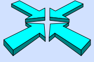
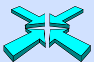
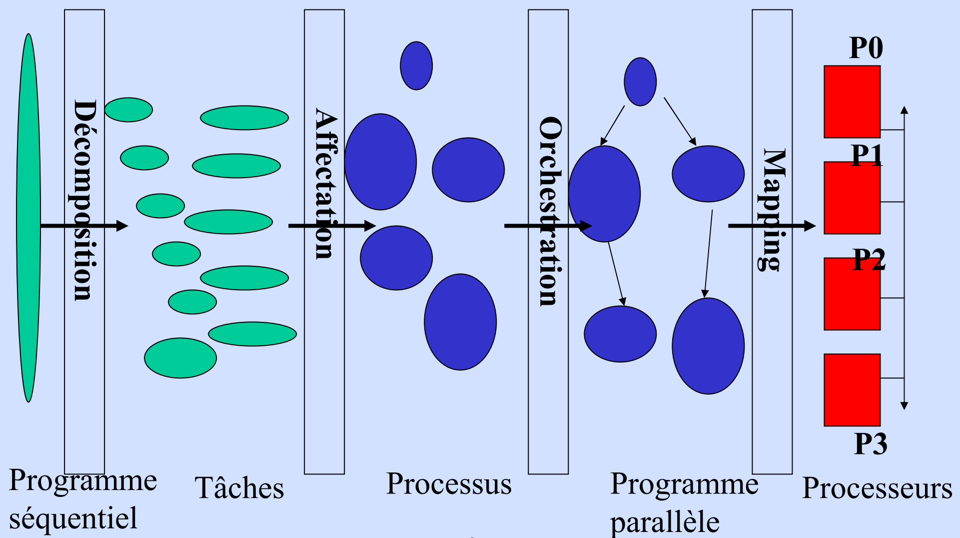
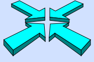


3) Programmation d'applications hautes performances



3.1) Le processus de parallélisation

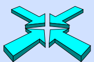




3.2) Objectifs du processus de parallélisation

Etapes	Dépendant de l'archi.	Principaux objectifs
Décomposition	Non	Déterminer suffisamment de concurrence (pas trop).
Affectation	Non	Equilibrer la charge. Réduire le volume de communication.
Orchestration	Oui	Réduire les communications en exploitant la localité des données. Réduire les coûts de communication et synchro. Réduire les opérations de sérialisation lors d'accès à des ressources partagées. Ordonnancer les tâches pour satisfaire les dépendances au plus tôt.
Mapping	Oui	Affecter les processus dépendants sur les mêmes procs

exo

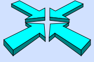


3.3) L'orchestration de processus parallèles

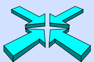
- **Le parallélisme de données (mode data parallèle)**
  Linéarisation des données et du code (pipe-line)

- **La mémoire partagée (Shmem)**
  Un seul espace d'adressage

- **Le mode “passage de message” (PVM, MPI...)**
  Le multi-domaine

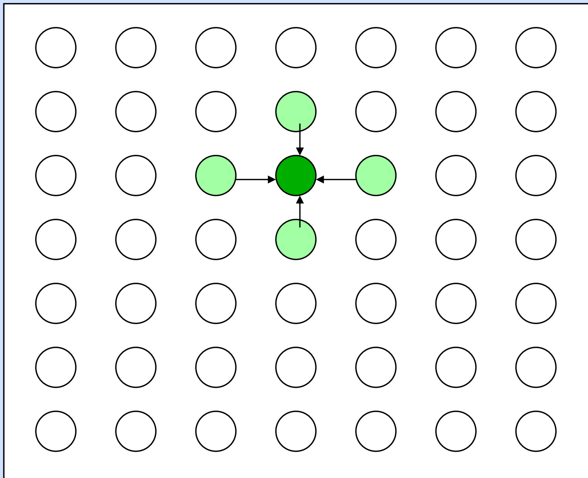


Le processus de parallélisation à travers un exemple

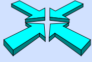


Mise en œuvre : algo de Gauss_Seidel

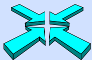
A[i,j] = 0,2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])



-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.

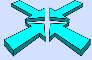


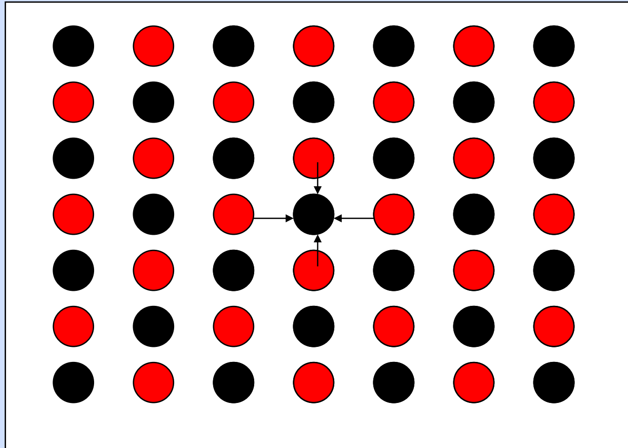
Algorithme séquentiel



```
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
```

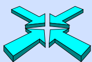
dépendances





Le balayage de la grille est divisé en deux :

Un pour les points rouges et un pour les points noirs

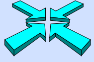


Décomposition

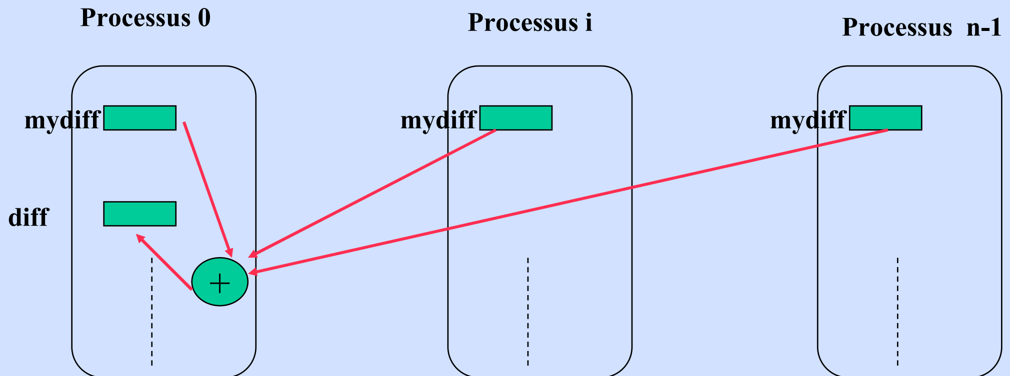
Sol1, parallélisme maximal : la mise à jour d'un point est une tâche.

