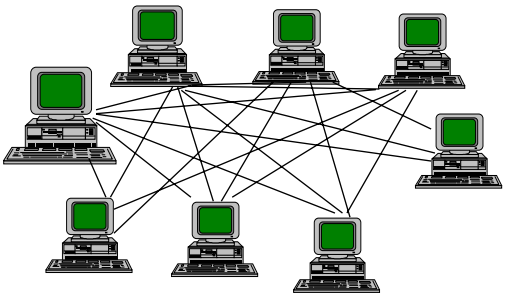
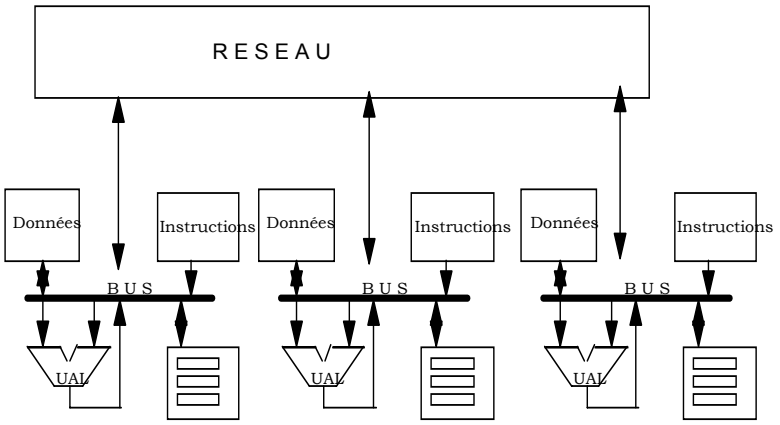
Canaux entre mémoires distribuées



Réseau d'interconnexion de processeurs distants

Architecture



Communication dans des systèmes client-serveur

- Sockets
  - Point de communication identifié par une adresse IP et un N° de port : le serveur attend des requêtes provenant de clients en scrutant sur le N° de port ; quand un process client émet une requête de connexion, l'hôte du client lui alloue un N° de port > à 1024.
  - Ex : un client sur un hôte Y avec l'adresse IP 192.56.4.40 souhaite établir une connexion sur un serveur Web (port 80) à l'adresse 162.25.20.5 ; à la machine Y sera assigné le port 1225. La connexion consistera en une paire de sockets:
    (192.56.4.40:1225) sur Y et
    (162.25.20.5:80) sur le serveur Web
  - Toutes les connexions doivent être uniques

Exemple : les sockets en Java

- Orientés connexion (TCP) → Socket class
- Sans connexion (UDP) → DatagramSocket class
- Multi connexion → MulticastSocket class
  sous classe de la précédente

Exemple

```
import java.net.* ;
import java.io.* ;

public class DateServer
{
  public static void main (String[ ] args) {
     try {
        ServerSocket sock = new ServerSocket (1250) ;
           // attente de connexion
           while (true) {
              Socket client = sock.accept() ;
              PrinterWriter pout = new PrinterWriter (client.getOutputStream(), true);
           // écriture de la date sur le socket
           pout.println(new java.util.Date().toString()) ;
           // fermeture du socket et poursuite
           // ecoute pour connexion
           client.close() ;
         }
     }
     catch (IOException ioe) {
         System.err.println(ioe) ;
     }
}
```

Création d'un socket : écoutera sur le port 1250

Ecoute sur le port et bloque

Le serveur crée un objet qu'il utilisera pour communiquer avec le client

Envoie la date au client avec la méthode println

Ferme cette connexion


```
import java.net.* ;
Import java.io.* ;

public class DateClient
{
  public static void  main (String [ ] args) {
     try {
           // connexion au socket du serveur
           Socket sock = new Socket ("127.0.0.1 ", 1250) ;
           InputStream in = sock.getInputStream() ;
           BufferedReader bin = new BufferedReader( new InputStreamReader(in));
           // lecture des données sur le socket
           String line ;
           while (( line = bin.readLine()) != null)
              System.out.println(line) ;
           // fermeture de la connexion avec le socket
           sock.close() ;
  }
  catch (IOException ioe) {
           System.err.prinltn(ioe) ;
  }
}
```

Création du socket et demande de ctx sur le port 1250
IP Autoréférence

Lecture sur le socket après ctx

Fermeture du socket et sortie

Le Remote Procedure Call (RPC)

- Echange de messages structurés
- Chaque message est envoyé à un démon RPC écoutant sur un port (identificateur du service proposé) du système distant, avec un identifieur de fonction à exécuter et des paramètres.
- La fonction est exécutée et un autre message porte les résultats en retour.
- Le mécanisme du RPC cache les détails de la communication en fournissant un stub du côté client : il localise le port sur le serveur et récupère les paramètres *(marshalling)* : fabrication des paquets pour transmettre les données.
- Un mécanisme identique existe du coté serveur.

