Parallélisme sur architectures à mémoires partagées

Bastien DI PIERRO

Université Claude Bernard Lyon 1

20 octobre 2016

Principe du parallélisme

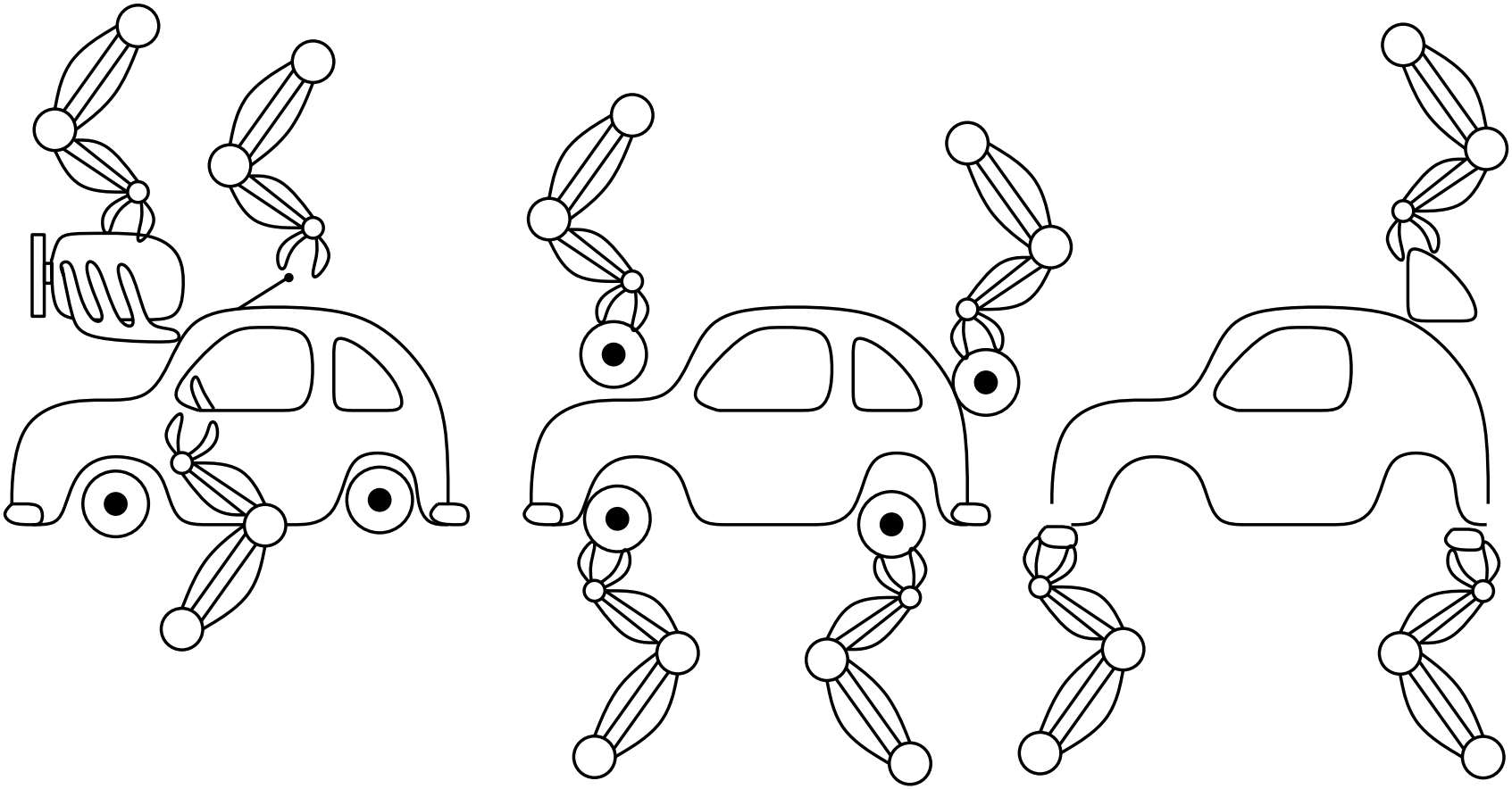
- Omniprésence des architectures multicoeurs/multiprocesseurs
- Parallélisme répond aux besoins des utilisateurs et aux capacités matérielles
- Programmation : mélange d'algorithmie et ingénierie
  - Une tâche $\Rightarrow$ plusieurs travailleurs ? (algo)
  - Travailleurs au même endroit ? séparés ? (ingé)
  - Communication / coordination ? (ingé)
  - Toujours occupés ? (algo)
  - Qui fait quoi ? quand ? où ? (algo)
- Algo = Papier + Crayon, ingé = machine