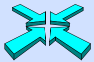
degré de concurrence
$n^2/2 + n^2/2$

```
14      while (!done) do
15              mydiff <-0 ;
16              for_all  i<-1 to n do          /* un nid de boucle parallèle */
17                      for_all  j<-1 +(i+1)mod2 step2 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                      mydiff += abs(A[i,j]-temp);
21                      end for_all
22                      for_all  j<-1 +(i+2)mod2 step2 to n do
23                      temp <- A[i,j] ;    /* sauvegarde de l'ancienne valeur  */
24                       A[i,j] <-0.2 *(A[i,j] + A[i,j-1] + A[i-1,j]+A[i,j+1] + A[i+1,j] )
25                      mydiff += abs(A[i,j]-temp);
226                     end for_all
27              end for_all
27.1            REDUCE (mydiff, diff, ADD) ;
28              if (diff/(n*n ) < TOL) done <-1 ;
29      end while
30      end
```



l'opération reduce

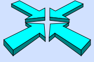




Sol2, décomposition en ligne : le travail pour une ligne complète est une tâche indivisible qui doit être affecté à un processeur.

```
14      while (!done) do
15              mydiff <-0 ;
16              for_all  i<-1 to n do          /* a parallel loop nest*/
17                      for   j<-1 to n do
18                      temp <- A[i,j] ;    /* save of the old value */
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                      mydiff += abs(A[i,j]-temp);
21                      end for
22              end for_all
22.1            REDUCE (mydiff, diff, ADD) ;
23              if (diff/(n*n ) < TOL) done <-1 ;
24      end while
25      end
```

Complexité de chaque tâche ?
Degré de concurrence ?
Nombre de communications par tâche ?



Affectation

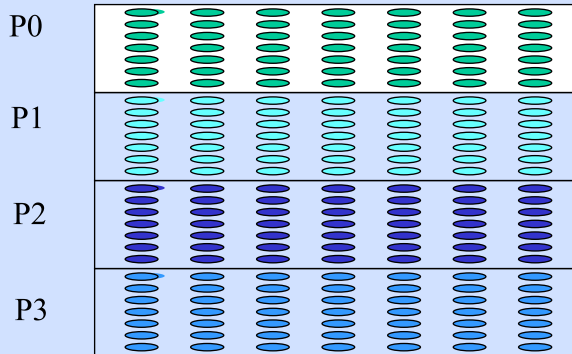
En utilisant une décomposition par lignes, comment affecter explicitement des lignes à des processus ?

P0

P1

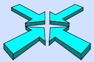
P2

P3



Chacun des 4 processus traite un nombre égal de lignes contigües de la grille.

Le degré de concurrence est p, le nombre de processus.



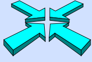
Orchestration pour un modèle data parallèle

G_MALLOC : Global allocation dans le tas

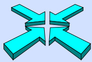
DECOMP : affectation des itérations aux processus et distribution des données

mydiff : instance par processus

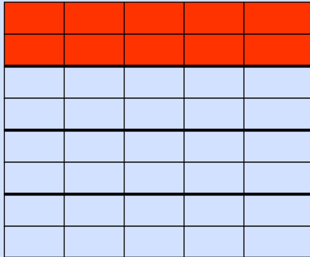
REDUCE : réduction globale sur les données partagées



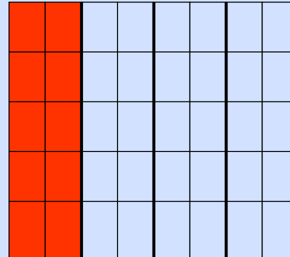
```
1     int n, nprocs;                              /* matrice de n+2 par n+2*/
2     float **A, diff =0;                         /* variables globales */
3     main ()
4     begin
5             read(n) ; read(nprocs) ;
6             A <- G_malloc (tableau de n+2*n+2 de doubles)
7             init(A);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

10    fonction solve (float **A)
11    begin
12    int i,j, done <-0 ;
13    float mydiff <-0, temp <-0 ;
13.1  DECOMP A(BLOCK,*,nprocs)     /* décomposition par lignes sur nprocs  processeurs */
14    while (!done) do
15            mydiff <-0 ;
16            for_all  i<-1 to n do          /* iération sur les points internes */
              *
              *
              *
27            end for_all
27.1          REDUCE (mydiff, diff, ADD) ;
23            if (diff/(n*n ) < TOL) done <-1 ;
24    end while
25    end
```

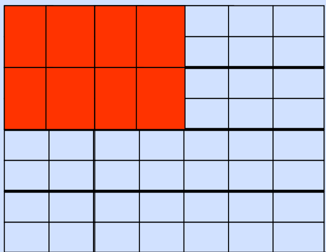
DECOMP A(BLOCK,*,nprocs)



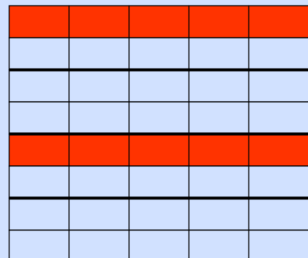
DECOMP A(*,BLOCK,nprocs)

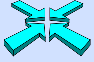


DECOMP A(BLOCK,BLOCK,nprocs)



DECOMP A(CYCLIC,*,nprocs)





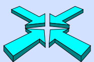
Orchestration pour un modèle à mémoire partagée