Sémantique de l'appel

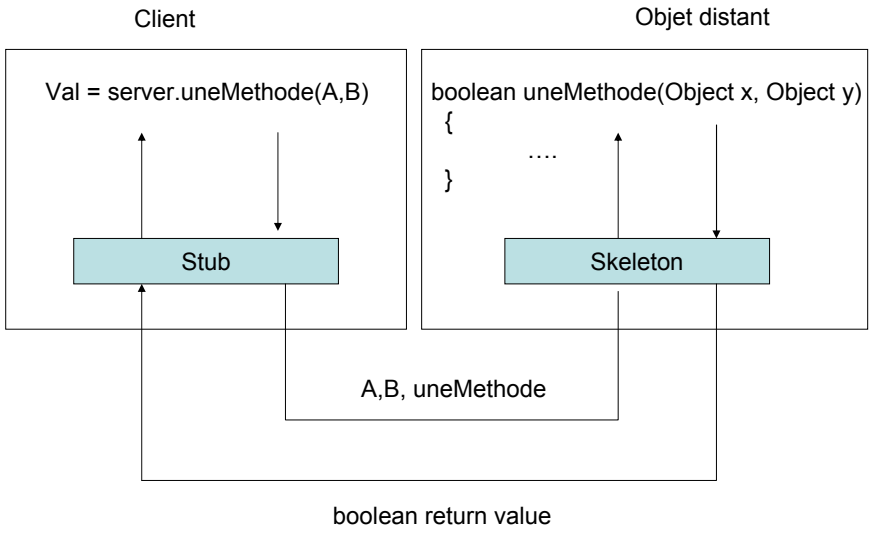
- Les messages doivent être échangés au moins une fois ou exactement une fois : nécessité d'attacher une estampille à chaque message. Le serveur gère un historique de tous les messages reçus. Le client peut envoyer plusieurs fois et être assuré qu'il ne sera exécuté qu'une fois par le serveur.
- Pour « exactement une fois » le serveur doit en plus acquitter le message reçu : le client peut renvoyer le message tant qu'il n 'a pas reçu de Ack du serveur.

Association client-serveur

- Prédéterminé par une adresse fixe de port : l'adresse est « en dur » dans le code
- Association dynamique par un mécanisme de rendez vous : un démon est actif sur un port RPC donné : un client envoie un message contenant le nom du RPC au démon demandant l'adresse du port du RPC dont il a besoin : le N° de port est renvoyé, et le RPC peut être envoyé à cette adresse jusqu'à la fin (normale ou anormale) de la tâche.

RMI Remote Method Invocation en Java

- RMI est un mécanisme de RPC en Java. Il permet d'invoquer une méthode sur un objet distant (une autre JVM)
- Un RPC fournit des appels vers des procédures ou des fonctions : RMI supporte l'invocation de méthodes
- RPC se limite à des paramètres ; RMI autorise des objets en paramètres.
- RMI utilise des stubs et des skeletons pour rendre transparent ces invocations :
  - Un stub est un proxy pour l'objet distant : il réside sur le client.
  - Le skeleton est responsable pour le dépliage des paquets transmis et pour invoquer la méthode demandée ; il fabrique des paquets avec le résultat et passe les données au client.



