
    - Convolution



Différentes architectures

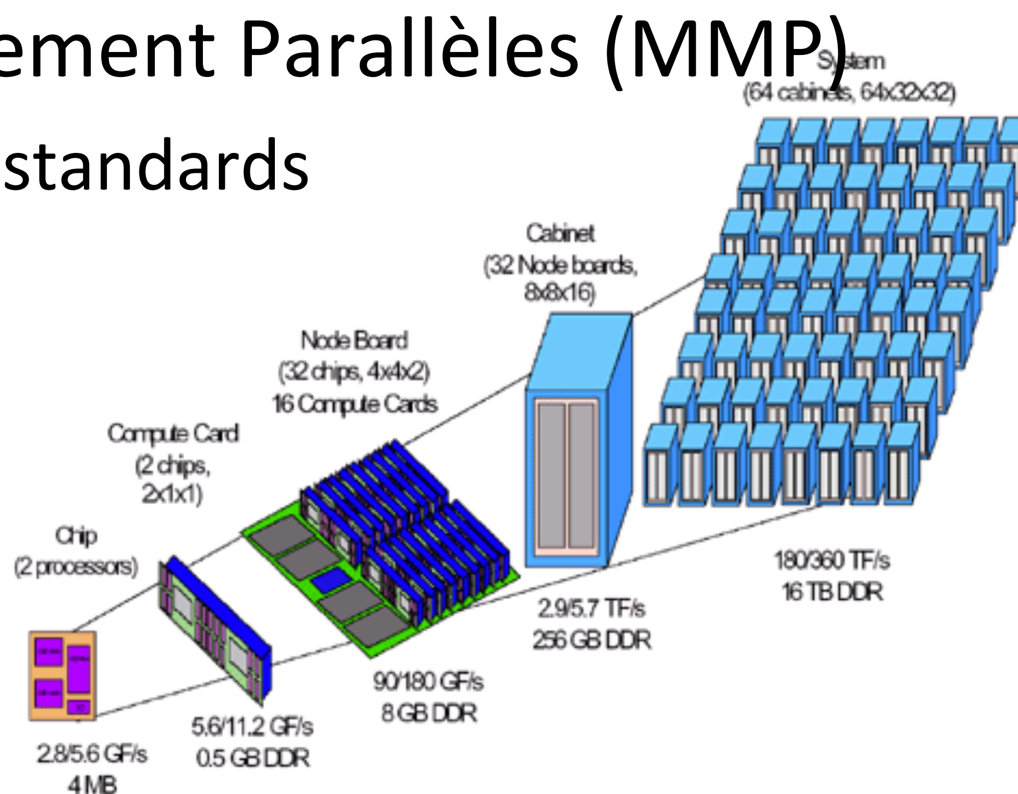
- Grappes (clusters)
- Machines hétérogènes
- Machines Massivement Parallèles (MPP)
  - Composants non standards
    - IBM Blue gene

Blue Gene



Différentes architectures

- Grappes (clusters)
- Machines hétérogènes
- Machines Massivement Parallèles (MMP)
  - Composants non standards
  - IBM Blue Gene



Différentes architectures

- Grappes (clusters)
- Machines hétérogènes
- Machines Massivement Parallèles (MMP)
- Grilles de calcul
  - Partage des ressources
  - Accès sécurisé
  - Gestion des jobs et des données
  - Standardisation nécessaire

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
- …

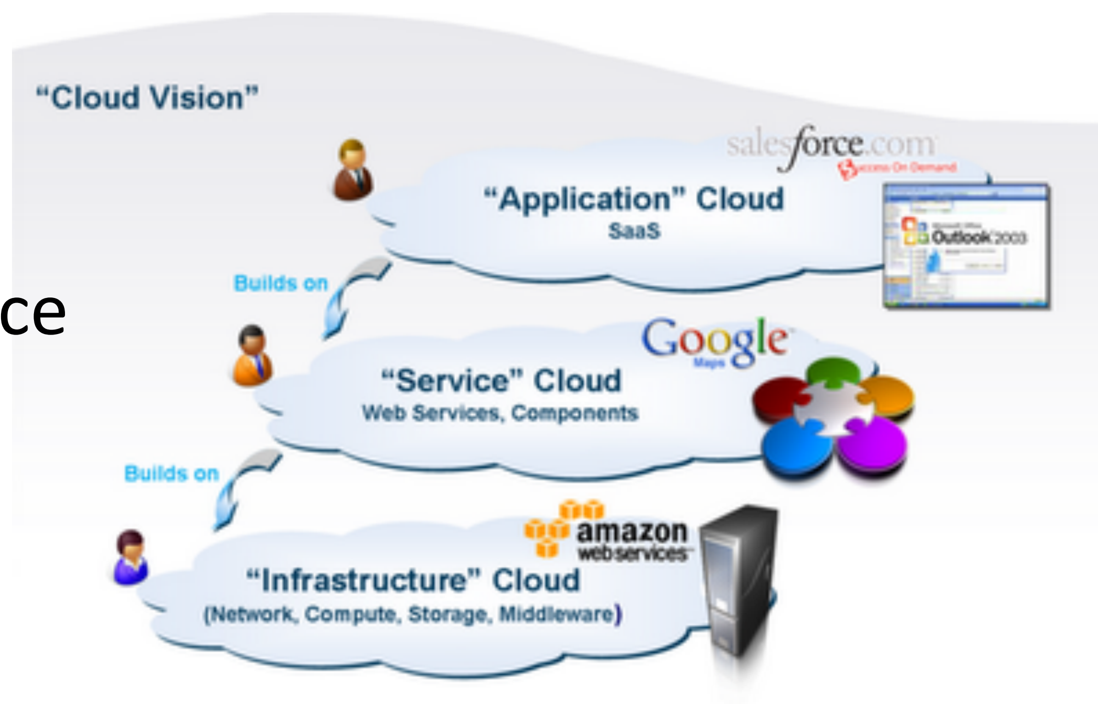
- >250 sites
- 48 countries
- >50,000 CPUs
- >20 PetaBytes
- >10,000 users
- >150 VOs
- >150,000 jobs/day

21:13:50 UTC

GridPP
UK Computing for Particle Physics

Différentes architectures

- Grappes (clusters)
- Machines hétérogènes
- Machines Massivement Parallèles (MMP)
- Grilles de calcul
- Cloud computing
  - Google MapReduce



Différentes architectures

- Grappes (clusters)
- Machines hétérogènes
- Machines Massivement Parallèles (MMP)
- Grilles de calcul
- Cloud computing
- Peer2Peer
  - folding@home
  - BOINC

OS Type	Native TFLOPS*	x86 TFLOPS*	Active CPUs	Total CPUs
Windows	300	300	288271	3664581
Mac OS X/PowerPC	3	3	4005	144010
Mac OS X/Intel	124	124	30281	143636
Linux	258	258	95723	602748
ATI GPU	713	752	5023	145772
NVIDIA GPU	3201	6754	20135	239452
PLAYSTATION®3	800	1688	28358	1086691
Total	5399	9879	450691	6026890

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,…
- 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,…
- Approches implicites → Facilité du développement
  - « mémoire virtuellement partagée »
  - « système distribué à image unique »

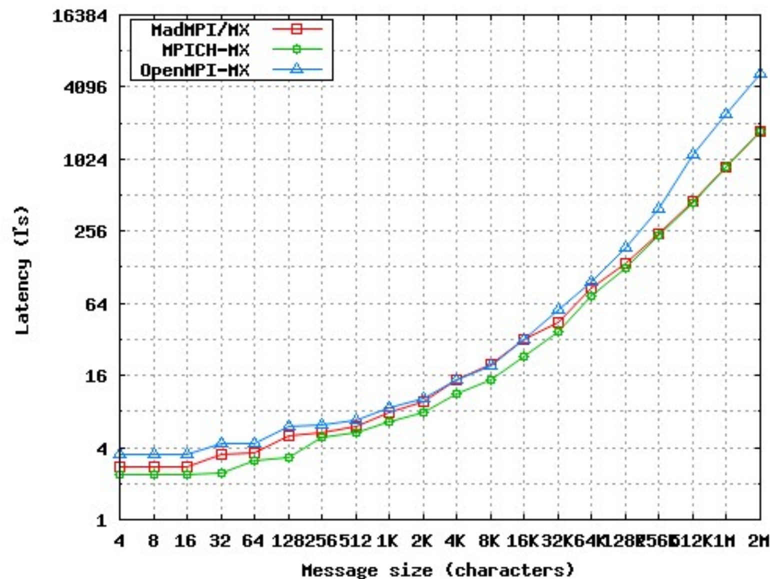
Paradigme passage de message

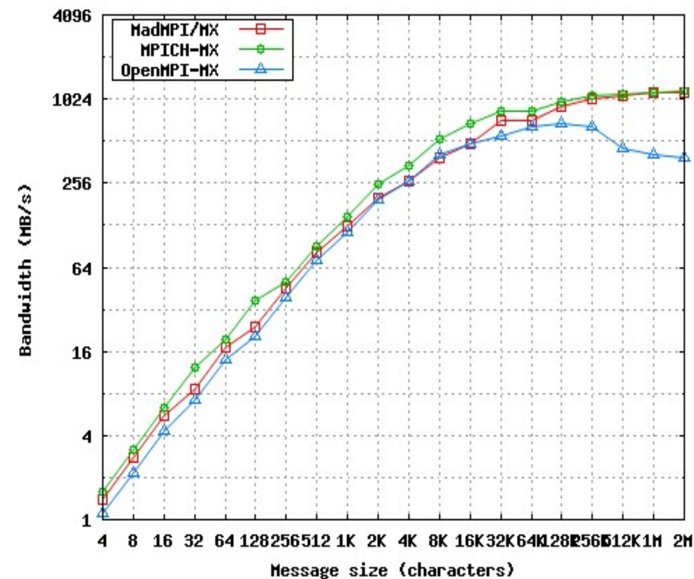
- Participation de l'émetteur et du récepteur
- Communication point à point :
  - Send(destinataire, buffer, taille)
  - Receive(émetteur, buffer, taille)
- Communication collective
  - Broadcast(groupe,émetteur, buffer, taille)

