Introduction aux architectures et programmes parallèles

Daniel Etiemble
de@lri.fr

Accélération

- Accélération (p processeurs) = $\frac{\textit{Performance (p processeurs)}}{\textit{Performance (1 processeur)}}$
- Pour un problème de taille fixe (entrées fixées), performance = 1/temps
- Accélération à problème fixé
  - (p processeurs) = $\frac{\textit{Temps (1 processeur)}}{\textit{Temps (p processeurs)}}$

Amdahl et Gustafson

S: Temps séquentiel
P: Temps parallélisable

Taille fixe

Amdahl

$$Acc(n) = \frac{S+P}{S+P/n}$$

Ts = S+P = fs + (1-fs)
Tp = S+P/n = fs +(1-fs)/n

$$Acc(n)_{n\to\infty} \to 1/f_s$$

Application extensible

Gustafson

P = a*n

$$Acc(n) = \frac{S+a.P}{S+P}$$

Ts = S+P = fs + n (1-fs)
Tp = S+P/n = fs +(1-fs)

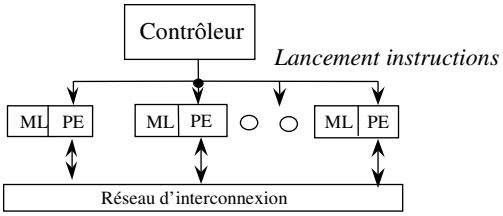
$$E(n) = f_s/n + 1 - f_s$$

Classification de Flynn

- Flot d'instructions x Flot de données
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Single Data
  - Multiple Instruction Multiple Data (MIMD)
- "Tout est MIMD" !

SIMD

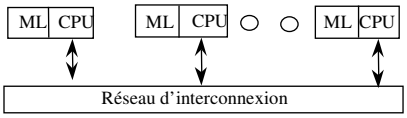
- Modèle d'exécution
  - CM2
  - Maspar



- Modèle obsolète pour machines parallèles
  - Processeurs spécifiques différents des CPU standards
  - Lancement des instructions incompatible avec les fréquences d'horloge des microprocesseurs modernes
  - Synchronisation après chaque instruction : les instructions conditionnelles sont synchronisées
- SPMD
- Extensions SIMD dans les microprocesseurs