Règle de passage de paramètres

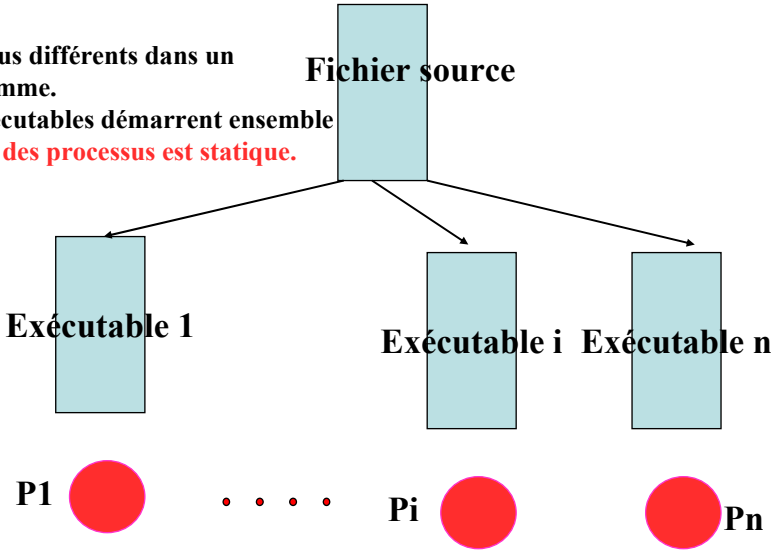
- Paramètres objets locaux
  - Passage par copy en « sérialisant » l'objet : un flot d'octets est transmis
- Paramètres objets distants
  - Passage par référence en implémentant l'interface java.io.Serializable

Passage de messages par l'utilisation de bibliothèque dédiée

```
#include ''mpi.h''

$ mpicc –c foo.c

$ mpirun –np  4 a.out
```


Le modèle SPMD

Des processus différents dans un seul programme.
Tous les exécutables démarrent ensemble
La création des processus est statique.

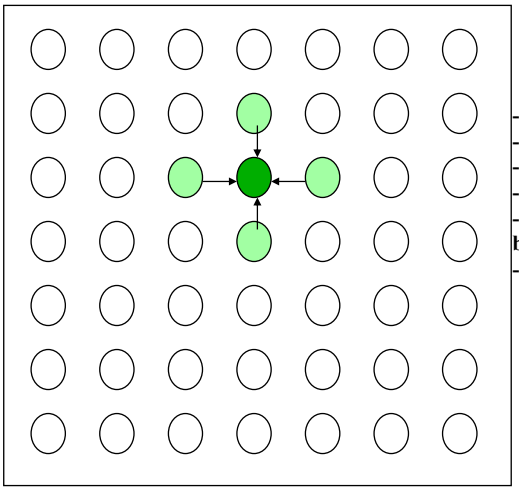


Primitives de base

NOM	Syntaxe	Fonction
CREATE	CREATE (procedure)	Crée un processus dont le main est procedure
SEND	SEND(src_addr, size, dest, tag)	Emet size octets de src_addr vers le processus dest avec l'Id tag
RECEIVE	RECEIVE(buff_addr, size, src, tag)	Reçoit un message avec l'Id tag et range size octets à buff_addr
SEND_PROBE (comm asynchrone)	SEND_PROBE(tag, dest)	Teste si le message avec l'Id tag a été expédié au process dest
RECV_PROBE (comm asynchrone)	RECV_PROBE(tag, src)	Teste si le message avec l'Id tag a été reçu par le process src
BARRIER	BARRIER(name, nb)	Synchro globale sur nb process La barrière est infranchissable tant que nb n'est pas atteint
WAIT_for_END	WAIT_for_END (nb)	Attend que nb process terminent

Exemple : l' algorithme de Gauss_Seidel

A[i,j] = 0,2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])



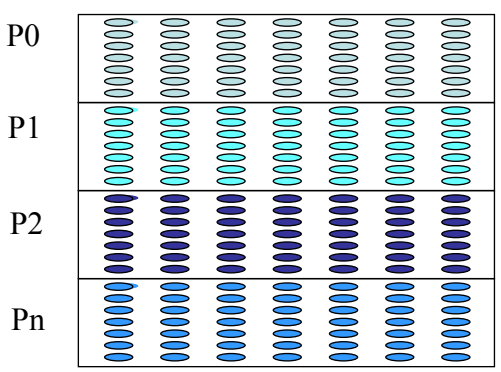
-Résolution d'équations aux dérivées partielles
-Méthode des différences finies
-Grille régulière de (n+2)*(n+2)
-Les points de la grille n*n sont calculés
-Parcours de gauche à droite puis de haut en bas.
-L'ordre du calcul n'a pas d'importance.