Objectifs des communications dans le cadre HPC

- Transmission performante – réseaux rapides
  - Faible latence (ordre de la microseconde)
  - Gros débit (ordre de du Go/s)

  Expérience du ping-pong :





Objectifs des communications dans le cadre HPC

- Transmission performante
  - Permettre le calcul durant les communications
    - Ne pas chercher à synchroniser le send et le receive
    - Découpler les appels à send/recieve de l'émission / réception physique des données

    ➔ Communication en arrière-plan

Objectifs des communications dans le cadre HPC

- Permettre le calcul durant les communications
  - Aspect bloquant des primitives de communication
    - Bloquant
      - Sortir de l'appel que lorsque le buffer est disponible
    - Non bloquant
      - Sortir de l'appel même si le buffer est indisponible
      - Nécessite une primitive pour tester / attendre la disponibilité du buffer

Objectifs des communications dans le cadre HPC

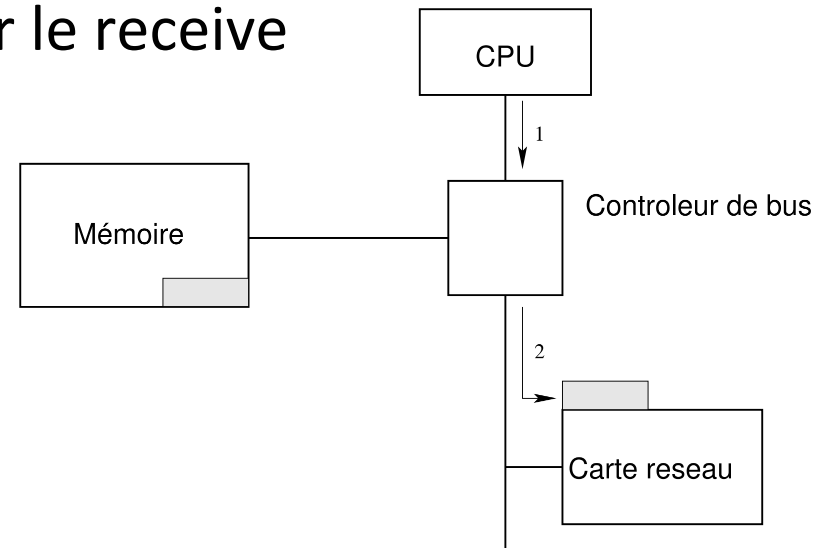
- Permettre le calcul durant les communications
  - Aspect bloquant des primitives de communication
  - Aspect synchronisant des primitives de communication
    - Synchrone
      - Sortir de l'appel que lorsque le récepteur a appelé receive
    - Asynchrone
      - Ne pas tenir compte du récepteur

Permettre le calcul durant les communications

- Trois principales techniques de communication exploitée
  - 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 les 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



200

0

4 1K 32K 1M

Taille des messages (octets)

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

CPU

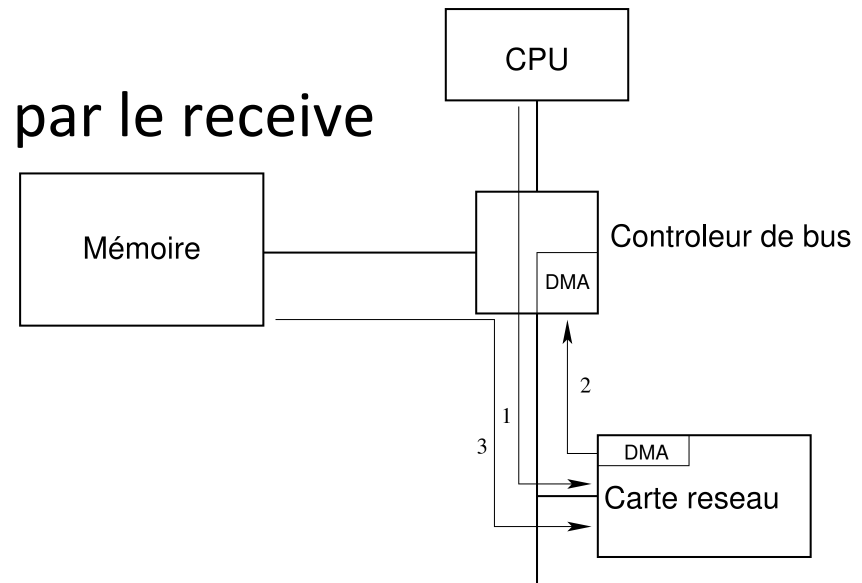
1

Mémoire

Controleur de bus

2

Carte reseau



DMA zéro copie réseaux rapides

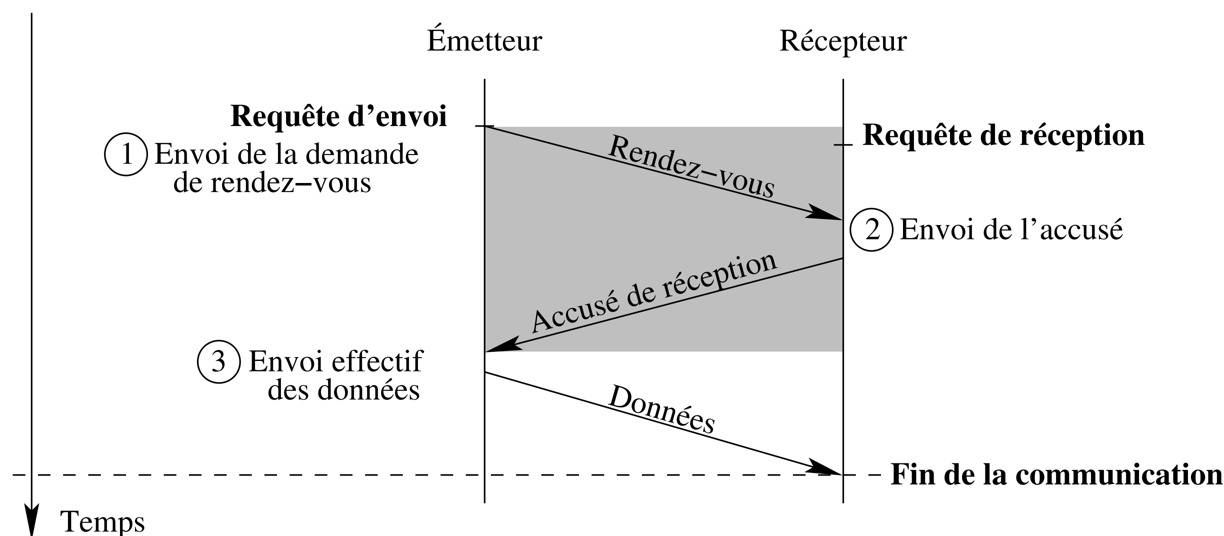
– Le processeur émetteur/récepteur informe la carte du transfert à réaliser