Principe du parallélisme

Design d'algorithme parallèle ("Introduction to parallel computing", Gramma *et al.*, Addison Wesley):

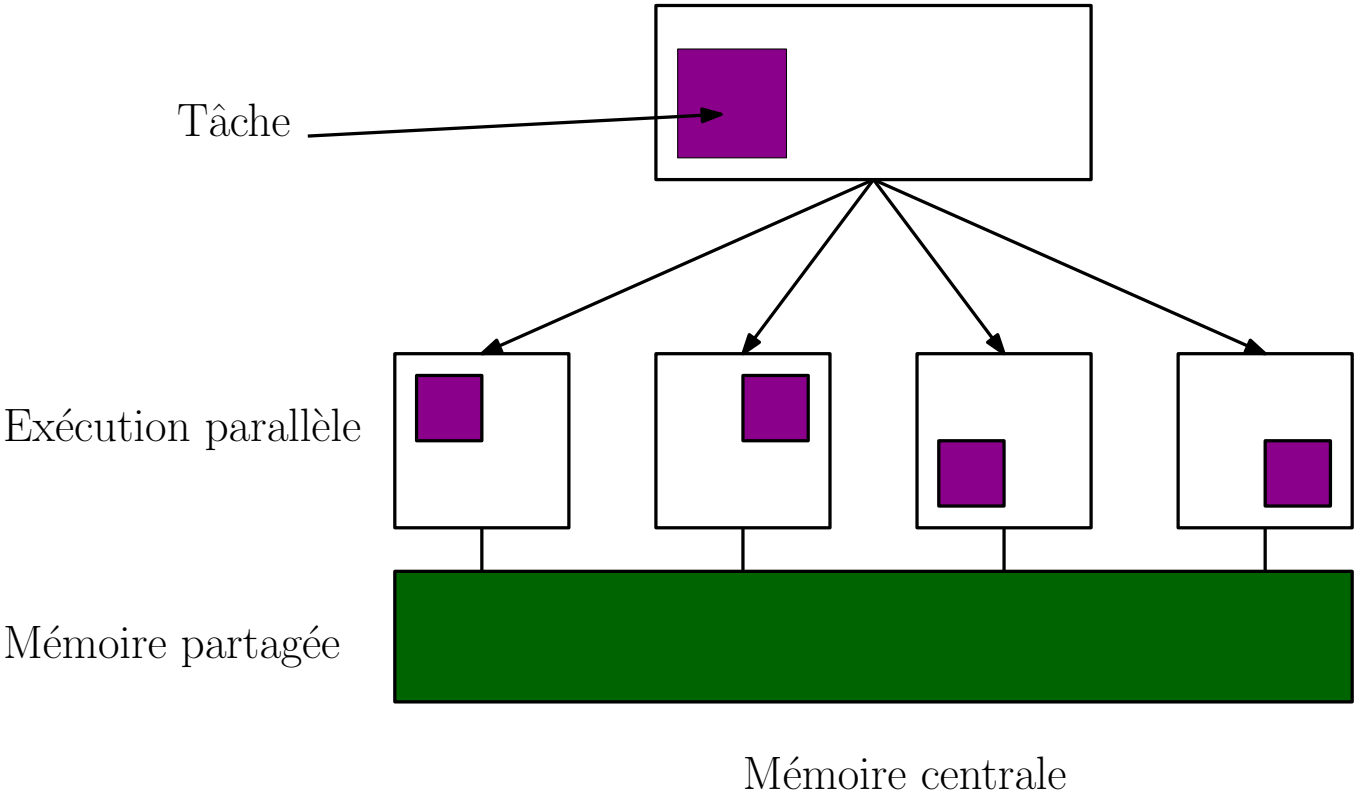
1. Décomposition en tâches (indivisibles) de différentes tailles
2. Liste de dépendance entre tâches et évaluation de leurs tailles
3. Routage des tâches sur les unités de calculs :
   - Planification / Synchronisation par rapport aux dépendances
   - Évite les temps d'attente
   - Minimisation du temps total
4. Distribution des entrées/sorties (clavier, ecran, fichier)
5. Gestion des accès aux données globales