Rappel de l'algorithme séquentiel

```
1       int n;                                              /* matrice de n+2 par n+2*/
2       float **A, diff =0;
3       main ()
4       begin
5               read(n) ;
6               A <- malloc (tableau de n+2*n+2 de doubles) ;
7               init(A);
8               solve(A);
9       end main

10      fonction solve (float **A)
11      begin
12      int i,j, done <-0 ;
13      float diff <-0, temp <-0 ;
14      while (!done) do
15              diff <-0 ;
16              for i<-1 to n do
17                      for j<-1 to n do
18                      temp <- A[i,j] ;    /* sauvegarde de l'ancienne valeur */
19                       A[i,j] <-0.2 *(A[i,j] + A[i,j-1] + A[i-1,j]+A[i,j+1] + A[i+1,j] )
20                      diff += abs(A[i,j]-temp);
21                      end for
22              end for
23              if (diff/(n*n )< TOL) done <-1 ;
24      end while
25      end
```


Algorithme parallèle

Soit nprocs le nombre de flots à engendrer : nprocs est un diviseur de la taille de la matrice n.



Chacun des n flots traite un nombre égal de lignes contigües de la grille (64 par exemple). Les flots peuvent être exécutés par n processeurs différents.

Programmation parallèle distribuée

```
1       int pid , n; nprocs                 /* No de process, matrice de n+2 par n+2*/
2       float **myA ;                       /* variables globales partagées */
3       main ()
4       begin
5               read(n) ;
5.1             read(nprocs) ;                      /* nprocs processeurs */
6               CREATE (nprocs-1, Solve)  ;
7               solve();
8               WAIT_FOR_END (nprocs-1) ;   /* synchro globale on attend la fin
                                                tous les fils */
9       end main
```



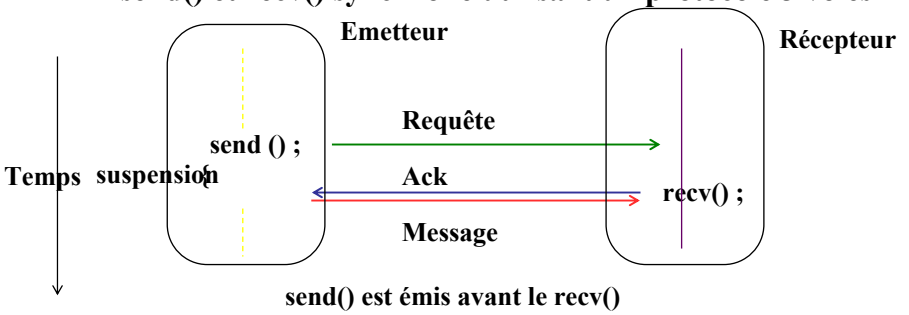
```
fonction solve (float **A)
11      begin
12      int i,j, done <- 0 , pid , n' = n/nprocs;       /* données locales à chaque process de nom  pid */
13      float mydiff, tempdiff, temp <-0, ;
13.1    myA <- malloc ( tableau de flottants de n/nprocs +2 * n+2) ;
13.2    init (myA) ; getpid(pid) ;
14      while (!done) do                    /* chaque process procède au test de fin */
15              mydiff <- 0 ;
15.1            if (pid !=0) then SEND(&myA[1,0], n*sizeof(float),pid-1,ROW) ;
15.2            if (pid != nprocs-1) then SEND(&myA[n',0], n*sizeof(float),pid+1,ROW) ;
15.3            if (pid != 0) then RECEIVE(&myA[0,0], n*sizeof(float),pid-1,ROW) ;
15.4            if (pid != nprocs-1) then RECEIVE(&myA[n'+1,0], n*sizeof(float),pid+1,ROW) ;

16              for i<-1 to n' do
17                      for  j<-1 to n  do
18                      temp <- myA[i,j] ;          /* sauvegarde de l'ancienne valeur */
19                      myA[i,j] <-0.2 *(myA[i,j] + myA[i,j-1] + myA[i-1,j]+myA[i,j+1] + myA[i+1,j] )
20                      mydiff += abs(myA[i,j]-temp);
21                      end for
22              end for
```

communication synchrone = DEADLOCK !