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

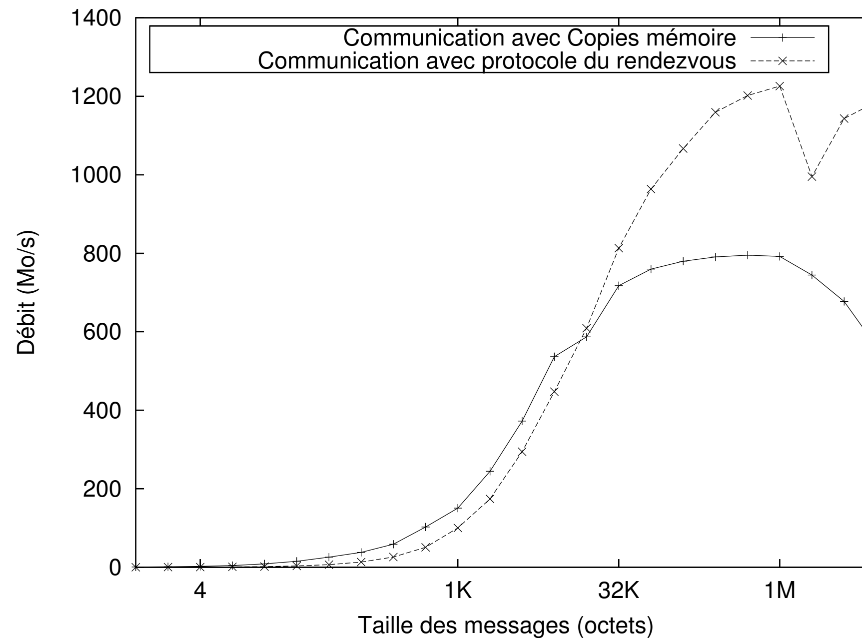
- Le processeur émetteur/récepteur informe la carte du transfert à réaliser

① La carte émettrice demande un rendez-vous à la carte réceptrice

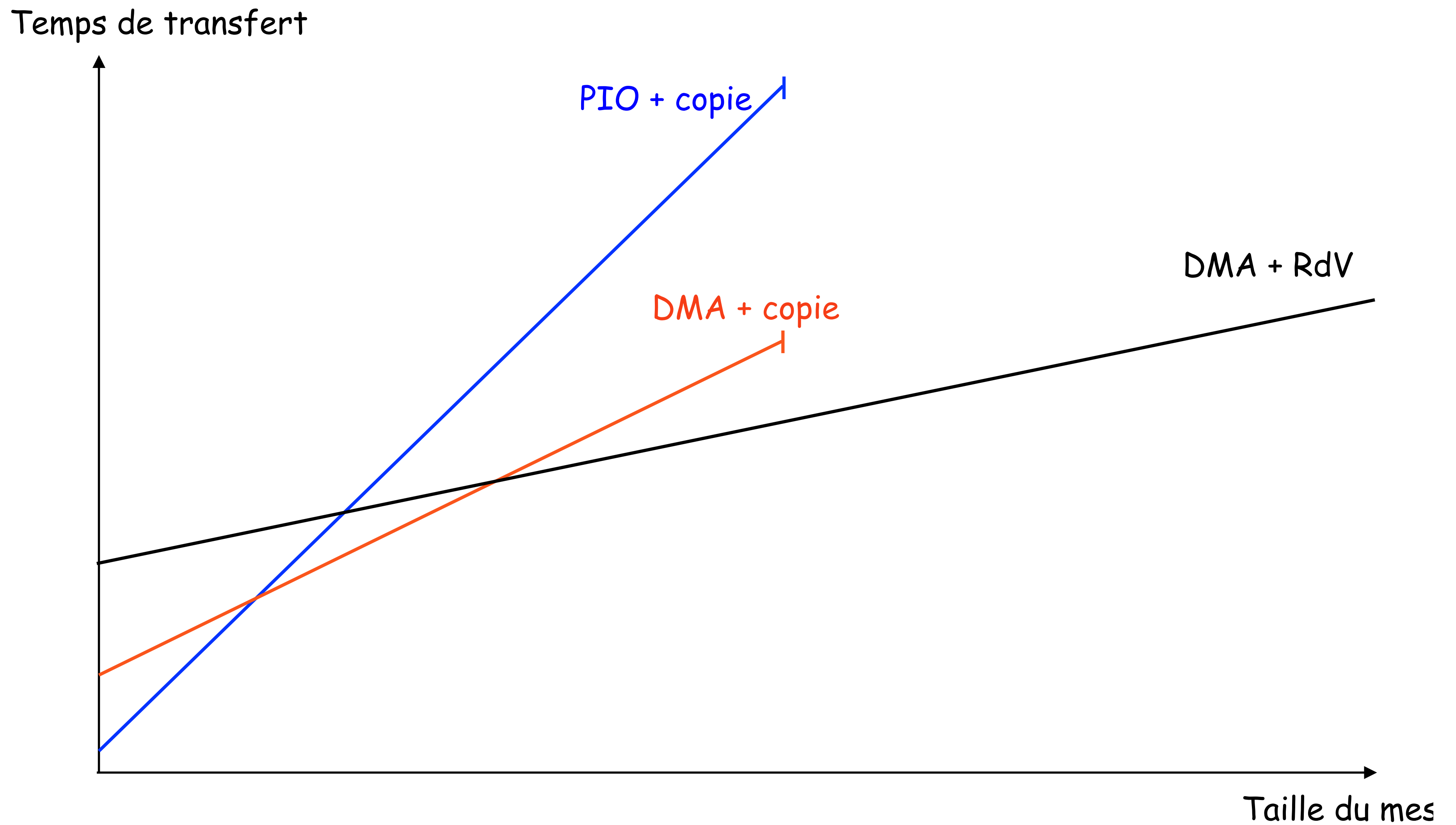
② La carte réceptrice prépare son DMA puis accorde le rendez-vous

③ La carte émettrice demande le transfert au contrôleur DMA

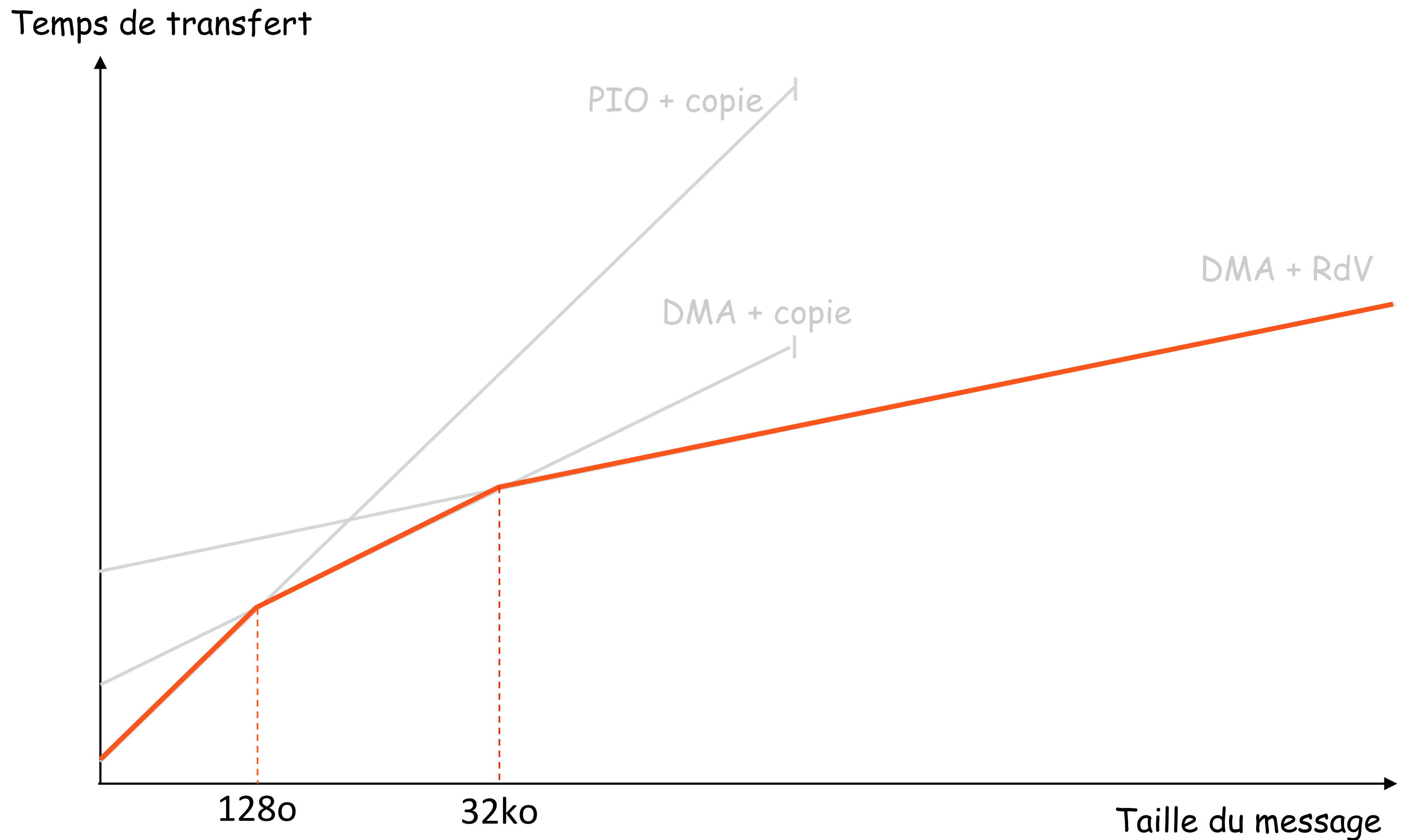
- Le transfert est réalisé via les buffers internes aux cartes