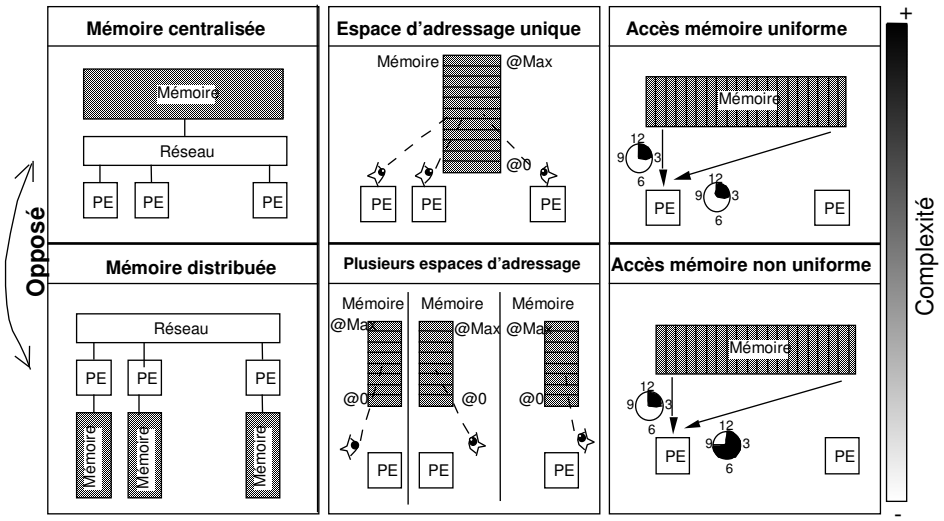
Version SPMD du SIMD

- Modèle d'exécution

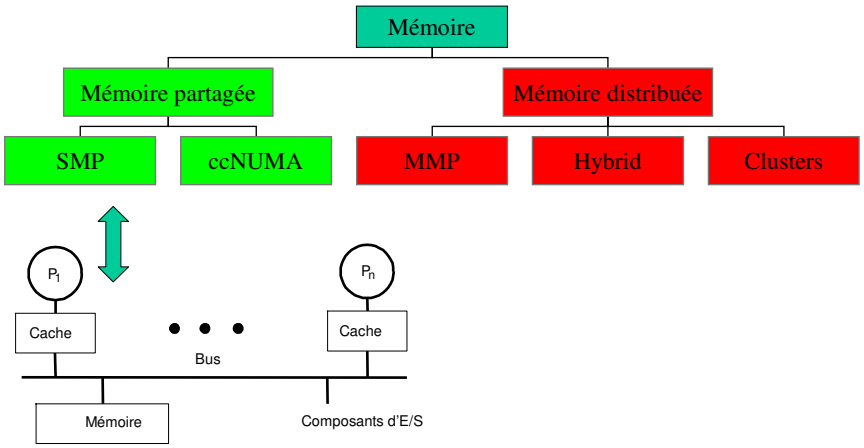


  - Chaque CPU exécute le même code.
  - Les CPU sont synchronisés par des barrières avant les transferts de données.

Organisation mémoire

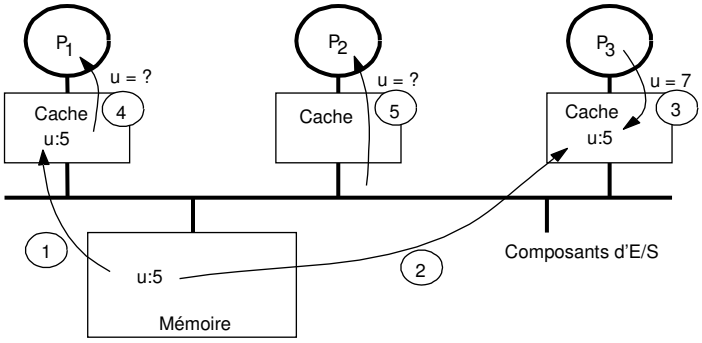


Classification mémoire



Modèle mémoire SMP

Le problème de cohérence des caches



1 P1 lit u
2 P3 lit u
3 P3 écrit dans u
4 P1 lit u
5 P2 lit u

Résultat des lectures 4 et 5
avec écriture simultanée ?
avec la réécriture ?

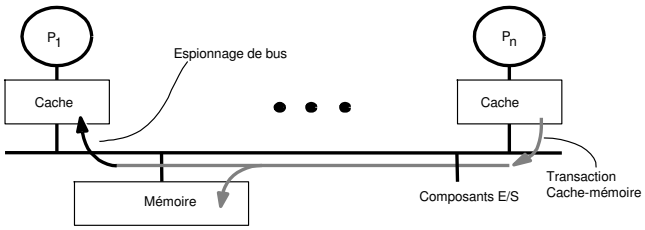
Cohérence des caches avec un bus

- Construite au dessus de deux fondements des systèmes monoprocesseurs
  - Les transactions de bus
  - Le diagramme de transitions d'états des caches
- La transaction de bus monoprocesseur
  - Trois phases : arbitrage, commande/adresse, transfert de données
  - Tous les composants observent les adresses ; un seul maître du bus
- Les états du cache monoprocesseur
  - Ecriture simultanée sans allocation d'écriture
    - Deux états : valide et invalide
  - Réécriture
    - Trois états : valide, invalide, modifié
- Extension aux multiprocesseurs pour implémenter la cohérence

La cohérence par espionnage de bus

- *Idée de base*
  - Les transactions bus sont visibles par tous les processeurs.
  - Les processeurs peuvent observer le bus et effectuer les actions nécessaires sur les évènements importants (changer l'état)
- *Implémenter un protocole*
  - Le contrôleur de cache reçoit des entrées de deux côtés :
    - Requêtes venant du processeur, requêtes/réponses bus depuis l'espion
  - Dans chaque cas, effectue les actions nécessaires
    - Mise à jour des états, fourniture des données, génération de nouvelles transactions bus
  - Le protocole est un algorithme distribué : machines d'états coopérantes.
    - Ensemble d'états, diagramme de transitions, actions
  - La granularité de la cohérence est typiquement le bloc de cache

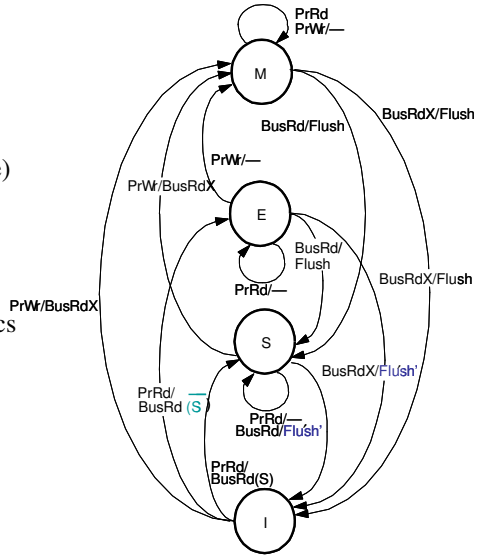
Cohérence avec cache à écriture simultanée



