

# Programmation parallèle

---

Ateliers Pratiques

**Prof. Mohamed AKIL & Ramzi MAHMOUDI**

**AMINA Workshop 2010**

# Agenda



- Introduction
- Parallélisez – un vrai besoin
- Parallélisez – une démarche organisée
- Mise en pratique

# Introduction

- La première question à se poser, c'est savoir si la parallélisation de l'application est nécessaire ?
- Écrire un logiciel séquentiel est déjà du travail, souvent difficile :  
**la parallélisation le rendra encore plus dur.**
- Il y a eu beaucoup de progrès du matériel informatique (processeurs de 3.7 GHz etc.)
- Pourtant il existe toujours des applications scientifiques qui consomment "trop" de ressources en temps : calculs (processeur) et cycles d'accès mémoire
- Pour celles-ci, la seule solution, pour des raisons techniques ou économiques, reste la '**parallélisation**'

# Parallélisez – un vrai besoin

Calcul parallèle est  
Omniprésent

## ● Multi-tâches :

- Augmenter la performance lors de l'affectation de tâches distinctes pour répondre à des événements non-déterministes
- Les applications interactives exigent de 'faire beaucoup de choses en parallèle'
- Aujourd'hui, la plupart des processeurs offrent de l'exécution en parallèle ('multi-core')

## ● Calcul distribué :

- Le calcul est intrinsèquement distribués parce que l'information est distribuée
- Exemple : gestion d'une entreprise ou de banque à l'échelle internationale (word-wide company)
- Les questions : la communication entre les plates-formes, la portabilité et la sécurité.

## ● Performance de calcul:

- Toutes les techniques implémentées aujourd'hui dans nos machines de bureau ont été développés dans les supercalculateurs de plusieurs depuis des années
- Les application de simulation sur des superordinateurs sont principalement numériques
- Les grands défis : la cosmologie, le repliement des protéines, **l'imagerie médicale**, la prédiction des tremblements de terre, le climat ... mais aussi : la simulation des armes nucléaires