Performances

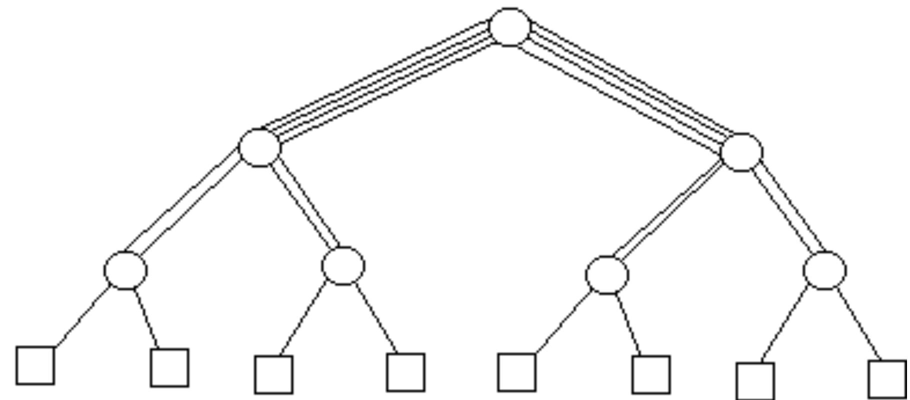


Performances



Objectifs des communications dans le cadre HPC

- Transmission performante
- Parallélisme des communications
  - Capacité a transmettre beaucoup de message en parallèle
  - *Bissection bandwidth*
    - *Partition du réseau en deux groupes pour les faire communiquer*



Objectifs des communications dans le cadre HPC

- Transmission performante
- Parallélisme des communications
- Fiabilité des communications
  - Matériel fiable
    - Correction à la volée des erreurs
  - Routage simplifié
    - Gestion de la congestion avant tout par le surdimensionnement
  - Sécurisation des communications
    - Isoler la grappe ou la sous grappe, les sessions

→ Éviter la pile TCP / IP au sein des clusters

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
- Normalisé depuis 1994
- Approche processus
  - Pas de false sharing, pas de problème de cohérence mémoire

- Plus souvent utilisé pour programmer les machines à mémoire commune qu'OpenMP ;-)
  - Qui peut le plus peut le moins
  - Les programmeurs cherchent à minimiser les communications
    - Effort d'optimisation plus important que sur OpenMP

Voir https://computing.llnl.gov/tutorials/mpi

Les points forts de MPI

- Standard
- Portable, facile à utiliser
  - Les messages ne se doublent pas (ordre fifo)
  - Mais difficile à débugger
- Simplifie la programmation efficace sur grappe
  - Sélectionne le bon mode de communication
    - Zéro copie
    - Pilote adapté (Mémoire partagée, infiniband, TCP/IP)
  - Opérations collectives (MPI-2 1996)
    - Distribution suivant la topologie (arborescent)
- Possibilité d'utiliser des threads
    - MPI + OpenMP
- 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
```

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);

        }
```

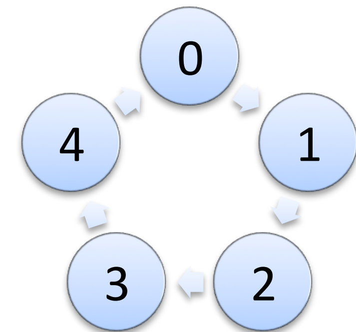
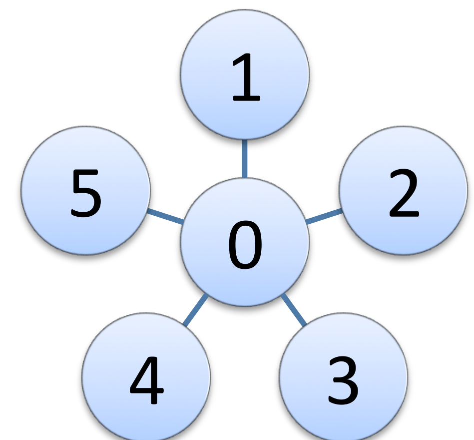


Schéma de calcul Maitre / Esclave
Jeton centralisé

```
if (rank == 0) {
        for(i = 0; i < 3*(size-1); i++) {
         MPI_Recv(&token, 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(&token, 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);
         //sleep(1);
         MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);

    }
```



Les modes de communication

- 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)

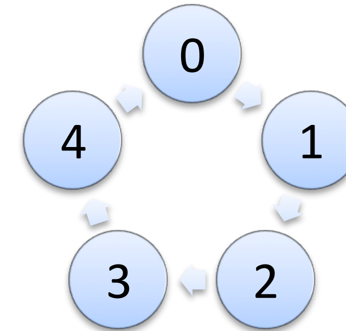
  Il faut faire attention à ce que le buffer ne soit pas être (ré-)utilisé trop tôt en émission comme en réception

  MPI_Wait (&request,&status)

  MPI_Test (&request,&resultat,&status)

  MPI_Waitall (count,&array_of_requests,&array_of_statuses)

Recouvrement calcul communication envoyer son identité à ses voisins




```
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

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

prev = (rank-1)%numtask;
next = (rank+1)%numtask;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

   { do some work }

MPI_Waitall(4, reqs, stats);
MPI_Finalize();
```

échange de deux buffers

Les processus 0 et 1 doivent placer dans le buffer b le contenu du buffer a de l'autre processus

```
MPI_send(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
MPI_recv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
```

échange de deux buffers

```
MPI_send(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
MPI_recv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
```

➔ Possibilité d'interblocage

Si taille a et b est suffisamment grande

- Attente de la libération d'un buffer interne pour une copie
- Attente de l'acquittement en mode rdv

échange de deux buffers

```
If(rank == 0)
{
 MPI_recv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
 MPI_send(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
}
else
{
 MPI_send(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
 MPI_recv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD);
}
```

échange de deux buffers

```
MPI_Request Sreq, Rreq;
MPI_Status Ssta, Rsta;
int a, b;
…
MPI_Irecv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD, &Rreq);
MPI_Isend(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD, &Sreq);
…
// calcul
...
MPI_Wait(&Sreq, &Ssta);
MPI_Wait(&Rreq, &Rsta);
…
```

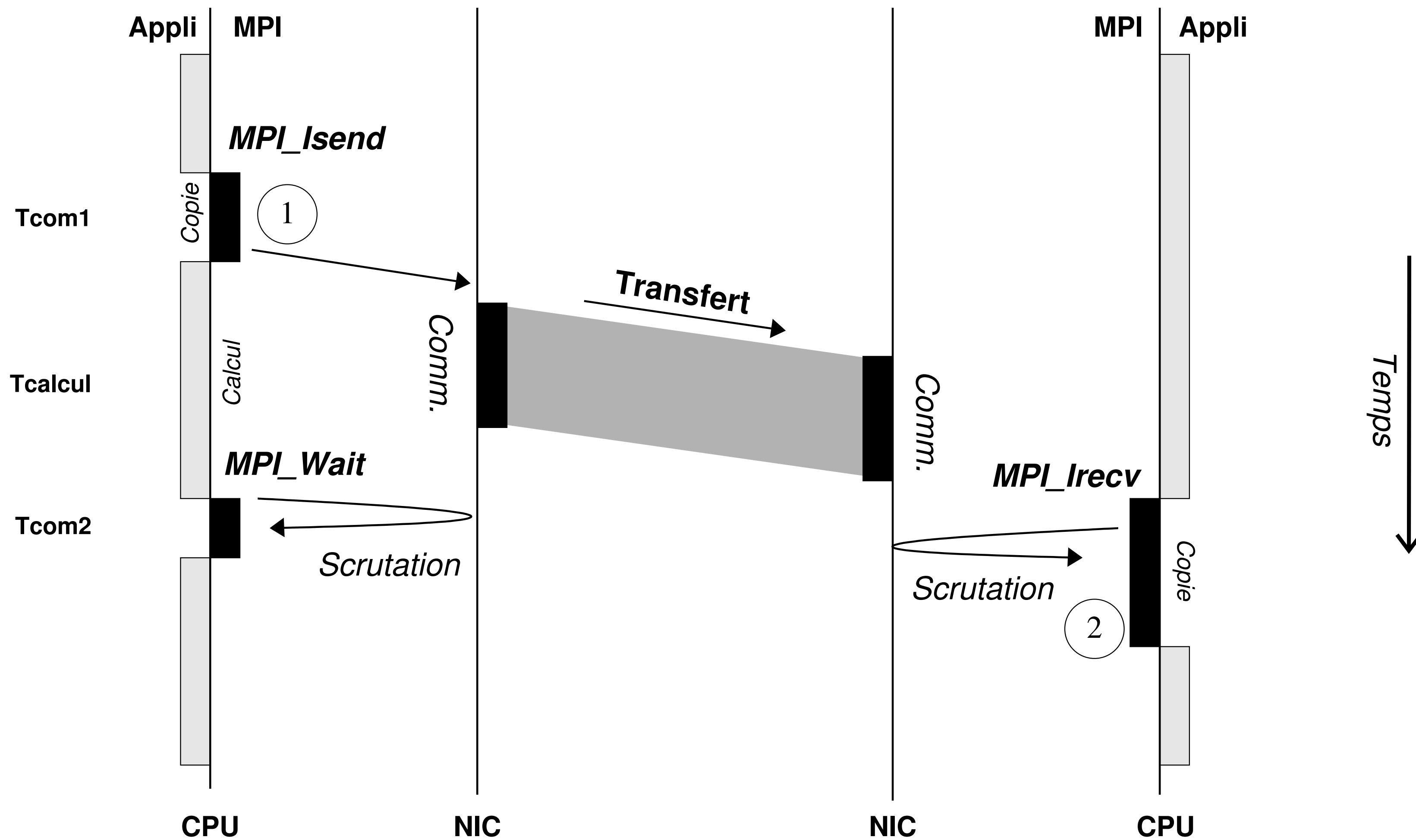
échange de deux buffers

```
MPI_Irecv(&a, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD, &Rreq);
MPI_Isend(&b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD, &Sreq);
...
MPI_Wait(&Sreq, &Ssta);
MPI_Wait(&Rreq, &Rsta);
…
```