Exemple : chaîne de production



Mémoire partagée

Toutes les unités de calcul accèdent à la **même mémoire**



.

Mémoire partagée

Toutes les unités de calcul accèdent à la **même mémoire** :

- Communication, synchronisation, gestion des tâches, des dépendances, etc ... se font de manière implicite !

- Les paradigmes de programmation se focalisent sur :
  - le lancement des unités de calculs parallèles
  - le partage des données
  - la synchronisation (ordre des tâches ...)
- Inconvénient : on est très vite limité (nombre d'unités de calculs, mémoire ...)

Idée

- Programmation très simple
- Saupoudrer un programme **séquentiel** pour aider le compilateur à paralléliser.
- Si le compilateur ne supporte pas l'openMP : code séquentiel par construction !
- Portabilité : indépendante du compilateur
- Langages supportés : Fortran, C, C++ (,légende : Matlab)

Bases

- Inclure des directives de compilation simples, standards et faciles
  (sont vues comme des commentaires si le compilateur ne supporte pas !)

```
#pragma omp nom_directive [option]
```

- Inclure les bibliothèques dans les headers

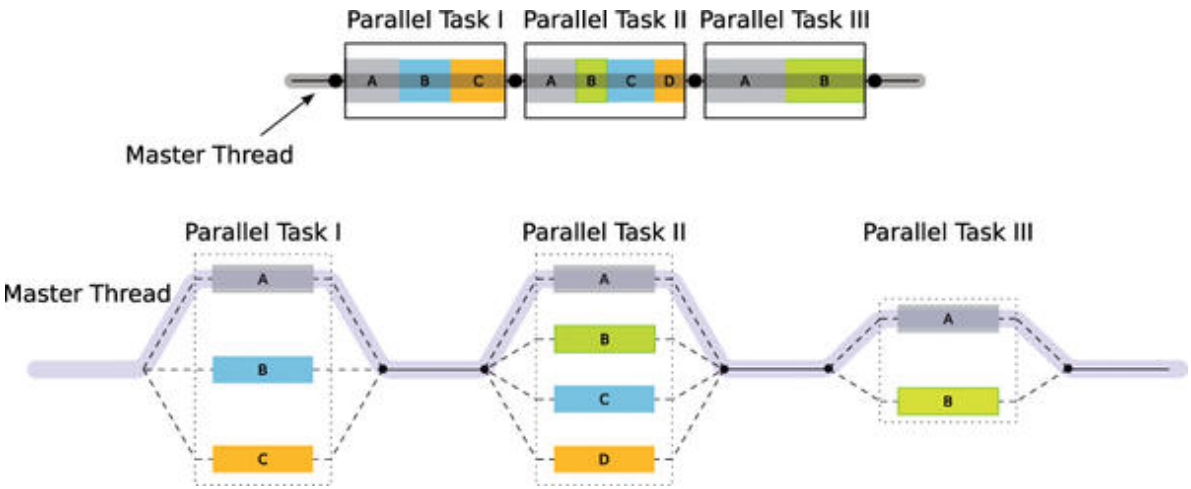
```
#ifndef _OPENMP
#include <omp.h>
#endif
```