# Parallélisez – une démarche organisée

Ecrire un algorithme parallèle

1

Injecter du parallélisme

2

Evaluer les performances

3

Optimiser

4



# Ecrire un algorithme parallèle

Step 1

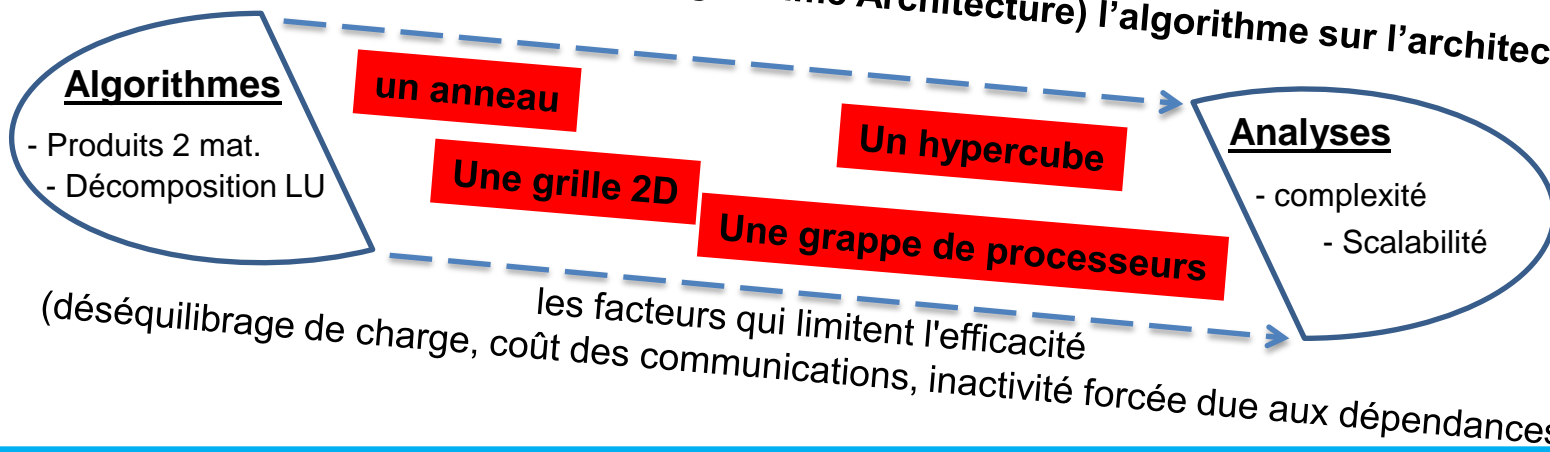
Step 2

Step 3

Step 4

- L'algorithmique parallèle emprunte beaucoup à l'algorithmique classique dans sa problématique: **conception, analyse, étude de la complexité.**
- Les résultats sont quelquefois moins précis, car les problèmes sont plus récents, et aussi plus difficiles.
- Fondamentalement, il y a quand même une nouvelle dimension, un **degré de liberté** supplémentaire avec **l'exploitation simultanée de plusieurs ressources.**

Mapper **efficacement** (adapter – Adéquation Algorithme Architecture) l'algorithme sur l'architecture



# Ecrire un algorithme parallèle

Step 1

Step 2

Step 3

Step 4

1

Une **topologie simple** ( anneau de processeurs ) permet de **concevoir** et d'écrire des **algorithmes parallèles**, sans rien connaître aux architectures de machines.  
( communications globales, produit matrice-vecteur ...)

2

Les **réseaux d'interconnexion**, les **mécanismes de routage** : Prendre en considération les propriétés topologiques du réseau ( cas de l'hypercube diffère des grilles toriques 2D de processeurs.)

3

**Équilibrage de charge pour plate-forme hétérogène (ou homogène)** . Autant l'équilibrage de charge 1D reste facile, autant l'équilibrage de charge 2D s'avère complexe.

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

**70%**

Comprendre son application

**20%**

Maitriser les concepts  
de parallélisme

**10%**

Choisir API  
de threading

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Pourquoi des threads ?

### Avantages

- a. Meilleures performances - Une façon simple de tirer profit du multi-core.
- b. Meilleure utilisation des ressources - Réduit les temps d'attente, même sur des systèmes multi processeurs.
- c. Partage des données efficace - Il est plus facile de partager des données via l'accès direct à la mémoire que recourir aux changes de messages

### Risques

- a. L'application est plus complexe
- b. Le débogage est difficile (conflits d'accès aux données, verrouillages...)

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

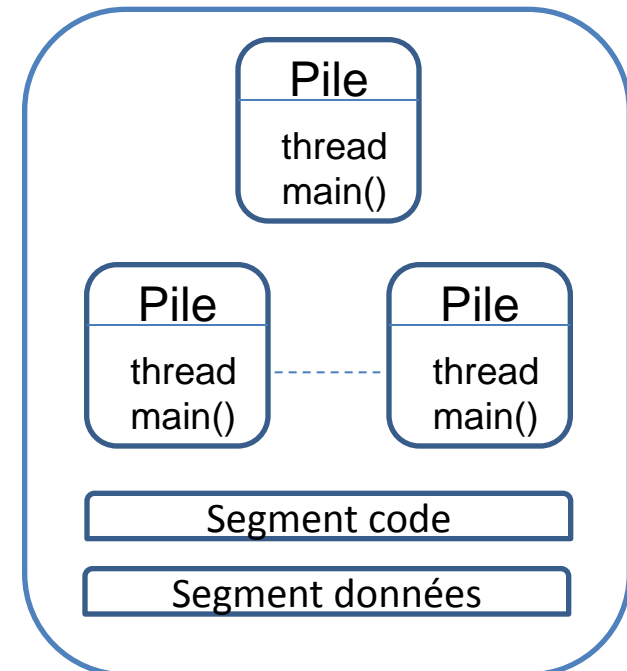
## ● Process vs threads ?

Les OS modernes chargent les programmes sous forme de process (Gestion ressources / Exécution ...)

Un process démarre depuis un point d'entrée sous forme d'un thread

Un thread peut créer par d'autres threads dans le contexte du même process : (Chaque thread dispose de sa propre pile)

Tous les threads à l'intérieur d'un même process partagent les segments de code et de données.