CREATE : création de processus

BARRIER : barrière de synchronisation

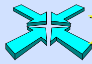
LOCK : pose d'un verrou

UNLOCK : déblocage

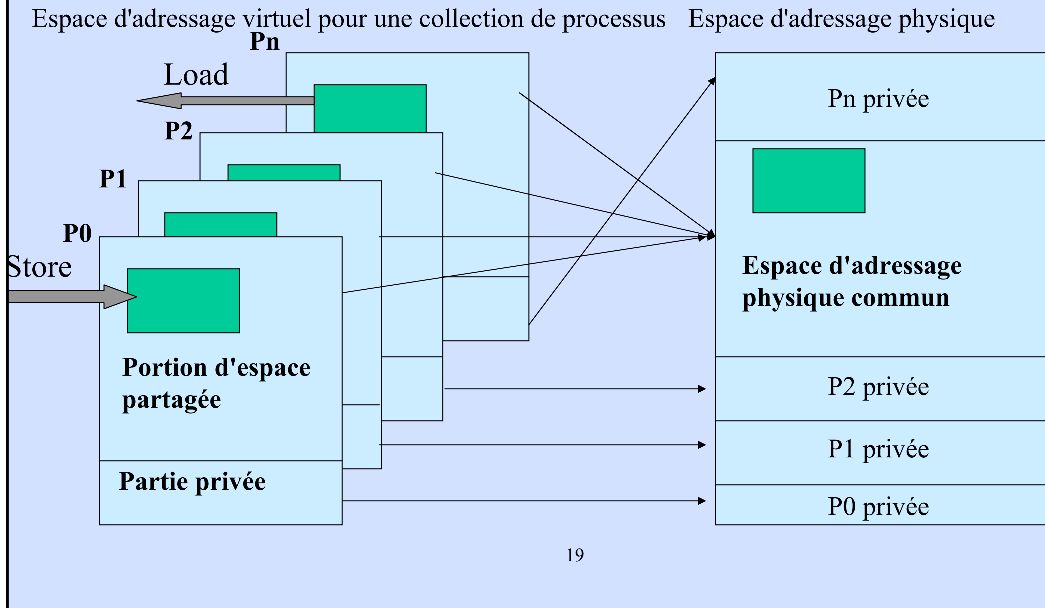


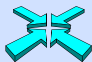
Mémoire partagée Primitives

NOM	Syntaxe	Fonction
CREATE	CREATE (p,proc,args)	Crée p processus dont le main est proc avec args argument
G_MALLOC	G_MALLOC (size)	Alloue une zone mémoire partagée de size octets
LOCK	LOCK (name)	Réalise un accès en exclusion mutuelle sur LOCK
UNLOCK	UNLOCK (name)	Relâche le précédent
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
wait for flag	while (!flag) ; WAIT (flag)	Attend que flag soit positionné
set flag	flag = 1 ; ou SIGNAL (flag)	Réveille des process bloqués sur la lecture du flag.

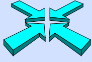


Modèle de programmes parallèles à mémoire partagée

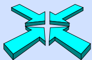




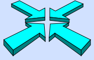
```
1      int n ,  nprocs;          /* No de process, matrice de n+2 par n+2*/
2      float **A, diff ;         /* variables globales partagées : différence dans un balayage
                                    courant*/
2.a    LOCKDEC(diff_lock)        /* declaration d'un verrou d'exclusion mutuelle */
2.b    BARDEC (bar1)             /* declaration d'une barrière pour synchro globale*/

3      main ()
4      begin
5              read(n) ;
5.1            read(nprocs) ;                 /* nprocs processus */
6              A <- G_MALLOC(n*sizeof(float))  ;
7              initialize(A) ;
8              CREATE (nprocs-1, Solve, A) ;
9              solve(A);                      /* le principal est aussi un processus */
10             WAIT_FOR_END (nprocs-1) ;              /* synchro globale on attend la fin
                                                       tous les fils */
11     end main
```



```
fonction solve (float **A)
11      begin
12      int i,j, done <-0 ; pid <-0 ;           /* données locales à chaque process de nom  pid */
13      float mydiff <-0, temp <-0 ;
13.1    int mymin =1 + (pid * n/nprocs) ;
13.2    int mymax = mymin +n/nprocs -1 ; /* n est un multiple de nprocs */
14      while (!done) do                        /* chaque process procède au test de fin */
15              diff = mydiff <-0 ;
15.1            BARRIER(bar1,nprocs) ; /* tous les process sont là avant de modifier diff */
16              for i<-mymin to mymax 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                      mydiff += abs(A[i,j]-temp);
21                      end for
22              end for
22.1            LOCK(diff_lock) ;           /* mise à jour du diff global */
22.2            diff  += mydiff ;           /* accumulation de tous les mydiff locaux */
22.3            UNLOCK(diff_lock) ;
22.4            BARRIER(bar, nprocs) ;  /* tous sont là, avant de tester la fin */
23              if (diff/(n*n )< TOL) done <-1 ; /* tous recoivent la même réponse */
23.1            BARRIER (bar1, nprocs) ;
24      end while
```

Synchronisation dans le modèle à mémoire partagée

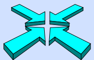


Exclusion mutuelle

Explication : Séquence 22.1 à 22.3
Permet de se prémunir du schéma d'exécution suivant qui conduit à un programme erroné
Soit 1 la valeur calculée de mydiff par chaque processeur ; après l'exécution de 2 tâches diff = 2.

Temps	Processeur 1	Processeur 2
1	→ R1 <- diff	→
2	→	→ R1 <- diff
3	→R1 <- R1 + mydiff /*de P1 */	→ R1 <- R1 + mydiff /* de P2*/
4	→diff <- R1 /* diff = 1 */	→
5		→ diff <- R1 /* diff = 1 */

diff vaut 1 alors que la valeur exacte est 2



Synchronisation globale

Explication de 22.4

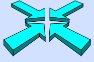
Paramètres : le nom de la barrière et le nombre de processus à synchroniser

Sémantique : attendre que p process soient arrivés à ce point et poursuivre

Implémente une synchronisation all-to-all

Synchonisation point-à-point par drapeaux

Processeur 1	Processeur 2
int flag = 0 ;	→ A =1 ;
	→ b : flag =1
→ a : while (flag is 0) do nothing;	
→ print A ;	



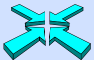
Consistance de la mémoire

- La cohérence ne dit rien sur l'instant à partir duquel l'écriture devient visible pour tous. Soit les 2 processus P1 et P2 (A = flag =0)

P1	**P2**
A = 1 ;	*while (flag == 0) wait ;*
flag = 1;	*print A ;*

La cohérence n'implique pas que la nouvelle valeur de A ne devienne visible pour P2 qu'après la mise à jour de la variable *flag*.