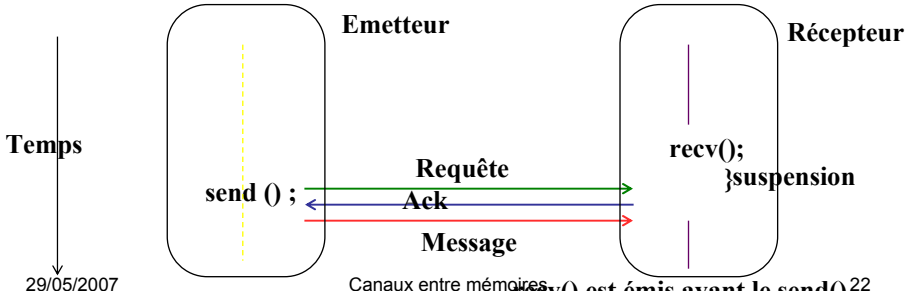
Les lignes des bords des domaines sont copiés dans myA[0,*] et myA[n'+1,*]

```
22.1    if (pid != 0) then SEND (mydiff, sizeof(float), 0,DIFF) ; /* le process 0 gère un global diff */
                           RECEIVE(done, sizeof(int), 0, DONE);
        else
22.2        for i <- 1 to nprocs-1 do
22.3        RECEIVE(tempdiff, sizeof(float), *, DIFF);
22.4        mydiff += tempdiff ;
            end_for
23          if (mydiff/(n*n )< TOL) done <-1 ;
23.1        for i<-1 to nprocs-1 do
23.2        SEND(done,sizeof(int),i,DONE) ;
            end_for
            end_for
24      end_if
25      end while
26      end_fonction
```


send() et recv() synchrone utilisant un protocole 3 voies

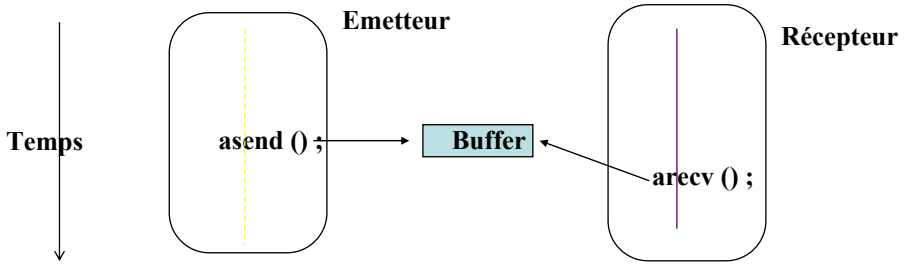


send() est émis avant le recv()



recv() est émis avant le send()

Transfer asynchrone de messages



Résolution du deadlock précédent

```
fonction solve (float **A)
11      begin
12      int i,j, done <- 0 , pid , n' = n/nprocs;        /* données locales à chaque process de nom  pid */
13      float mydiff, tempdiff, temp <-0, ;
13.1    myA <- malloc ( tableau de flottants de n/nprocs +2 * n+2) ;
13.2    init (myA) ; getpid(pid) ;
14      while (!done) do                    /* chaque process procède au test de fin */
15              mydiff <- 0 ;
15.1            if (pid !=0) then ASEND(&myA[1,0], n*sizeof(float),pid-1,ROW) ;
15.2            if (pid != nprocs-1) then ASEND(&myA[n',0], n*sizeof(float),pid+1,ROW) ;
15.3            if (pid != 0) then ARECEIVE(&myA[0,0], n*sizeof(float),pid-1,ROW) ;
15.4            if (pid != nprocs-1) then ARECEIVE(&myA[n'+1,0], n*sizeof(float),pid+1,ROW) ;
15.5            WAIT_FOR_END (nprocs) ;
16              for i<-1 to n' do
17                      for  j<-1 to n  do
18                      temp <- myA[i,j] ;          /* sauvegarde de l'ancienne valeur */
19                      myA[i,j] <-0.2 *(myA[i,j] + myA[i,j-1] + myA[i-1,j]+myA[i,j+1] + myA[i+1,j] )
20                      mydiff += abs(myA[i,j]-temp);
21                      end for
22              end for
```