# Injecter du parallélisme

Step 1

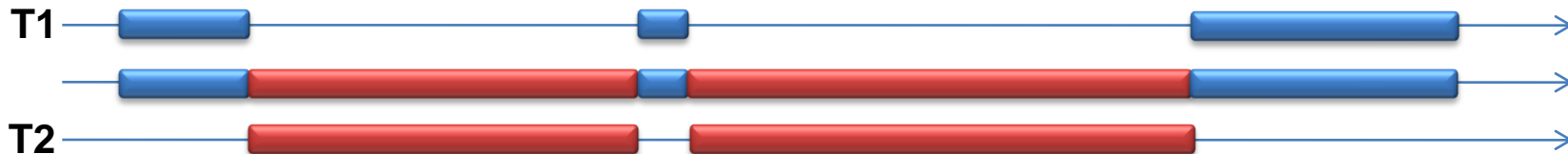
Step 2

Step 3

Step 4

## • Simultanéité vs Parallélisme ?

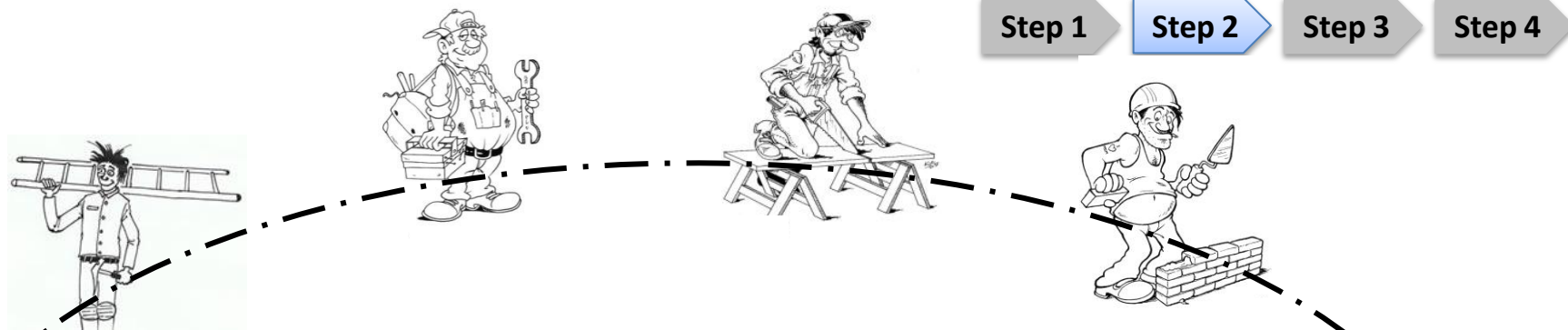
- Simultanéité : deux threads ou plus sont actifs à un instant donné (peuvent tourner sur un monocore)



- Parallélisme : deux threads ou plus tournent à un instant donné (multicore obligatoire)



# Injecter du parallélisme



## • Threading pour des fonctionnalités

Attribuer les threads aux différentes fonctions de l'application

- Méthode la plus facile car les chevauchements sont peu probables
- **Mais** il pourrait y avoir des problèmes de séquençement

Exemple: *construire une maison*

Maçon, menuisier, couvreur, plombier,...

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Threading pour les performances?

Améliorer les **performances des calculs** !!

Améliorer la **flexibilité** ou le **flux** !!



Dans la vie quotidienne, de nombreux exemples :

- Chaine de montage automobile : chaque ouvrier effectue une tâche qui lui est attribuée
- Recherche de morceaux de Skylab : diviser la zone de recherches
- Caisses de grandes surfaces : plusieurs caisses fonctionnent en parallèle

# Injecter du parallélisme

Step 1

Step 2

Step 3

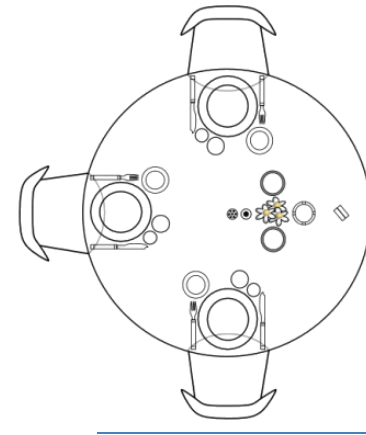
Step 4

## ● Flexibilité ?

**Effectuer des tâches élémentaires le plus rapidement possible**

Exemple : mise en place d'une table dans un restaurant

- ✓ Un pour les assiettes
- ✓ Un pour plier et poser les serviettes
- ✓ Un pour les couverts
- ✓ Un pour les verres