Alternative :

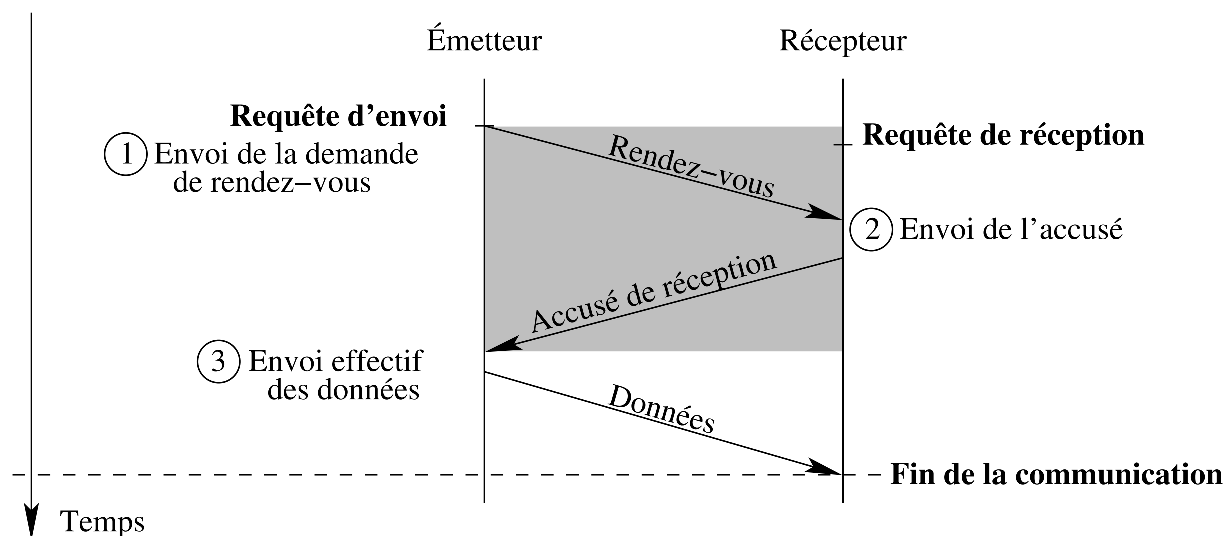
```
int MPI_Sendrecv(
        void *sendbuf, int sendcount,MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status )

int MPI_Sendrecv(&a, taille, MPI_INT, 1-rank, tag,
                 &b, taille, MPI_INT, 1-rank, tag, MPI_COMM_WORLD, &req);
```

Recouvrement calcul communication comportement avec copie



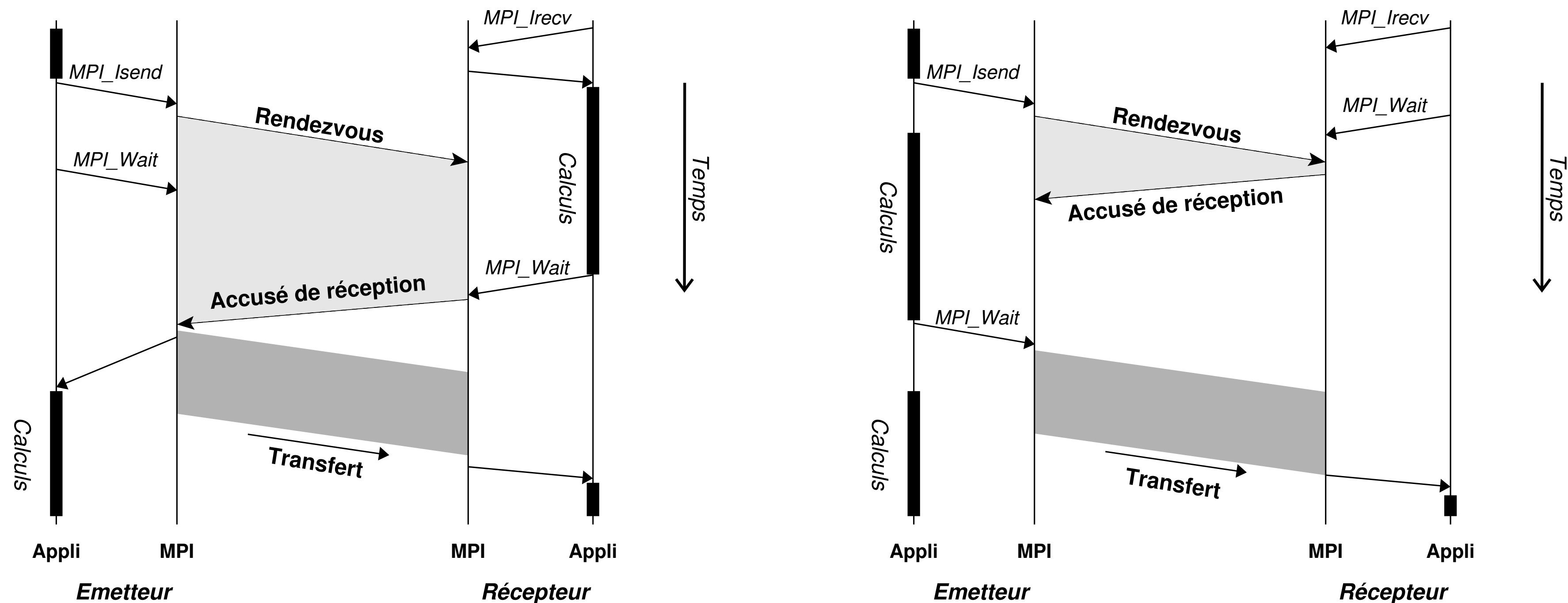
Recouvrement calcul communication
Rappel : DMA zéro copie sur réseau rapide

– 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



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.

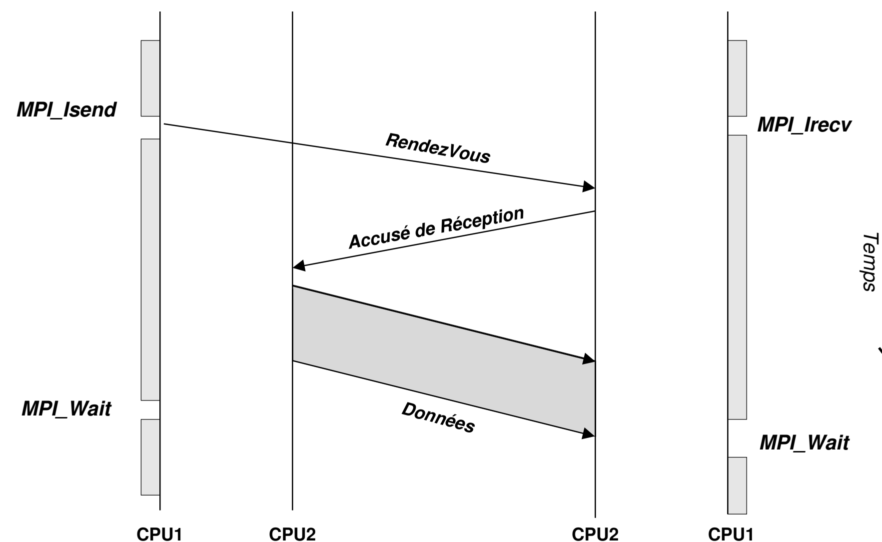


Émission bloquée en attente alors qu'émetteur et récepteur sont prêts

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)