- Compiler en activant l'openMP

```
gcc monprog.c -o progPara -fopenmp
```

Exécution parallèle

- Unité de calcul : *threads*
- *thread* “maitre” (principal) : identifiant 0
  Est actif durant toute l'exécution du programme (parallèle ou non)
- *threads* “esclave” : identifiants >0
  Sont actifs uniquement en région parallèle
- Région parallèle exécutée par une équipe de *threads* (fork-join)



Exécution parallèle

- On demande l'ouverture de région parallèle :

```
printf("Je suis le master et je suis tout seul");
#pragma omp parallel
{
  printf("Je suis le thread %d sur %d\n",omp_get_thread_num(),
  omp_get_num_threads());
}
```

DEMO

- Identifiant des threads :
  - omp_get_thread_num() : id du thread
  - omp_get_num_thread() : nombre de threads total
- Sélection du nombre de threads :
  - export OMP_NUM_THREADS=8 (dans le shell)
  - omp_set_num_thread(8)

Portée des variables

- La mémoire globale est partagée, on peut sécuriser l'accès aux données
- Dans une région parallèle, une donnée peut être :
  - partagée (shared) : connue et modifiable par tous les threads
  - privée (private) : accessible seulement au thread local

```
int A[12], depart, i;

omp_set_num_thread(4);

#pragma omp parallel private(i) shared(A)
{
   depart = omp_get_thread_num()*3;
   for(i=depart; i<depart+3; i++)
   {
     A[i] = i*(i-1) + 1:
   }
}
```

Portée des variables

- Si rien n'est précisé :
  - Une variable déclarée en dehors de la région parallèle est shared
  - Une variable déclarée dans la région parallèle est private

```
int a =1;
#pragma omp parallel
{
  int b = 0;
  // a est shared, b est private !
}
```

- En dehors de la région parallèle, une variable private a la valeur du master

```
int x = 1;
#pragma omp parallel private(x)
{
  x = omp_get_thread_num() + 1;
  // ici x vaut 1 ou 2 ou 3 ... selon le thread
}
// ici x = 1 !
```

Accès concurrents

```
int x = 0;
#pragma omp parallel shared(x)
{
  x = omp_get_num_thread();
}
printf("x = %d \n",x);
// resultat totalement aleatoire !
```

Accès concurrents

```
int x = 0;
#pragma omp parallel shared(x)
{
  x = omp_get_num_thread();
}
printf("x = %d \n",x);
// resultat totalement aleatoire !
```

Accès concurrent = Mal

Firstprivate

- Private : un nouveau doublon de la variable est créé pour chaque thread. Mais le doublon n'est pas initialisé !
- Firstprivate : les variables (des threads) sont privées mais en plus sont toutes initialisées à la valeur en dehors de la section parallèle

Firstprivate

- Private : un nouveau doublon de la variable est créé pour chaque thread. Mais le doublon n'est pas initialisé !
- Firstprivate : les variables (des threads) sont privées mais en plus sont toutes initialisées à la valeur en dehors de la section parallèle
- Exemple :

```
int var1=42, var2=-14;
#pragma omp parallel private(var1), firstprivate(var2)
{
   printf("var1 = %d var2 = %d \n", var1, var2);
}
```

Donne :

var1 = 32711 var2 = -14
var1 = 38698 var2 = -14
var1 = 27841 var2 = -14
var1 = 29851 var2 = -14

Boucles parallèles

- Omniprésentes dans les codes de calculs

Boucles parallèles

- Omniprésentes dans les codes de calculs
- Si les boucles sont **indépendantes**
  $\Rightarrow$ Bonnes candidates pour la parallélisation !
- Découpage entre les threads
  $\Rightarrow$ Chaque thread prend un jeu de valeurs de i

Boucles parallèles

- Omniprésentes dans les codes de calculs
- Si les boucles sont **indépendantes**
  $\Rightarrow$ Bonnes candidates pour la parallélisation !
- Découpage entre les threads
  $\Rightarrow$ Chaque thread prend un jeu de valeurs de i
- OpenMP s'occupe du "découpage" !

```
int i, A[100], B[100], C[100];
#pragma omp parallel
{
    #pragma omp for
    for(i=0;i<100;i++)
    {
      C[i] = A[i]*B[i];
    }
}
```

DEMO

Attention aux accès concurents !