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

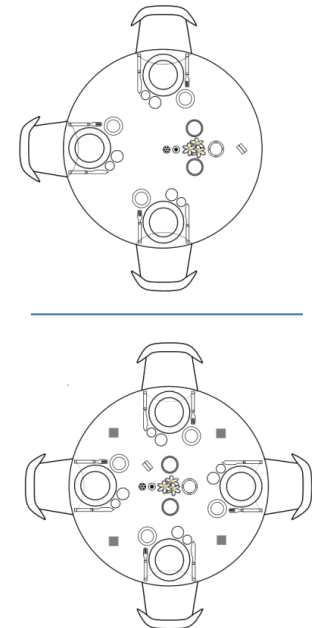
## ● Volume ?

**Terminer un maximum de tâches dans un même lapse de temps :**

plusieurs solutions – il faut tester pour trouver la meilleure.

Exemple : mise en place des tables d'un banquet

- ✓ Plusieurs serveurs, un par table
- ✓ Serveurs spécialisés pour la vaisselle, les couverts, les verres, etc.



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Les API de parallélisation

☺ **Bas niveau** - WinThreads et Posix

☺ **Encapsulation** – **OpenMP** et Thread **B**uilding **B**locks

☺ Cas **special clustering** - MPI (Message Passing Interface)

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● OpenMP – un choix judicieux

### **Mise en œuvre rapide** –

Encapsulation de threading à base de pragmas de compilation en C++ et directives en Fortran

### **'Désactivable'** –

Donc l'application peut tourner en mono-thread sans difficulté de mise en œuvre

### **Encapsule** – la mécanique de bas niveau

**Portable** – C++ et Fortran ainsi que Windows, Linux et Mac

**Gratuit**



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Objectifs

1. Comprendre comment paralléliser une application avec des directives OpenMP de base
2. Utiliser la synchronisation OpenMP pour coordonner l'exécution des threads et les accès à la mémoire



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Qu'est-ce qu'OpenMP ?

- ☺ Directives de compilation pour la programmation multithread
- ☺ Sa mise en œuvre est facile en Fortran et C/C++
- ☺ Support des modèles de parallélisation de données
- ☺ Parallélisation incrémentale
- ☺ Regroupe le fonctionnement série et parallèle dans une seule source

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

[www.openmp.org](http://www.openmp.org)

[www.openmp.fr.nf](http://www.openmp.fr.nf)

# Injecter du parallélisme

Step 1

Step 2

Step 3

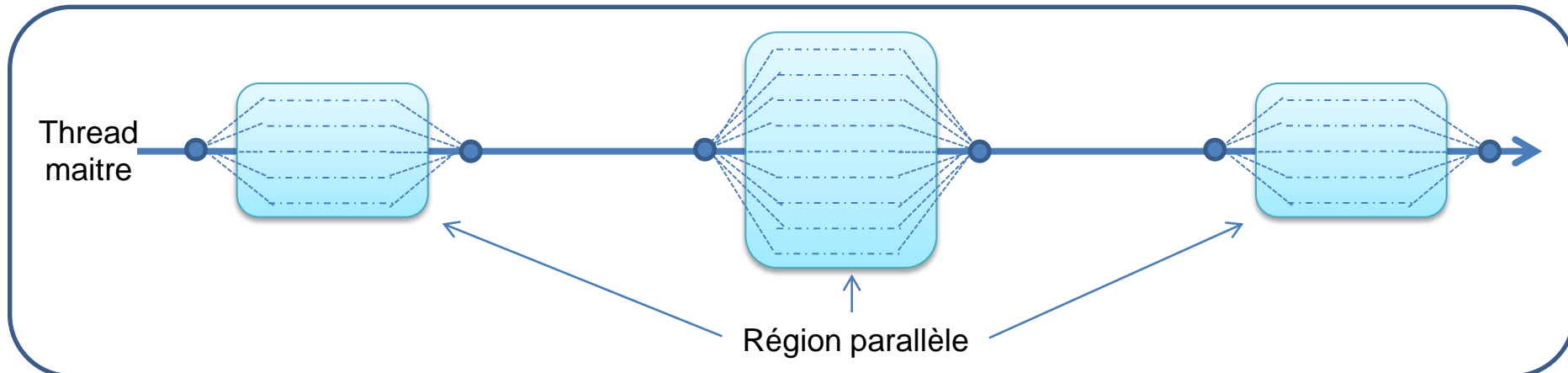
Step 4

## ● Modèle de programmation avec OpenMP?

Parallélisme par séparation-regroupement :

a. Le Thread maître engendre un jeu de threads selon les besoins

b. Le parallélisme est ajouté d'une façon incrémentale: le programme séquentiel se transforme progressivement en programme parallèle