- Extension simple du monoprocesseur : les caches à espionnage avec propagation d'écriture
  - Pas de nouveaux états ou de transactions bus
  - Protocoles à diffusion d'écriture
- Propagation des écritures
  - la mémoire est à jour (écriture simultanée)
- Bande passante élevée nécessaire

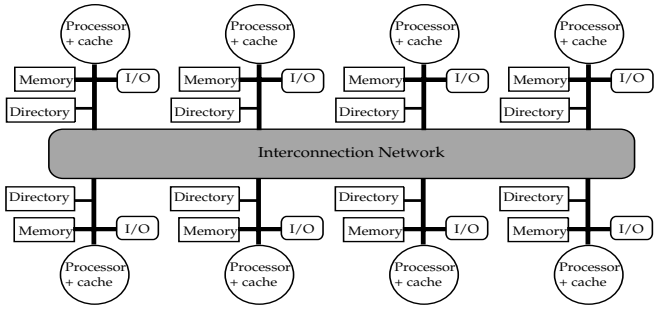
Cohérence de cache avec écriture différée : MESI

- 4 états
  - Exclusif
    - Seule copie du bloc
    - Non modifié (= Mémoire)
  - Modifié
    - Modifié (!= Mémoire)
  - Partagé
    - Copie dans plusieurs blocs
    - Identique en mémoire
  - Invalide
- Actions
  - Processeurs
  - Bus



Cohérence des caches

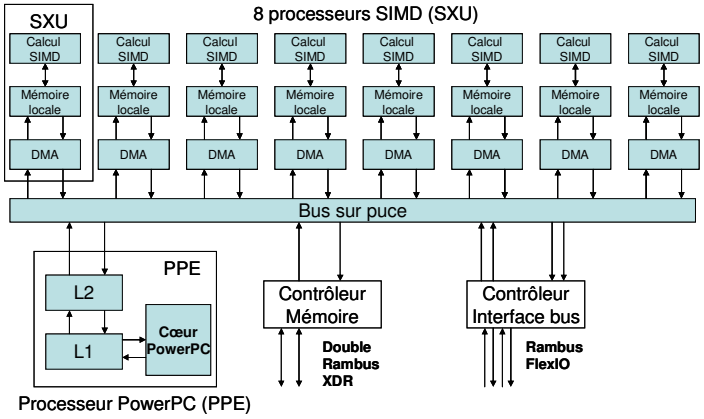
- Espionnage de bus
  - Multiprocesseurs
- Répertoires
  - Lorsque le bus est remplacé par un réseau d'interconnexion ou réseau sur puce (NoC)
  - cc-NUMA : cohérence de caches avec accès non uniforme.



"Caches logiciels"

- Utilisation de mémoires locales
- Transferts gérés par logiciel (avec DMA)

Processeur Cell



Modèles de programmation

	PARALLÉLISME DE DONNÉES	TÂCHES CONCURRENTES
PRINCIPLE	*Opérations successives sur des données multidimensionnelles*	*Le programme est décomposé en tâches concurrentes indépendantes*
DONNEES	Données distribuées dans les mémoires locales (distribution, alignement, partage) **Plusieurs espaces d'adressage**	Données dans une mémoire logiquement partagée **Espace d'adressage unique**
CONTRÔLE	Un seul programme de type séquentiel Contrôle centralisé ou distribué - instructions (SIMD) - code (SPMD)	Distribué (tâches concurrentes) - Accès aux variables partagées - coopération entre processus

Modèles de programmation (2)

Parallélisme de données (HPF)	SPSA : Un seul programme Un seul espace d'adressage	→	**AUTOMATIQUE** Ordonnancement du calcul Accès lointain Synchronisation
Mémoire partagée	MISA : Plusieurs flots d'instructions Un seul espace d'adressage	→	**MANUEL** Ordonnancement du calcul Synchronisation
Passage de messages (MPI)	MIMA : Plusieurs flots d'instructions Plusieurs espaces d'adressage	→	**MANUEL** Ordonnancement du calcul Accès lointain Synchronisation