•Pour établir un ordre sur les accès à la même variable par plusieurs processus, on attend d'un système mémoire qu'il respecte l'ordre des lectures/écritures vers différentes variables (A et flag) réalisées par le même processus.

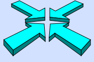


Orchestration pour un modèle à mémoire distribuée

CREATE : création de processus

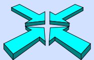
SEND : envoi de données

RECEIVE : réception de données

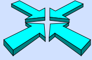


Mémoire distribuée Primitives

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



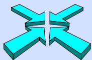
```
1    int pid , n; nprocs                    /* No de process, matrice de n+2 par n+2*/
2    float **myA ;                          /* variables globales partagées */
3    main ()
4    begin
5         read(n) ;
5.1       read(nprocs) ;                    /* nprocs processeurs */
6         CREATE (nprocs-1,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



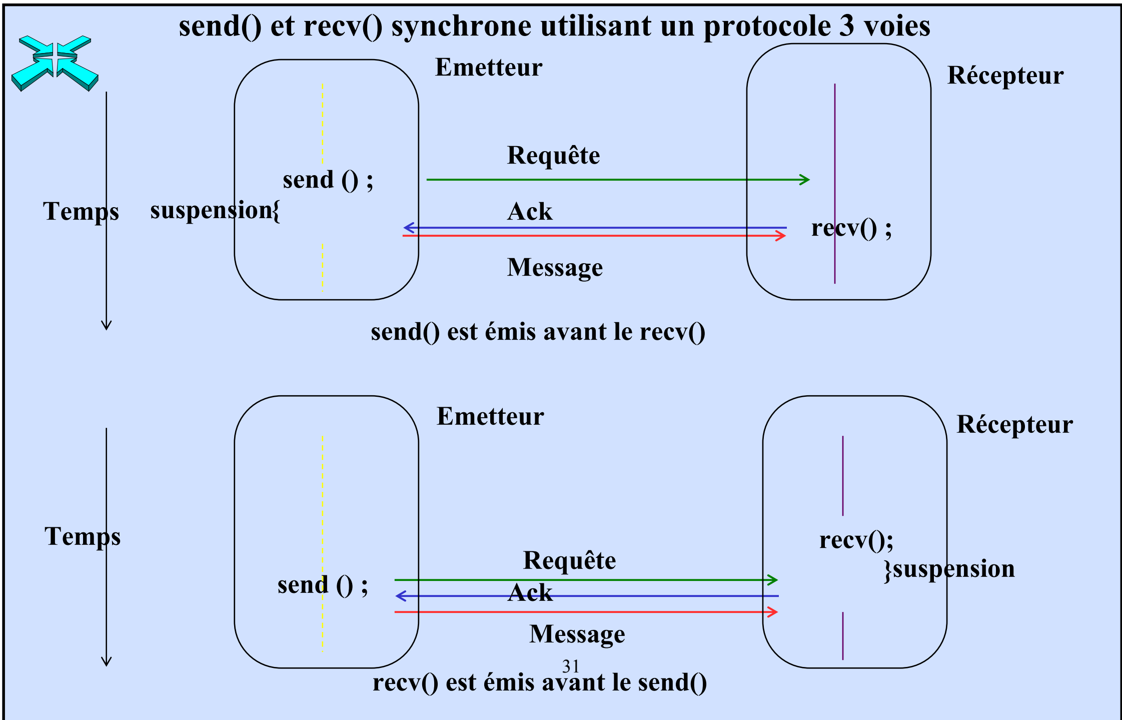
```
fonction solve (float **A)
11      begin
12      int i,j, done <- 0 , pid , n' = n/nprocs;      /* données locales à chaque process de nom  pid */
13      float mydiff, tempdiff, temp <-0, ;
13.1    myA <- malloc ( tableau de flottants de n/nprocs +2 * n+2) ;
13.2    init (myA) ;
14      while (!done) 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
```

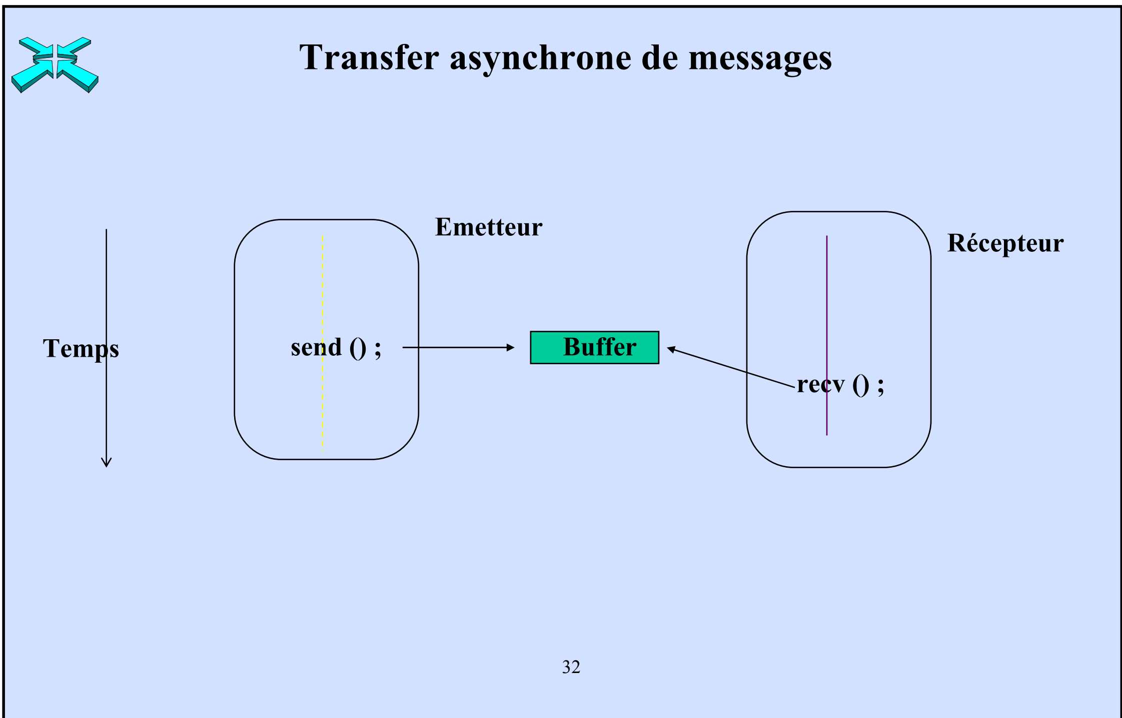
Les lignes des bords des domaines sont copiés dans myA[0,] et myA[n'+1,*]*

communication synchrone => DEADLOCK !