- Cas pathologique : la boucle n'est pas indépendante !

```
int i, A[100], B[100];
#pragma omp parallel
{
    #pragma omp for
    for(i=0;i<100;i++)
    {
      A[i] = B[i]*A[i-1];
    }
}
```

$\Rightarrow$ Catastrophe !

DEMO

Race condition

- Race condition : un système essaie de faire *en même temps* (parallèle) deux opérations sur la même donnée
- On ne peut pas maîtriser la façon dont une variable shared est utilisée en écriture

Race condition

- Race condition : un système essaie de faire *en même temps* (parallèle) deux opérations sur la même donnée
- On ne peut pas maîtriser la façon dont une variable shared est utilisée en écriture
- Exemple : somme

```
int i, A[100], somme;
#pragma omp parallel shared(somme)
{
  #pragma omp for
  for(i=0;i<100;i++)
  {
    somme += A[i];
  }
}
```

$\Rightarrow$ résultat aléatoire !

Race condition

- Race condition : un système essaie de faire *en même temps* (parallèle) deux opérations sur la même donnée
- On ne peut pas maîtriser la façon dont une variable shared est utilisée en écriture
- Exemple : somme

```
int i, A[100], somme;
#pragma omp parallel shared(somme)
{
  #pragma omp for
  for(i=0;i<100;i++)
  {
    somme += A[i];
  }
}
```

  $\Rightarrow$ résultat aléatoire !

- Solution : les threads doivent être **synchronisés**

Synchronisation : les barrières

- On doit spécifier au compilateur les sections à synchroniser
- Permet à tous les threads de s'attendre
- Nécessaire pour certaines opérations
- En pratique : peu utilisée

```
#pragma omp barrier
```

Synchronisation : les sections critiques

- Utilisation de sections dites “critiques”

Synchronisation : les sections critiques

- Utilisation de sections dites “critiques”
- Zones où les threads fonctionnent l’un après l’autre

Retour à la somme :

```
#pragma omp parallel shared(somme) firstprivate(psomme)
{
  #pragma omp for
  for(i=0;i<10;i++)
  {
    psomme += A[i];
  }
  // ici les threads doivent s'attendre
  // et faire le calcul dans l'ordre
  #pragma omp critical
  {
    somme += psomme;
  }
}
```

DEMO

Synchronisation : les sections critiques

- Utilisation de sections dites “critiques”
- Zones où les threads fonctionnent l’un après l’autre

  Retour à la somme :

```
#pragma omp parallel shared(somme) firstprivate(psomme)
{
  #pragma omp for
  for(i=0;i<10;i++)
  {
    psomme += A[i];
  }
  // ici les threads doivent s'attendre
  // et faire le calcul dans l'ordre
  #pragma omp critical
  {
    somme += psomme;
  }
}
```

DEMO

- Problème : les threads risquent de s’attendre !

Opération de réduction

- Réduction : accumuler les contributions de chaque thread à la fin d'une boucle
- Permet des opérations globales avec des variables privates

Opération de réduction

- Réduction : accumuler les contributions de chaque thread à la fin d'une boucle
- Permet des opérations globales avec des variables privates
- On doit spécifier l'opérateur *et* la variable
- Encore la somme !

```
int somme = 0;
#pragma omp parallel reduction(+:somme)
{
    #pragma omp for
    for(i=0; i<=10; i++)
    {
        somme += i;
    }
}
```