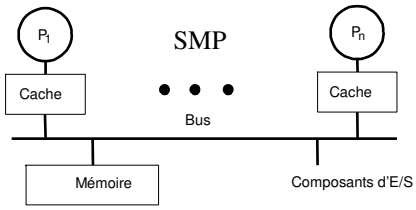
+ -

Complexité pour le compilateur

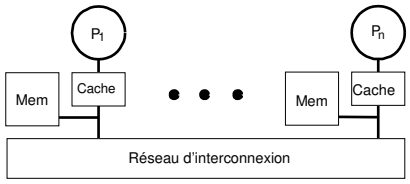
Complexité pour le programmeur

- +

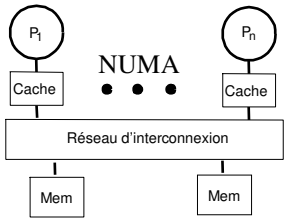
MIMD : extensions naturelles du système mémoire



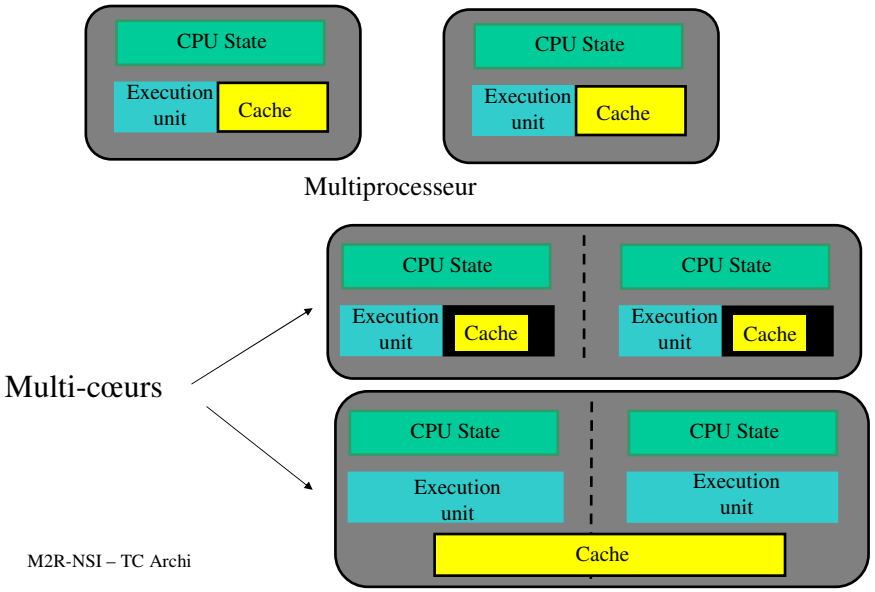
Multiprocesseurs



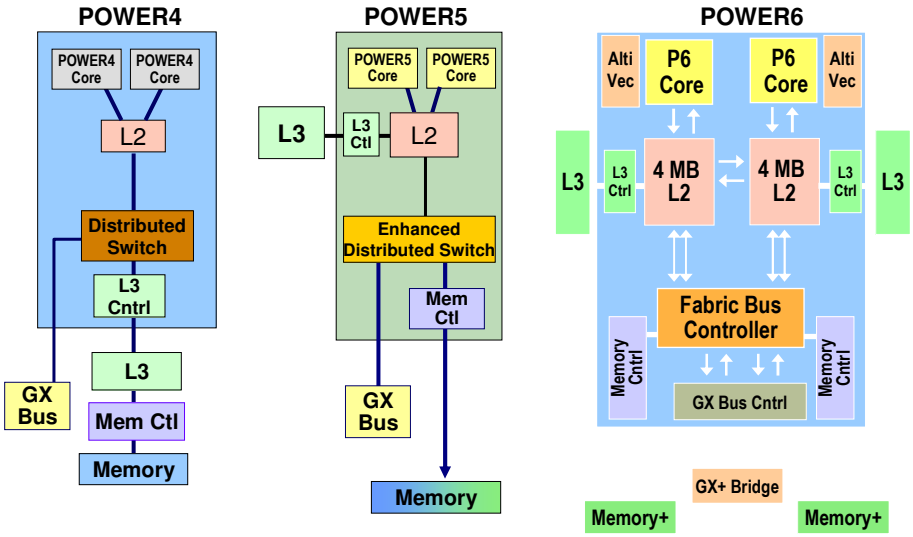
Multi-ordinateurs



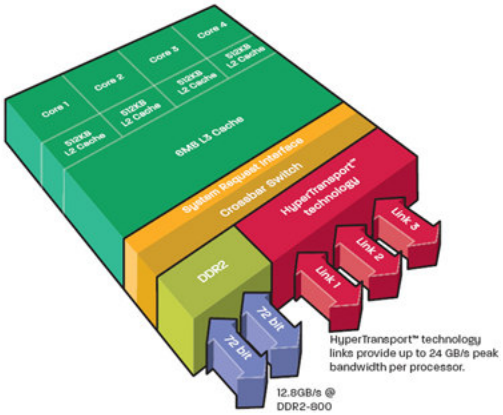
Multiprocesseurs et multi-cœurs



Multi-coeurs : POWER4 / POWER5 / POWER6

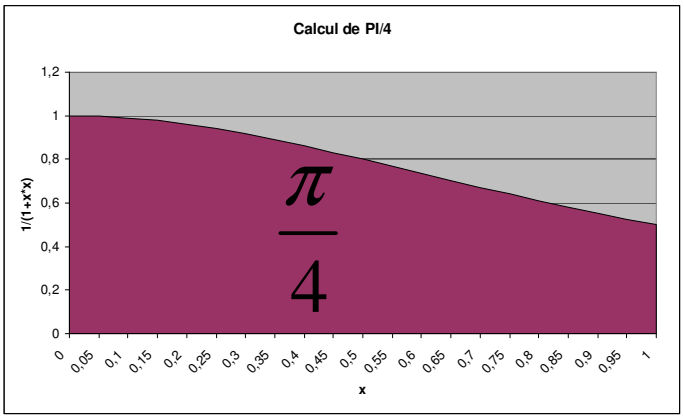


AMD Opteron



Un exemple: le calcul de π

$$\int_0^1 \frac{1}{1+x^2}dx = \arctan(1) - \arctan(0) = \frac{\pi}{4}$$

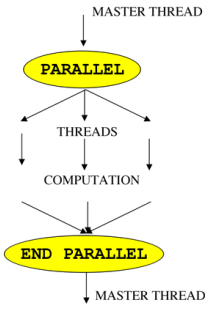


Programme PI : version séquentielle

```
static long etapes = 100000;
double pas;
void main ()
{       int i;   double x, pi, sum = 0.0;

        pas = 1.0/(double) etapes;

        for (i=1;i<= etapes; i++){
                x = ((double)i-0.5)*pas;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = pas * sum;
}
```


Primitives de base en « mémoire partagée »

- Création et destructions de threads
  - CREATE (p, proc, arg) : création de p threads qui commence à exécuter la procédure proc avec les arguments arg
  - WAIT_for_end (p) : fin d'exécution de p processus