```
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
Emetteur
Récepteur
Temps
send () ;
suspension{
Requête
Ack
Message
recv() ;
send() est émis avant le recv()
Emetteur
Récepteur
Temps
recv();
}suspension
send () ;
Requête
Ack
Message
recv() est émis avant le send()
31


Transfer asynchrone de messages
Emetteur
Récepteur
Temps
send () ;
Buffer
recv () ;
32

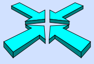
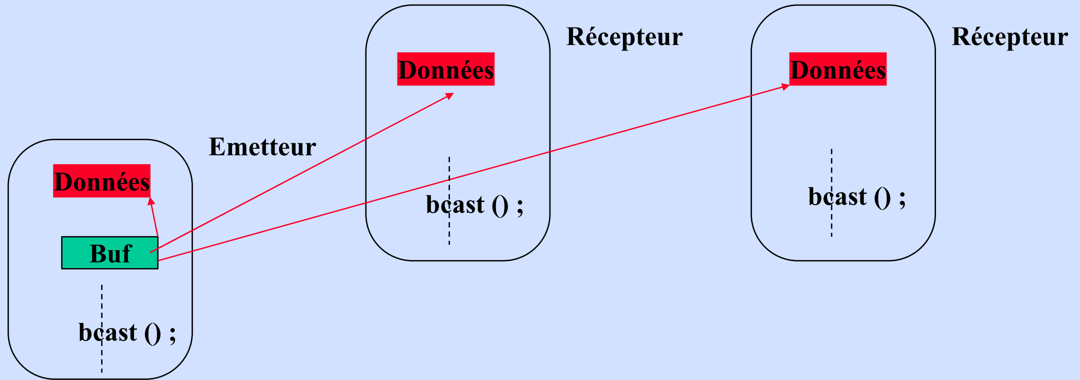


Schéma de communications : Broadcast



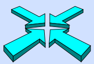
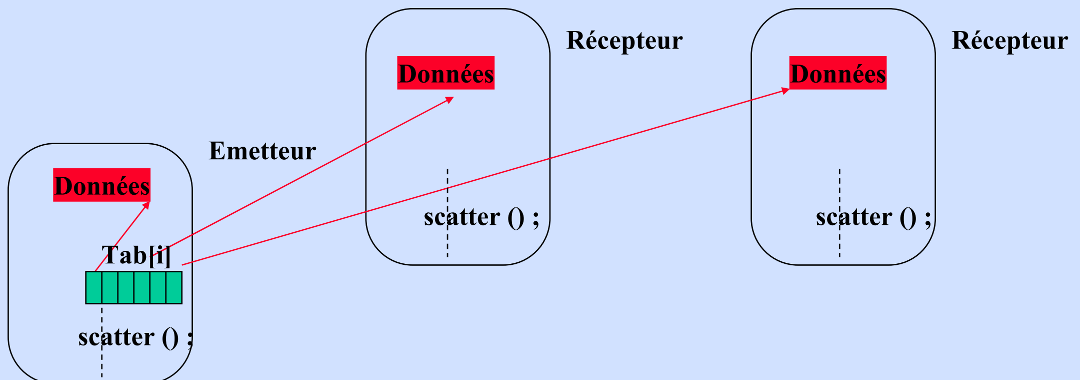


Schéma de communications : Scatter



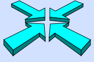
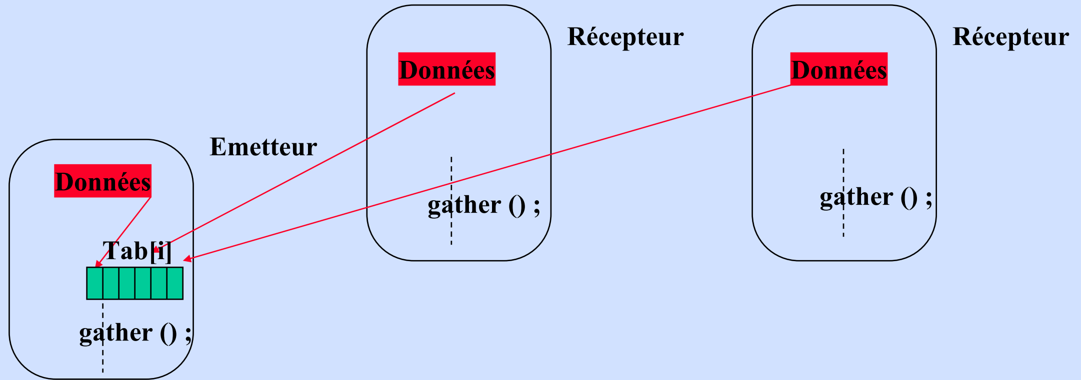
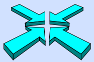


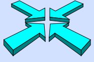
Schéma de communications : Gather





Exemple : communication des diff locaux et détermination du done :

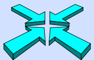
```
22.1    REDUCE (0, mydiff,sizeof(float), ADD) ;
22.2    if (pid == 0) then
23              if (mydiff/(n*n )< TOL) done <-1 ;
23.1            end_if
23.2    BROADCAST(0,done,sizeof(int),1,DONE) ;
25      end while
26      end_fonction
```



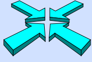
Un exemple de bibliothèque de passage de messages : MPI