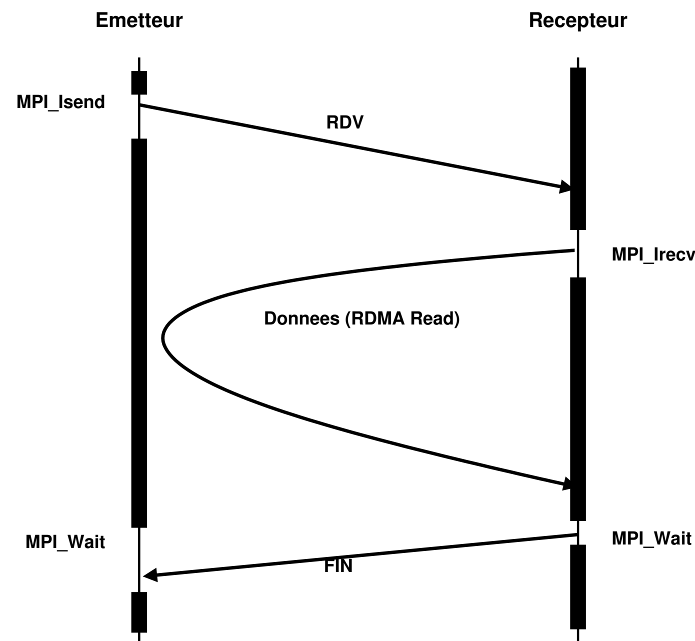
Solutions au problème du rendez-vous

- 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



Thèse de François Trahay

Autres primitives de communication

- Ssend termine lorsque le buffer est réutilisable lorsque la réception a commencé et que le buffer peut être réutilisé (mode synchrone)

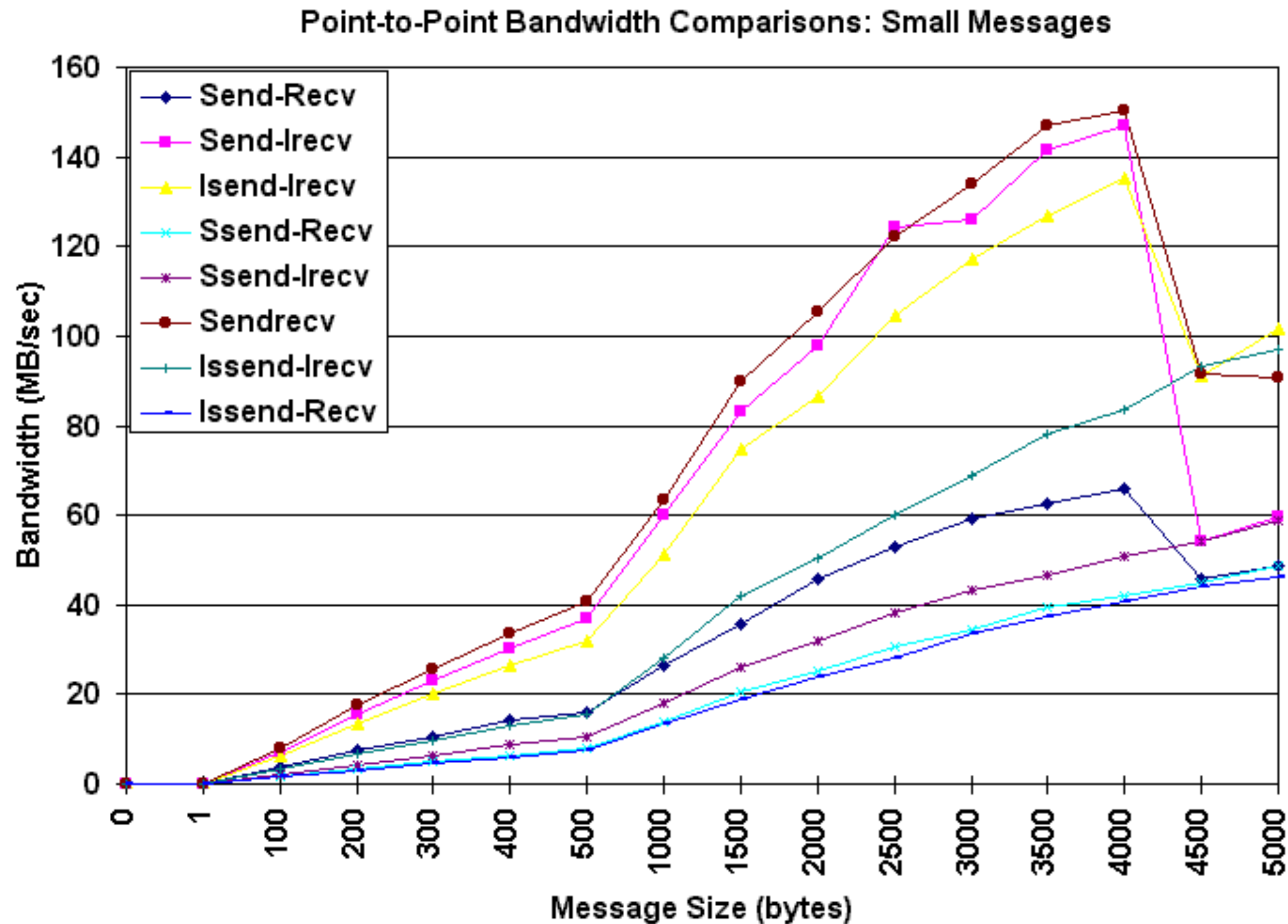
- Rsend (Ready send) termine lorsque le buffer peut être réutilisé
  - Le recv correspondant est supposé préalablement posté
  - Évite la demande de rendez-vous

- Bsend termine après recopie des données dans un buffer
  - Buffer_attach(void*,size_t) / Buffer_detach pour allouer *le* buffer du processus

- Irsend, Ibsend, Issend
  - influent sur l'interprétation du test/wait qui indique la fin de la réception

Performances (LLNL)

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



Performances (LLNL)

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

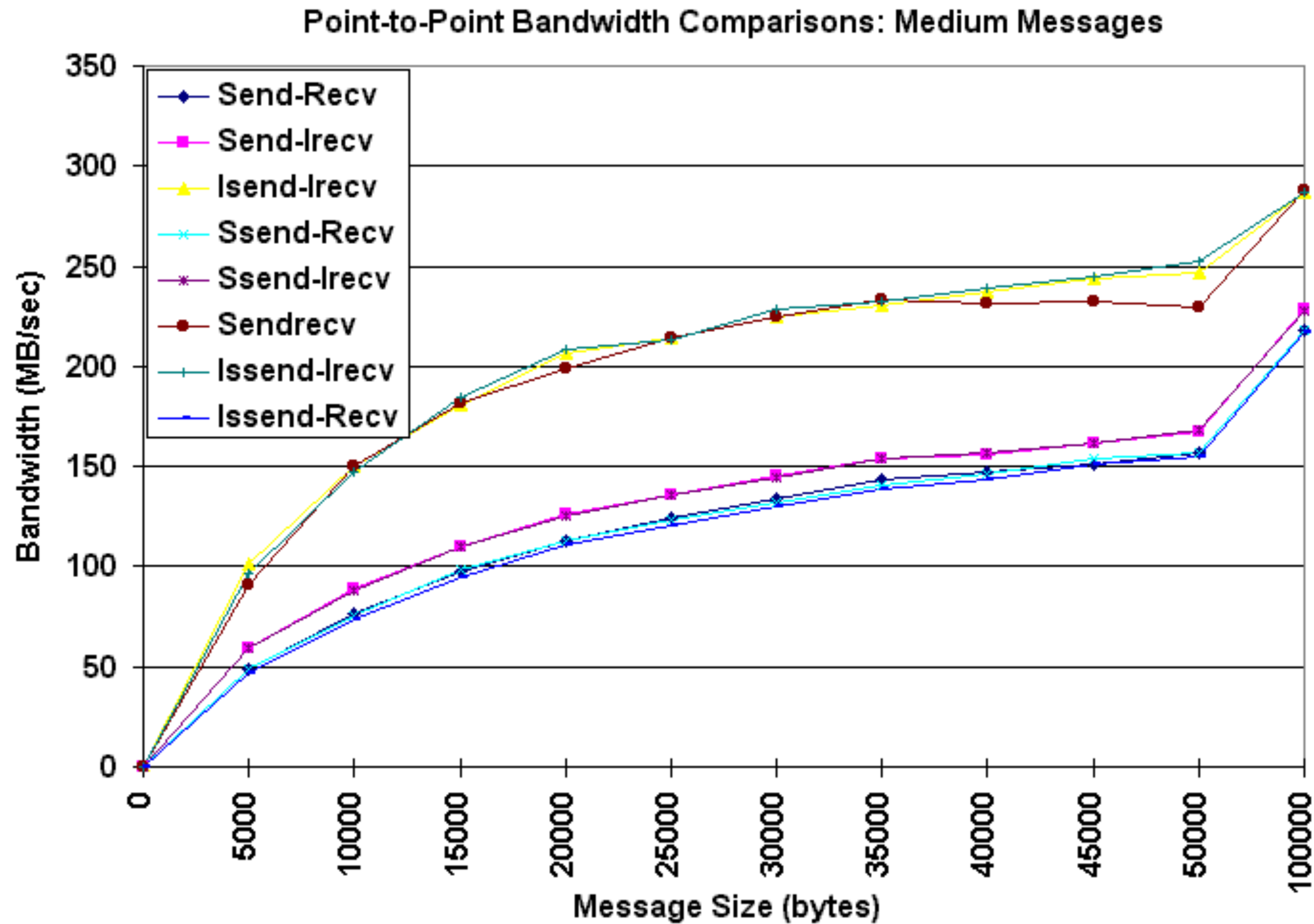
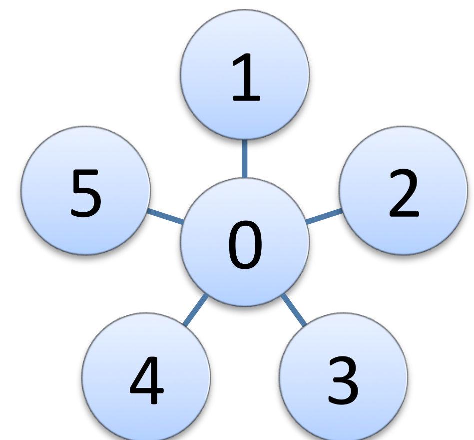