- Variables privées et partagées
- Accès exclusif à des variables partagées
  - LOCK(var) et UNLOCK(var) : accès exclusif à une région critique de code
- Barrières
  - Barrière (nom, p) : synchronisation de p threads




```
for (i=1;i<= etapes; i++){
        x = ((double)i-0.5)*pas;
        sum = sum + 4.0/(1.0+x*x)
```

-Répartir les itérations sur différents threads
-Réduction des sommes partielles dans une variable globale

PI open MP : approche SPMD

```
#include <omp.h>
static long etapes = 100000;        double pas;
#define NB_THREADS 2
void main ()
{               int i; double x, pi, sum[NB_THREADS];
                pas = 1.0/(double) etapes;
                omp_set_num_threads(NB_THREADS)
#pragma omp parallel
{               double x; int id;
                id = omp_get_thread_num();
                for (i=id, sum[id]=0.0;i< etapes; i=i+NB_THREADS)
               {
                        x = ((double)i+0.5)*pas;
                        sum[id] += 4.0/(1.0+x*x);
                }
}
                for(i=0, pi=0.0;i<NB_THREADS;i++)
                        pi += sum[i] * pas;
}
```

Programmes SPMD :

Chaque thread exécute le même code avec l'identificateur de thread spécifiant le travail spécifique du thread.