- MPI_Init (&argc, &argv)
- MPI_Comm_size(MPI_COMM_WORLD, &p) ; /* nombre de processus */
- MPI_Send(message, liste-param..., &status)
- MPI_Recv(message, liste-param..., &status)
- MPI_Finalize()

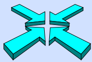
Modes de communication standard, synchrone, ready et bufferisé



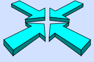
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[ ]) {
        int myrank, num_procs, err ;
err = MPI_Init (&argc, &argv) ;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD,&num_procs) ;
if (myrank == 0 {
        printf ("  proc %d de %d : Hello World du maitre \n" , myrank, num_procs) ;
        } else   {
         printf (" proc %d de %d : Hello World de l ’esclave \n ", myrank, num_procs) ;
                }
```



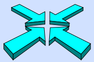
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mat.h"
#include <mpi.h>
#include "chrono.h"

#define MAITRE if (myid==0)
#define ESCLAVES if (myid!=0)

void initialisation(double *T,int n)
{
  double nn = n*n;
  int i;
  for(i=0 ; i<nn ; i++)
    T[i] = (double) rand() / (double) RAND_MAX;
}

double trace(double* T,int n)
{
  int i;
  double sommeTermesDiagonaux = 0;
  for(i=0 ; i<n ; i++)
    sommeTermesDiagonaux += T[i*(n+1)];
  return sommeTermesDiagonaux;
}

int main(int argc,char *argv[])
{
  clock_t dateDebut,dateFin;
  int n,i;
  double *A;
  double *B;
  double *C;
  double *DEC;
  int namelen;
  int nb_procs,myid;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Status *stat;


  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Get_processor_name(processor_name, &namelen);

  n = atoi(argv[1]);
```

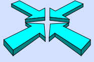
```c
 MAITRE
     {
       A = (double*) calloc(n*n,sizeof(double));
       B = (double*) calloc(n*n,sizeof(double));
       C = (double*) calloc(n*n,sizeof(double));

       printf("\n\n\t-------------- Multiplication de matrice %dx%d sur %d
 noeuds --------------\n",
            n,
            n,
            nb_procs);

       initialisation(A,n);
       initialisation(B,n);

       dateDebut = clock();
       chrono("Global");
       chrono("Distribution B");
       MPI_Bcast(B,n*n,MPI_DOUBLE,0,MPI_COMM_WORLD);
       chrono("Distribution B");

       chrono("Envois des parties de A");
       for (i=1; i<nb_procs; i++)
       {
         DEC=A+i*(n*n/nb_procs);
         MPI_Send(DEC,
                n*n/nb_procs,
                MPI_DOUBLE
                ,i
                ,0
                ,MPI_COMM_WORLD);
       }
    chrono("Envois des parties de A");

    chrono("Calcul d'une partie");
    produitMatrice(A,B,C,n/nb_procs,n);
    chrono("Calcul d'une partie");

    chrono("Reception des resultats");
       for (i=1; i<nb_procs; i++)
       MPI_Recv(C+i*(n*n/nb_procs),
              n*n/nb_procs,
              MPI_DOUBLE,
              i,
              0,
              MPI_COMM_WORLD
              ,stat);
    chrono("Reception des resultats");

       chrono("Global");
       dateFin = clock();

       /*    printf("Execution en %5.2f secondes - Trace(C) = %.4f\n",
            (double) (dateFin-dateDebut)/(double) CLOCKS_PER_SEC,trace(C,n));
       */
       chrono("BILAN");
     }
```



```
  ESCLAVES
    {
      A = (double*) calloc(n*n/nb_procs,sizeof(double));
      B = (double*) calloc(n*n,sizeof(double));
      C = (double*) calloc(n*n/nb_procs,sizeof(double));

      MPI_Bcast(B,n*n,MPI_DOUBLE,0,MPI_COMM_WORLD);

      MPI_Recv(A,n*n/nb_procs,MPI_DOUBLE,0,0,MPI_COMM_WORLD,stat);

      produitMatrice(A,B,C,n/nb_procs,n);

      MPI_Send(C,n*n/nb_procs,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    }

  free(A);
  free(B);
  free(C);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

4) Etude de performances



Améliorer l'efficacité du calculateur parallèle

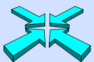
Limites physiques : Expédier 1 bit à 100 mètres prend un temps théorique équivalent à l'exécution locale de 1.000 instructions.

demain

10.000 voire 100.000 instructions

Solutions:

- Réduire les latences en améliorant les localités : nouvelles politiques de gestion mémoire
- Tolérer les latences restantes en pipelinant communication et calcul



4.1) Anatomie du modèle passage de messages

Standard de programmation : MPI

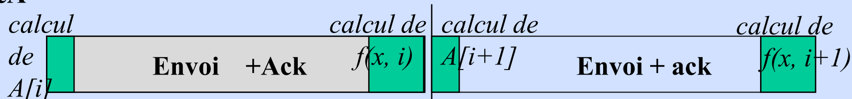
ProcA

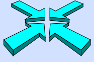
```
For i<-0 to n-1 do
   calcul de A[i] ;
   write  A[i] ;
   send (A[i] to ProcB) ;
   calcul de f(x, i) ;
  suite ...
End for
```

ProcB