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Syntaxe des pragma OpenMP

La plupart des commandes OpenMP se présente sous forme de directives ou pragmas. Pour C et C++, les pragmas prennent la forme suivante :

```
#pragma omp directives [clause [clause]...]
```

# Injecter du parallélisme

Step 1

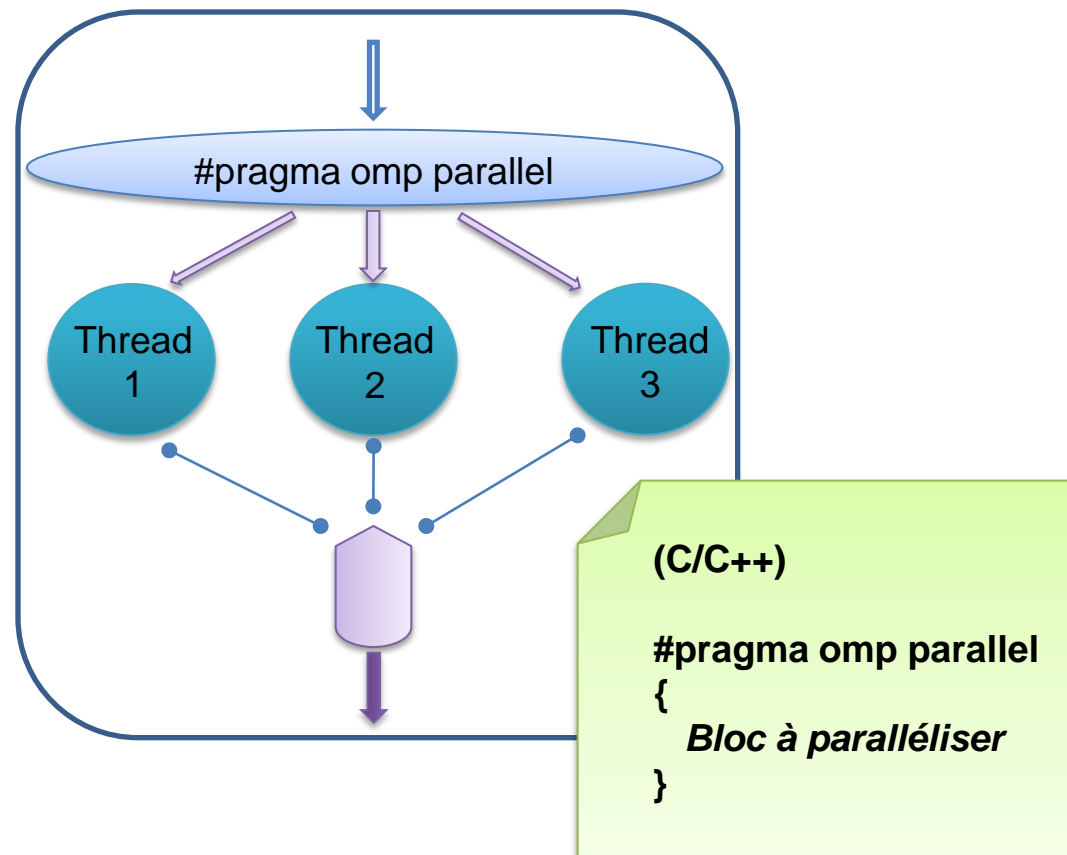
Step 2

Step 3

Step 4

## ● Région parallèles

- ☺ Une région parallèle est définie sur plusieurs blocs de code structuré.
- ☺ Les threads sont créés 'parallèle'.
- ☺ Les threads bloquent en fin de région.
- ☺ Les données sont partagées par les threads à moins que cela ne soit spécifié autrement.



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Combien de threads ?

☺ Définir la variable d'environnement pour le nombre de threads

```
set OMP_NUM_THREADS = 2
```

☺ Il n'y a pas de valeurs standard par défaut.