Opération de réduction

- Réduction : accumuler les contributions de chaque thread à la fin d'une boucle
- Permet des opérations globales avec des variables privates
- On doit spécifier l'opérateur *et* la variable
- Encore la somme !

```
int somme = 0;
#pragma omp parallel reduction(+:somme)
{
    #pragma omp for
    for(i=0; i<=10; i++)
    {
        somme += i;
    }
}
```

- Opérateurs de réductions possible :
  - arithmétiques : +,-,*,/
  - logiques : &&, ||

Ordonnancement des boucles

- Un boucle parallèle est découpée entre les threads en **chunks**
  chunk = région contigüe de taille donnée.

  ```
  for ( i =1; i <=12; i++)
  ```

  sur 4 threads :
  $\{ \{1,2,3\} , \{4,5,6\} , \{7,8,9\} , \{10,11,12\} \}$

Ordonnancement des boucles

- Un boucle parallèle est découpée entre les threads en **chunks**
  chunk = région contigüe de taille donnée.

  ```
  for(i=1;i<=12;i++)
  ```

  sur 4 threads :
  $\{ \{1,2,3\} , \{4,5,6\} , \{7,8,9\} , \{10,11,12\} \}$
- En région parallèle une boucle peut être découpée de plusieurs façons :
  - Chaque thread effectue le même nombre d'itétations (statique)

```
#pragma omp parallel for schedule(static)
for(i=0;i<N;i++)
  // travail identique pour tous les threads
```

Ordonnancement des boucles

- Un boucle parallèle est découpée entre les threads en **chunks**
  chunk = région contigüe de taille donnée.

  ```
  for(i=1;i<=12;i++)
  ```

  sur 4 threads :
  $\{ \{1,2,3\} , \{4,5,6\} , \{7,8,9\} , \{10,11,12\} \}$
- En région parallèle une boucle peut être découpée de plusieurs façons :
  - Chaque thread effectue le même nombre d'itétations (statique)

    ```
    #pragma omp parallel for schedule(static)
    for(i=0;i<N;i++)
      // travail identique pour tous les threads
    ```

  - Le nombre d'itérations dépend de la quantité de travail (dynamique)

    ```
    #pragma omp parallel for schedule(dynamic)
    for(i=0;i<N;i++)
      // travail variable pour tous les threads
    ```

- Thread safe : morceau de programme qui fonctionne correctement sur plusieurs threads
- Nécessite l'intervention de régions critiques (pertes de performances)
- Les entrées/sorties ne sont pas thread safe (accès concurrent)
- Fonctions non thread safe :
  - rand() (même nombre sur plusieurs threads)
  - ctime() (threads ne connaît pas l'heure)
  - system() (l'appel au système est séquentiel !)
  - ...

Placement des régions parallèles

```
#pragma omp for
for(i=0;i<10;i++){
    for(j=0;j<100;j++){
        for(k=0;k<1000;k++){
            a[i][j][k] = ...
        }
    }
}
```

- Mauvais si $N_{threads}\%10 \neq 0$
- Bonne localité spatiale

```
for(i=0;i<10;i++){
    for(j=0;j<100;j++){
#pragma omp for
        for(k=0;k<1000;k++){
            a[i][j][k] = ...
        }
    }
}
```

- Optimisation parallèle
- Surcoût accès memoire

Placement des régions parallèles

```
#pragma omp for
for(i=0;i<10;i++){
    for(j=0;j<100;j++){
        for(k=0;k<1000;k++){
            a[i][j][k] = ...
        }
    }
}
```

```
for(i=0;i<10;i++){
    for(j=0;j<100;j++){
#pragma omp for
        for(k=0;k<1000;k++){
            a[i][j][k] = ...
        }
    }
}
```

- Mauvais si $N_{threads}\%10 \neq 0$
- Bonne localité spatiale

- Optimisation parallèle
- Surcoût accès memoire

- Cas où gain parallélisme $<$ surcoût
- Possibilité de rendre le calcul parallèle sous condition :

```
#pragma omp parallel for if (n>100)
for(i=0 ; i<N ; i++)
    a[i] = ...
```