```
For i<-0 to n-1 do
        receive (myA[i] from ProcA) ;
         use myA[i] ;
         calcul de g(y, i) ;
         suite ...
End for
```

ProcA





4.2) Anatomie du modèle mémoire partagée

Standard de programmation : OpenMP

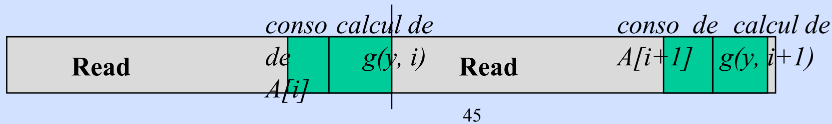
ProcA

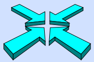
For i<-0 to n-1 do
calcul de A[i] ;
write A[i] ;
calcul de f(x, i) ;
suite ...
End for
Flag <- 1 ;

ProcB

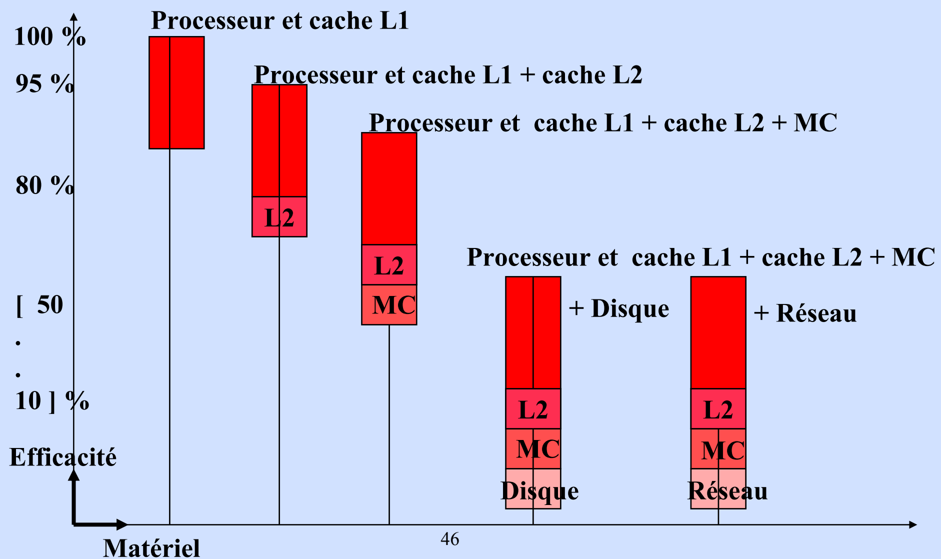
While flag = 0 itère;
For i<-0 to n-1 do
read A[i];
use A[i] ;
calcul de g(y, i) ;
suite ...
End for

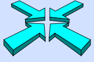
ProcB





4.1) Réduire et tolérer les latences



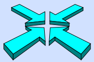


4.1.1) Les paramètres de l'efficacité

$$T_p = t_0 + t_{n+1} + \max t_i$$

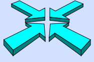
$$T_{min} = (t_0 + t_{n+1}) + \frac{T_{par}}{n}$$

Equilibre de charge $= \dfrac{T_p - T_{min}}{T_{min}}$



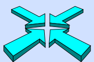
4.1.2) Réduire les latences

- Réduire les latences
  - Réduire les temps d 'accès à chaque niveau de la hiérarchie mémoire - tendre vers les coûts matériels
  - Re-structurer l'organisation système pour réduire la fréquence des accès à latence importante
    - Duplication des données - cache
    - Amélioration de la localité des données
  - Re-structurer l'application
    - Décomposer et allouer localement le calcul au processeur
    - Re-structurer les données pour améliorer les localités spatiales et temporelles

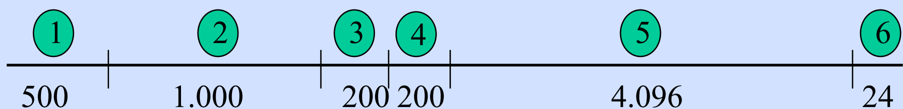


4.1.3) Tolérer les latences

- Tolérer les latences résiduelles : recouvrir calcul et communication
  - Réduire les temps d'exécution à l'intérieur du même processus (concerne la mémoire et le réseau, pas le disque)
  - Latences relativement faibles et donc difficiles à recouvrir : le temps de commutation de contexte de processus classique n'est pas compatible avec ces temps de latences



4.1.4) Identification des latences



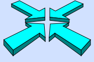
- Accès mémoire ou communication
  - temps processeur, temps interface, délai de transit, l'occupation de la bande passante, la contention
- Latence de synchronisation

Temps séparant l'exécution d 'une opération de synchronisation de la poursuite du calcul (opération lock, barrier)

While (!done) do
mydiff = diff = 0 ;
Barrier(bar1, nprocs) ;
For i <- mymin to mymax do

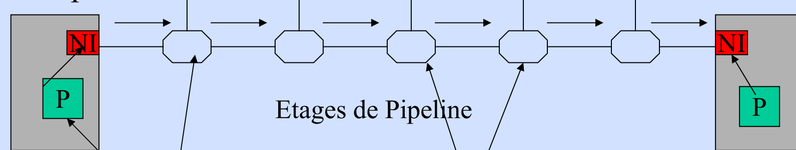
- Latence de l'instruction

Temps de traversée du pipe-line



4.2) Solutions

• Latences des accès mémoires ou des communications : Pipe-line sur les accès distants

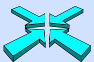


Recouvrement communication avec communication.
Recouvrement calcul par communication.

- 1) Transfert par bloc
  - Regroupement de messages pour réaliser des messages plus longs

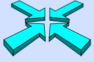
(limité par la bande passante du réseau) pour amortir l'overhead de constitution des messages. Ne nécessite qu'un seul ack du receveur.

La latence du processeur ou du NIC n'est pas impactée.



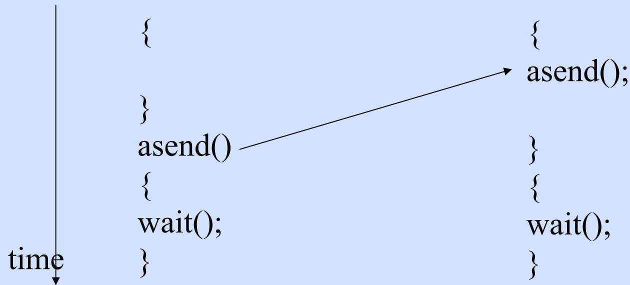
- 2) Prefetching
  - Anticiper la communication en la préparant au plus tôt :

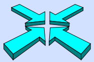
  - à l'initiative de l'émetteur qui l'expédie au récepteur avant que celui-ci ne la demande
  - à l'initiative du receveur : communication établie à la demande
- 3) Asynchronisme
  - Le calcul se poursuit par des opérations indépendantes sans attendre le ack du recepteur : - à l 'initiative de l'émetteur
- 4) Mutithreading
  - communication asynchrone mais commutation vers un thread d 'un autre contexte
  - Un programme multithreadé est un programme parallèle décomposé en P processus, P > nbprocs et P/nbprocs threads sont mappés sur chaque processeur
  - Nécessite un parallélisme additionnel pour tolérer les latences



4.2.1) Exemple de prefetching

La precommunication génère une communication avant le point ou l'opération apparaît dans le programme.



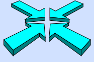


ProcA

For i<-0 to n-1 do
  calcul de A[i] ;
  write A[i] ;
  a-send (A[i] to **ProcB) ; /* envois asynch*/**
End for
/ le calcul est transféré ds une autre boucle */*
For i<-0 to n-1 do
calcul de f(x, i) ;
suite ...

ProcB

a-receive (myA[0] from **ProcA) ;**
/* pipe-line comm-calcul*/
    For i<-0 to n-2 do
    a-receive (myA[i+1] from **ProcA) ;**
    while (!recv-probe(*myA[i]) itère ;*
    use myA[i] ;
    calcul de g(y, i) ;
    suite ...
    End for
while (!recv-probe**(***myA[n-1]) itère ;*
use myA[n-1] ;
calcul de g(y, n-1) ;



Sur le processeur A tous les sends sont émis avant le calcul de f.
Les messages sont envoyés au receveur aussitôt que possible =>

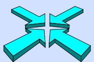
- **pression sur les buffers d'entrée du receveur**

Si l'overhead sur le NIC est important il est avantageux de maintenir un flot de calcul sur le processeur =>

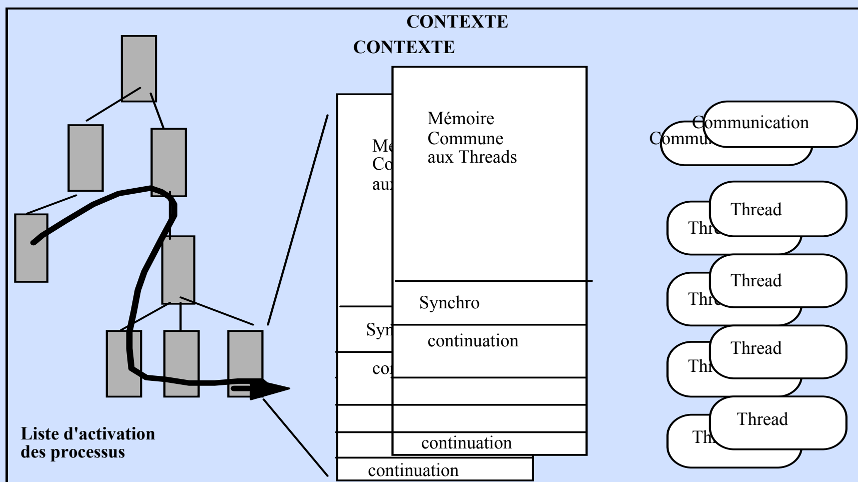
- **construction d 'un pipe-line logiciel**

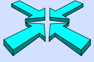
Le *a-receive* envoie la spec du receive au gestionnaire de message et autorise le processeur à continuer. Quand la donnée arrive le NIC est averti et il transfère la donnée ds l'application qui l'a demandée. L'application teste que la donnée est arrivée *recv-probe* avant son utilisation. Si elle est arrivée alors un nouveau *a-receive* est émis pour préparer le suivant =>

- **recouvrement du coût du NIC et éventuellement le coût du transit et du receive.**



Le multithreading





Modèle de programmation

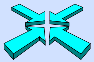
```
DO I = 1, N
        S = 0.0
        DO J = 1,N
                S = S + A(I,J)
        END DO
        B(I)=S
END DO
```

Chaque itération I devient un thread.
La boucle interne fait le “grain”