Schéma de calcul Maitre / Esclave
Ready send

```
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);

                }
```



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, &flag, &status)
  - Attendre un message sans en effectuer la réception
  - flag = un message répondant aux contraintes précisées est disponible
  - La variable status est renseignée
  - Ex. : dimensionner un buffer de réception
  - MPI_Iprobe()

Optimisations
Communications persistantes

Enregistrer des requêtes pour les rejouer plusieurs fois

- Économise du temps sur la création de la requête
- Factorise du code

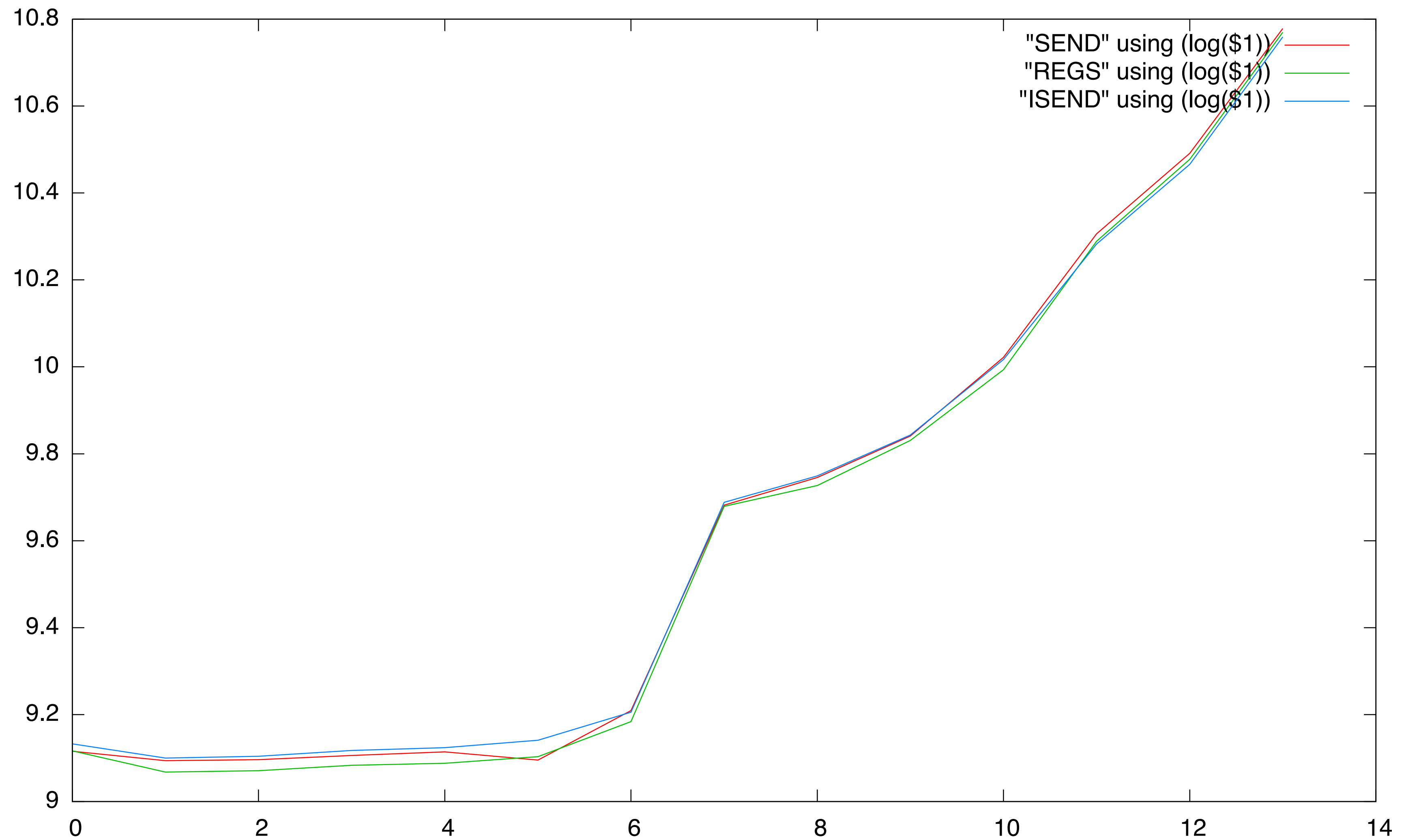
- int MPI_Send_init(buf, count, datatype, dest, tag, comm, &request)
- int MPI_Recv_init(buf, count, datatype, source, tag, comm, &request)

- 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)

Expérience sur infini

~200 cycles de gain sur un aller/retour



Optimisation création de datatypes

- Interface pour construire des paquets de données
  - Regrouper de façon automatisée des données
  - Économiser des requêtes
- Empaqueter / dépaqueter des données (PVM)
  - Type MPI_PACKED
  - MPI_PACK() / MPI_UNPACK()
    - Recopie dans un buffer
- Définir un type dérivé (datatype) pour éviter si possible les octets
  - à partir des types de base
    - MPI_CHAR MPI_SHORT MPI_INT MPI_LONG MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT MPI_UNSIGNED_LONG MPI_UNSIGNED MPI_FLOAT MPI_DOUBLE MPI_LONG_DOUBLE MPI_BYTE MPI_PACKED
  - En créant des structures (struct)
  - En juxtaposant plusieurs fois un même datatype (contiguous)
  - En extrayant itérativement et régulièrement des données d'un datatype (vector)
  - En extrayant itérativement des données suivant un tableau d'index certains éléments d'un tableau (indexed)

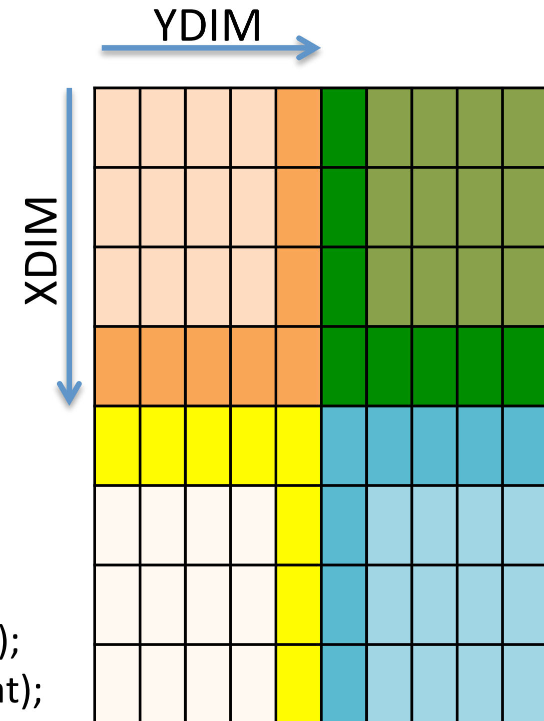
Exemple de construction d'un datatype

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

MPI_Type_vector(count, blocklenth, stride, oldtype, &newtype)