False Sharing

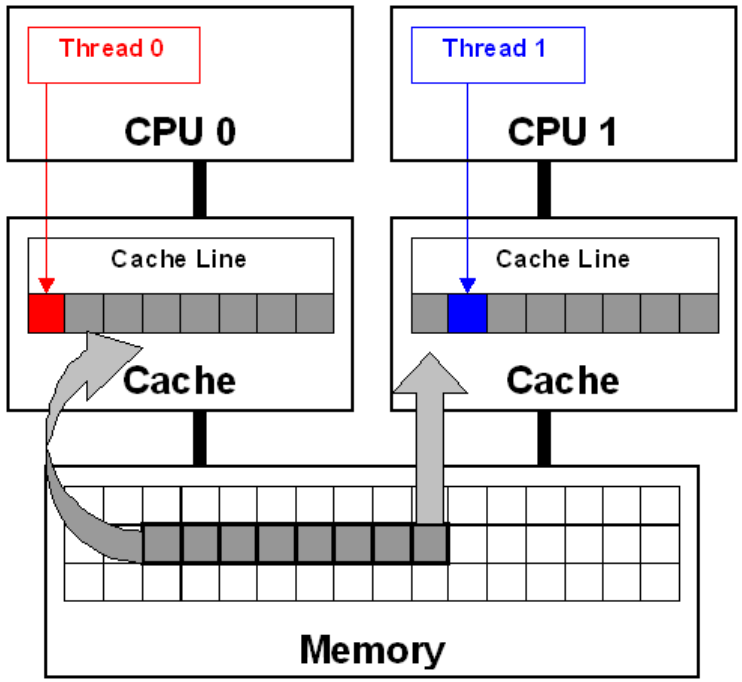
- Plusieurs threads agissent sur le même tableau

+

- Le tableau est stocké dans le cache

  ⇒ False Sharing : fort ralentissement dû à la cohérence des caches !

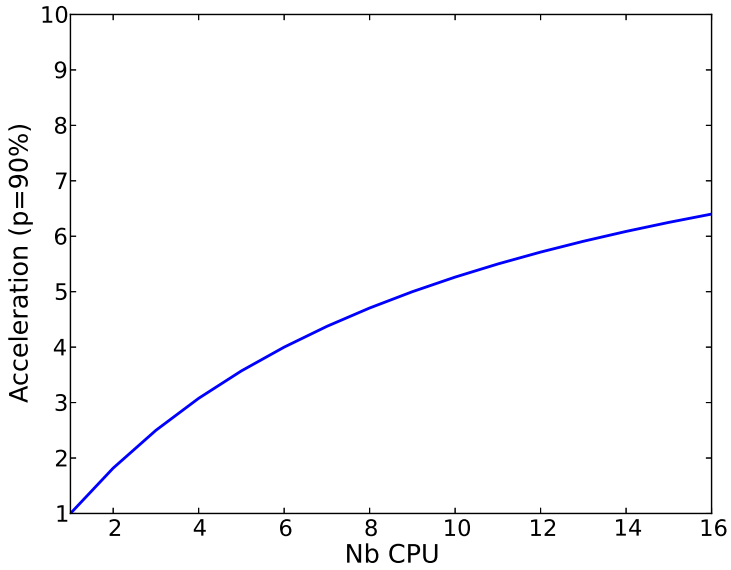
```
int localTab[nbThread];

#pragma omp parallel
{
    localtab[monThread] = ...
}
```