Schéma de communications collaboratives : Broadcast

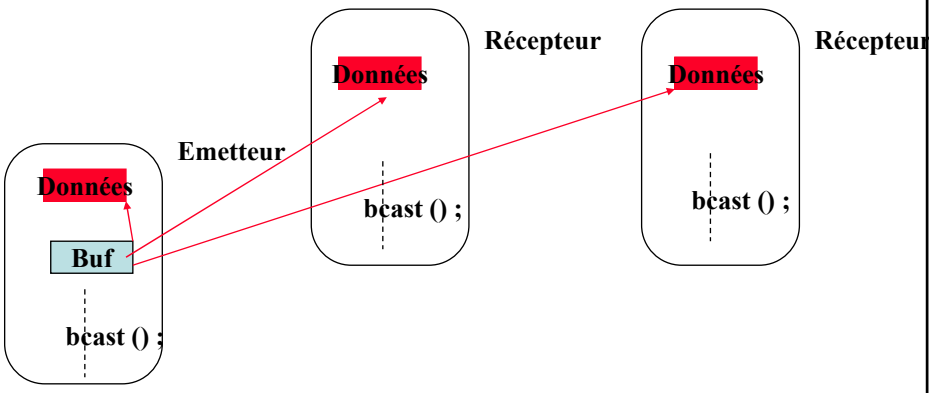


Schéma de communications collaboratives : Scatter

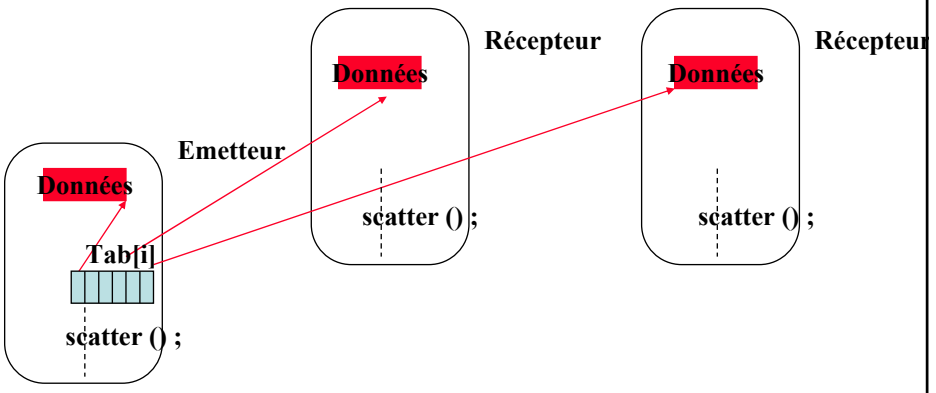
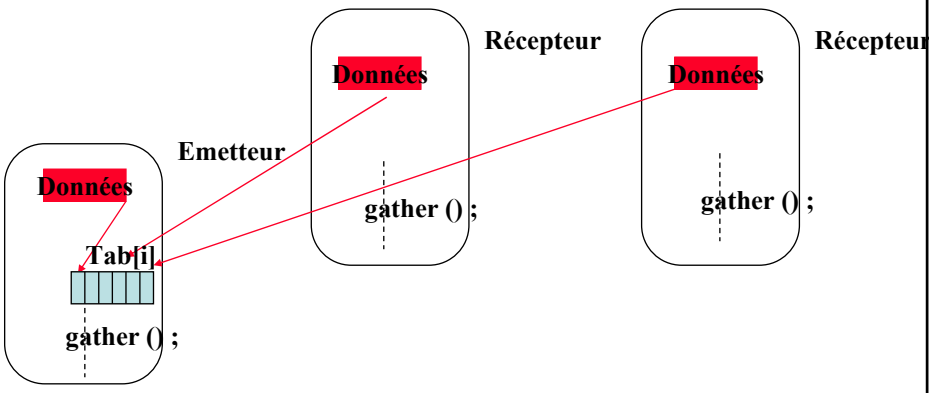


Schéma de communications collaboratives : Gather



Exemple pour la communication des diff locaux et la détermination du done :

```
22.1    REDUCE (0, mydiff,sizeof(float), ADD) ;
22.2    if (pid == 0) then
23              if (mydiff/(n*n )< TOL) done <-1 ;
23.1            end_if
23.2    BROADCAST(0,done,sizeof(int),1,DONE) ;
25      end while
26      end_fonction
```