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

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


MPI_Isend(&tab[out][1][1],1,colonne_t,Gauche,1,MPI_COMM_WORLD);
MPI_Isend(&tab[out][1][YDIM],1,colonne_t,Droite,1,MPI_COMM_WORLD);
MPI_Irecv(&tab[in][1][1],1,colonne_t,Gauche,1,MPI_COMM_WORLD,&etat);
MPI_Irecv(&tab[in][1][YDIM],1,colonne_t,Droite,1,MPI_COMM_WORLD,&etat);
```



- 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…

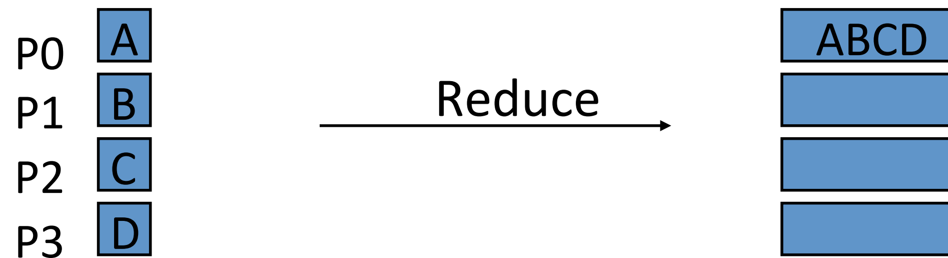
Opérations & communication collectives

- Toutes les opérations collectives sont bloquantes (pour le moment)
  - 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

MPI_Barrier(communicator)

Opérations collectives

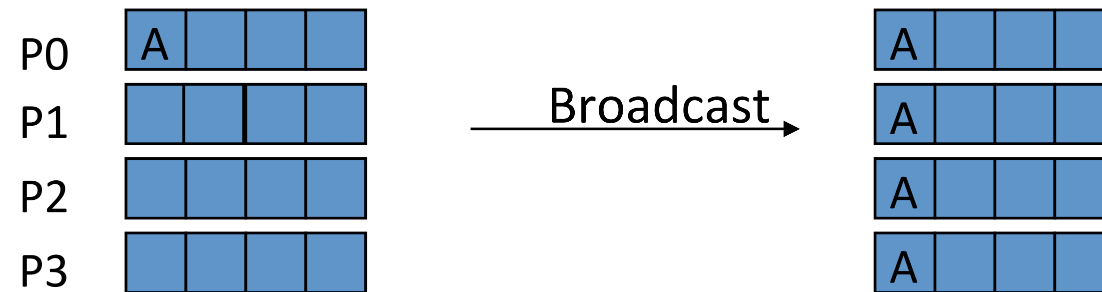
- MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
  - MPI_MAX, MIN, SUM, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC
  - MPI_Op_create(function, commute, &op)
  - MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
    - Non déterminisme => au final tous les nœuds n'ont pas forcément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

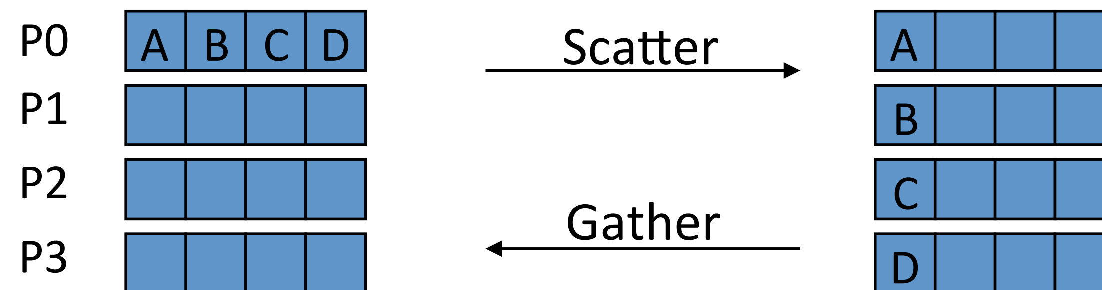


Communications collectives

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

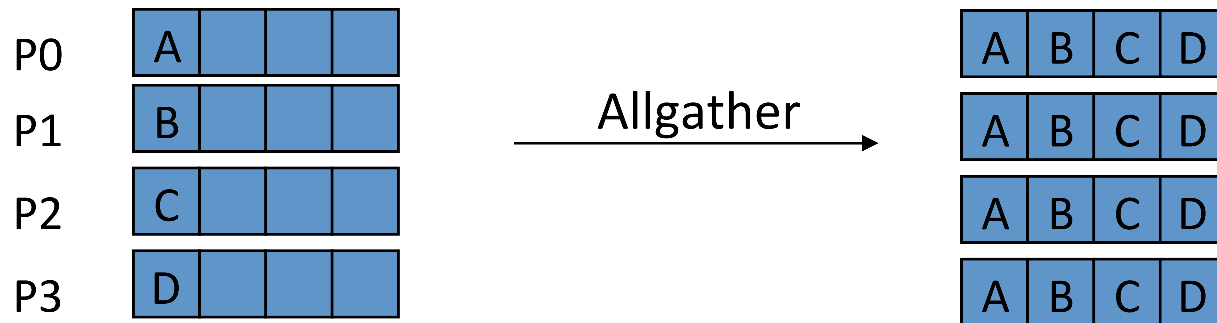
Bloquant mais non synchronisant





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

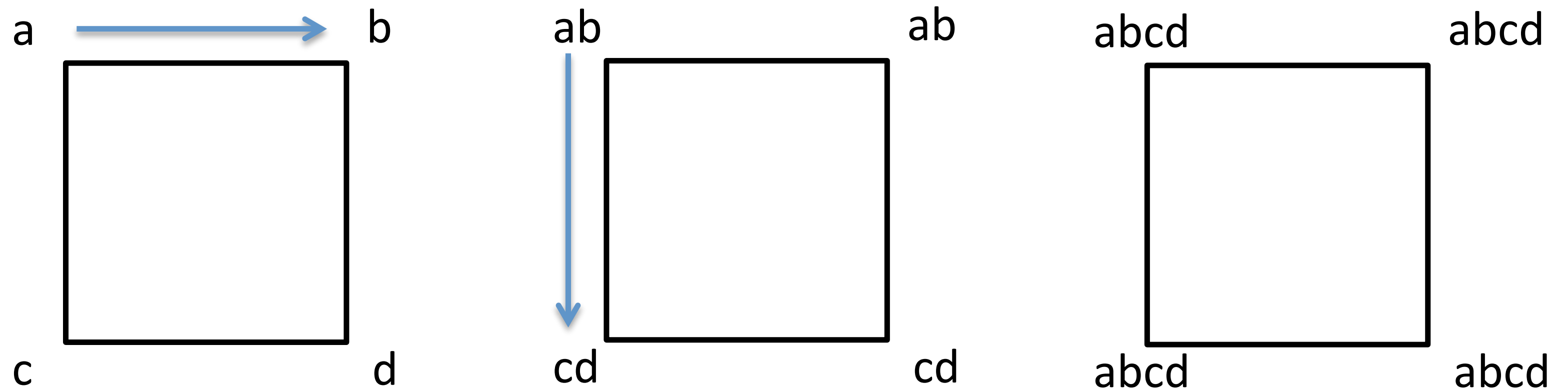
Communications collectives



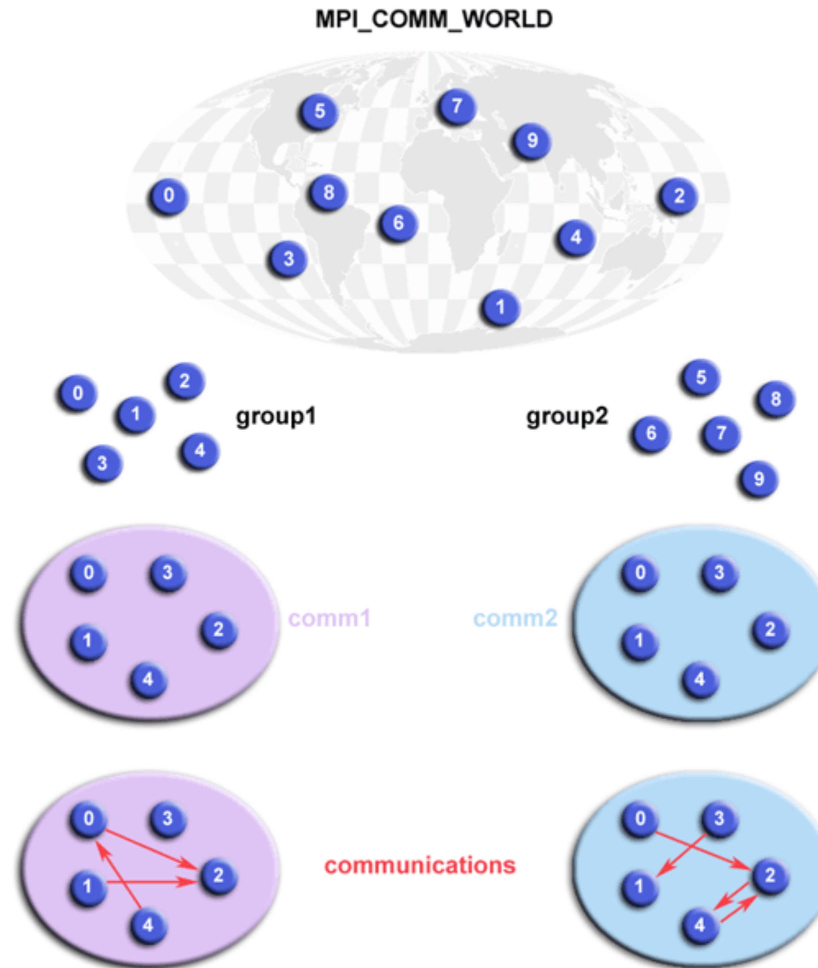
Protocole d'échange total
Hypercube

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



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();
}
```

Et bien encore plus… MPI 2

- MPI permet de
  - Mettre en œuvre une topologie virtuelle
    - Permet au runtime 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
  - Utiliser des threads
    - 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