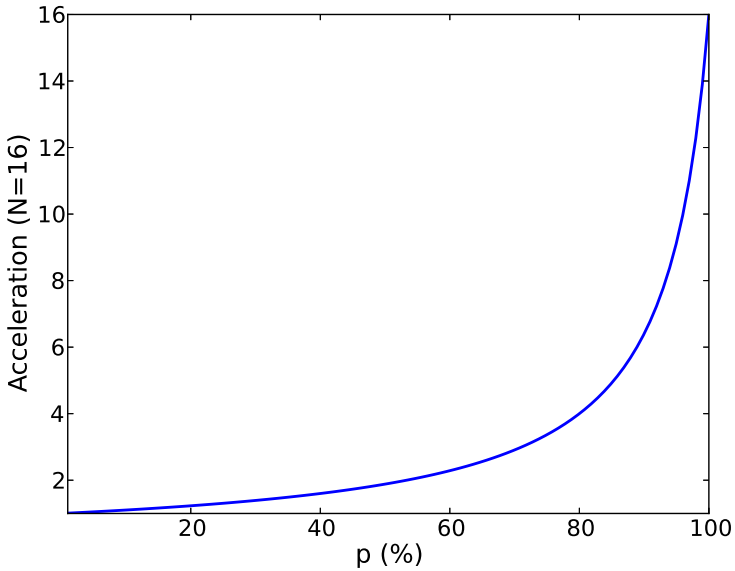


Performances

- La performance d'un code parallèle est limitée par la portion séquentielle !
- Suit la loi d'Amdhal : $S = \frac{N}{N(1-p)+p}$

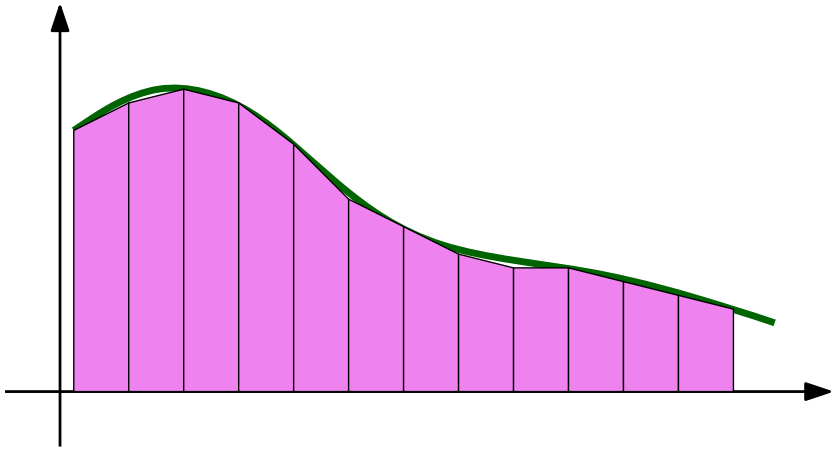
  p : portion parallèle , N : nombre de threads





Calcul de pi

$$\pi = \int_0^1 \frac{4.0}{1+x^2} dx$$



```
int N=1000000000,i;
double dx = 1.0/((double) N);
double pi=0.,x1,x2;

#pragma omp parallel for reduction(+:pi) private(x1,x2)
for( i=0 ; i<N ; i++)
{
   x1 = ( (double) i )*dx;
   x2 = ( (double) i+1)*dx;
   pi += 4.0*dx*0.5*(1.0/(1.0+x1*x1)+1.0/(1.0+x2*x2));
}
printf("pi = %.12lf\n", pi );
```

Algorithme de Luhn (carte bancaire)

numero	4	4	1	7	1	2	3	4	5	6	7	8	9	1	1	6
1^{er}	4		1		1		3		5		7		9		1	
x2	8		2		2		6		10		14		18		2	
2^{eme}		4		7		2		4		6		8		1		6
somme		12		9		4		10		16		22		19		8
total%10																0

```
int A[16]={4,4,1,7,1,2,3,4,5,6,7,8,9,1,1,6};
int somme=0, resultat=0,i;
#pragma omp parallel firstprivate(somme)
{
  #pragma omp for
   for ( i = 0 ; i<16 ; i+=2 )
   {
      somme += 2*A[i] + A[i+1];
   }
   #pragma omp critical
   {  resultat += somme; }
}
resultat = resultat % 10;
```

Conclusion

- Facile à programmer

- Pas si facile d'avoir de bonnes performances !
- Problème architecture, d'accès concurrent, de cohérences des caches ...
- Gestion de l'ordonnancement des tâches