```
DO I=1,N
      X(I) = A*X(I) + Y(I)
END DO
```

Devient : N-1 threads //

```
         Fork N-1 alpha
         I=N
alpha : X(I) = A*X(I) + Y(I)
        Join N
```

Modèle de threads

Thread1
Thread2

Context Switch

UF1
UF2
UF3
UF4

P1

Superscalaire multithreadé

UF1
UF2

P1

UF1
UF2

P2

SMT : on chip biprocesseur (MAJC de Sun, PowerIV IBM)

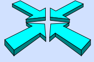
thread1
thread2
thread3

UF1
UF2
UF3
UF4

Switch à chaque cycle (autorise des latences importantes) (TERA, CRAY MTA))

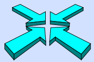
UF1
UF2
UF3
UF4

Simultaneous multithreading (SMT) (PentiumIV, Alpha24464)



CONCLUSIONS

- **Les architectures des systèmes hautes performances sont désormais construites à base d'architectures parallèles composés d'éléments standard :**
  - **à mémoire partagée**
  - **à mémoires distribuées**
- **Les modèles de programmation se standardisent :**
  - **OpenMP**
  - **MPI**
- **Les latences sont nombreuses et pénalisent la rentabilité du système**
  - **de la hiérarchie mémoire (cohérence et consistance)**
  - **des réseaux de communication**
- **Re structurer les algorithmes systèmes de gestion de la machine et les algorithmes des applications en fonction de ces contraintes.**



applications militaires
prévisions météo...

jusqu'à 10,000 procs.

Dizaine de calculateurs :applications scientifiques

BD, exploration pétrolière, méca des structures, aérodynamique..

plus de 64 procs.

Milliers de calculateurs ; calculateurs spécialisés

Marché de l'Entreprise

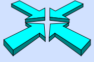
quelques dizaines de procs 10-48

Centaines de milliers à plusieurs millions de serveurs

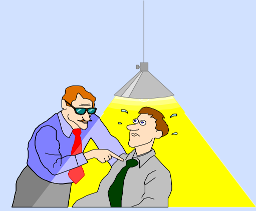
Conventionnelles stations de travail

2-4 proc sur la même carte

millions de petits multiprocesseurs (SMP)



L’AVENIR ?



- **Généralisation de la technique “multithread”**
  * Architecture des processeurs
  * Système d’exploitation
- **Parallélisation automatique de programmes**
  * Pre-processeur, instructions ou directives
- **Calculateurs parallèles =**
  * Réseau de clusters de processeurs (metacomputing)
  * Cluster de processeurs sur une seule mémoire virtuelle