Program PI OpenMP: Work sharing construct

```
#include <omp.h>
static long etapes = 100000;        double pas;
#define NB_THREADS 2
void main ()
{               int i;          double x, pi, sum[NB_THREADS];
                pas = 1.0/(double) etapes;
                omp_set_num_threads(NB_THREADS)
#pragma omp parallel
{               double x;     int id;
                id = omp_get_thread_num();       sum[id] = 0;
#pragma omp for
                for (i=id;i< etapes; i++){
                        x = ((double)i+0.5)*pas;
                        sum[id] += 4.0/(1.0+x*x);
                }
}               for(i=0, pi=0.0;i<NB_THREADS;i++)
                        pi += sum[i] * pas;
}
```

Programme PI OpenMP: clause privée et section critique

```
#include <omp.h>
static long etapes = 100000;        double pas;
#define NB_THREADS 2
void main ()
{           int i;      double  x, sum, pi=0.0;
            pas = 1.0/(double) etapes;
            omp_set_num_threads(NB_THREADS)
#pragma omp parallel private (x, sum)
{
            id = omp_get_thread_num();
            for (i=id,sum=0.0;i< etapes;i=i+NB_THREADS){
                        x = ((double)i+0.5)*pas;
                        sum += 4.0/(1.0+x*x);}
#pragma omp critical
            pi += sum*pas;
}
}
```


Programme PI OpenMP : For parallèle + réduction

```
#include <omp.h>
static long etapes = 100000;        double pas;
#define NB_THREADS 2
void main ()
{           int i;      double x, pi, sum = 0.0;
            pas = 1.0/(double) etapes;
            omp_set_num_threads(NB_THREADS)
#pragma omp parallel for reduction(+:sum) private(x)
            for (i=1;i<= etapes; i++){
                        x = ((double)i-0.5)*pas;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = pas * sum;
}
```

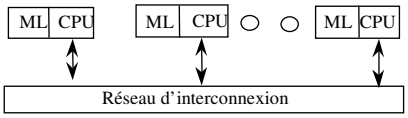
OpenMP ajoute 2 à 4 lignes de code

SECTIONS CRITIQUES

```
#pragma omp parallel private (x, sum)
{
        id = omp_get_thread_num();
        for (i=id,sum=0.0;i< etapes;i=i+NB_THREADS){
                x = ((double)i+0.5)*pas;
                sum += 4.0/(1.0+x*x);
        }
#pragma omp critical
        pi += sum
```

pi est partagé et sum (chaque processeur) est privé.
Chaque itération pi+=sum (chaque processeur) doit être atomique.

Approche MPI



- Initialiser l'environnement
- Identifier le nombre d'ordinateurs et les identifier
- Calcul dans chaque ordinateur, avec (ou sans) communication de données intermédiaires
- Opération globale pour obtenir le résultat

```
for (i=1;i<= etapes; i++){
        x = ((double)i-0.5)*pas;
        sum = sum + 4.0/(1.0+x*x)
```

-Répartir les itérations sur différents ordinateurs
-Réduction des sommes partielles dans une variable globale

Programme PI : version MPI

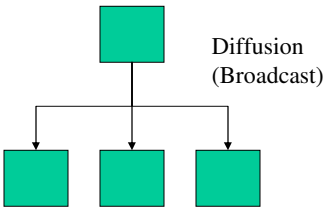
```
#include <mpi.h>
#include <math.h>
int main(argc, argv)
int argc; char *argv[];
{       int done=0, n=100000, myid, numprocs, i;
       double mupi, pi, h, sum, x;
MPI_Init (&argc, &argv)
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Bcast(&n, 1,MPI_INT, 0, MPI_COMM_WORLD);

h=1.0/(double) n; sum=0.0;
for (i=myid+1;i<= n; i+=numprocs)
{
                x = ((double)i-0.5)*h;
                sum+ = 4.0/(1.0+x*x);
}
mypi = pas * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);

if (myid==0)
            printf();
}
MPI_Finalize();
Return 0;
}
```

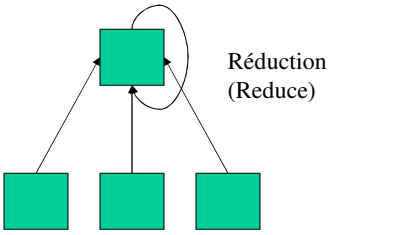

Primitives MPI utilisée dans PI

- MPI_Init (&argc, &argv)
  - Initialise l'environnement MPI
- MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  - Fournit le nombre de processeurs
- MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  - Chaque processus reçoit son rang dans le groupe associé avec un communicateur. Chaque processus reçoit une valeur **myid** différente
- MPI_Finalize
  - Clôt l'environnement MPI

Primitive MPI globale dans le programme PI

- MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  - La valeur de n est diffusée à chaque processus. Communication collective.
- MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);





Pi= Mypi(0)+ Mypi(1)+ Mypi(2)+ Mypi(2)