☺ Sur beaucoup de systèmes : # de threads = # de processeurs

☺ Les compilateurs Intel utilisent ceci par défaut

# Injecter du parallélisme

Step 1

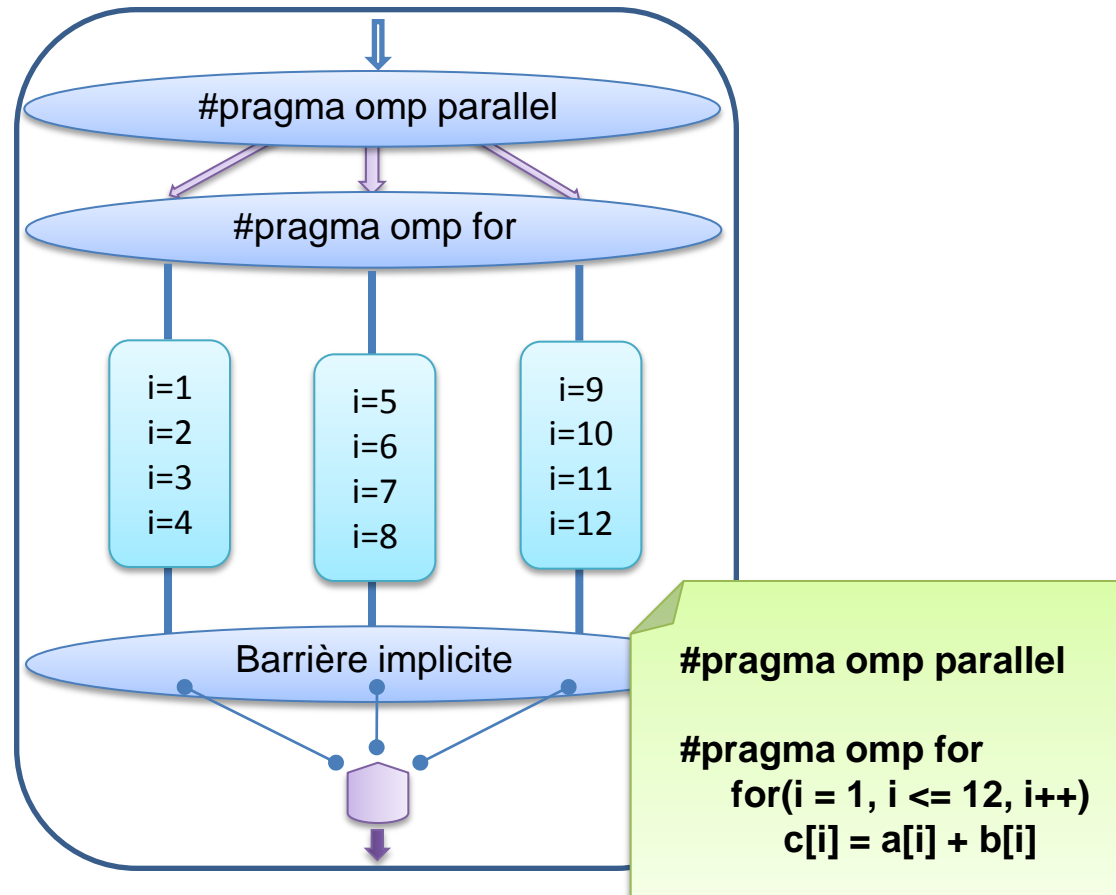
Step 2

Step 3

Step 4

## ● Partage des tâches

- ☺ Partager les itérations de la boucle entre les threads.
- ☺ Doit se trouver dans la région parallèle
- ☺ Doit précéder la boucle
- ☺ Les threads se voient attribués un ensemble indépendant d'itérations
- ☺ Les threads doivent attendre la fin du partage de travail



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## Combiner les pragmas

☺ Ces deux segments de code sont équivalents

```
#pragma omp parallel  
  
#pragma omp for  
  for(i = 1, i <= max, i++)  
    res[i] = huge();
```

```
#pragma omp parallel for  
  for(i = 1, i <= max, i++)  
    res[i] = huge();
```

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Environnement de données

OpenMP utilise un modèle de **mémoire partagée**

- ☺ Avec un système à mémoire partagée, on a la possibilité d'utiliser OpenMP comme moyen de parallélisation.
- ☺ Un programme parallèle consiste alors en un groupe de fils d'exécution (*threads*).
- ☺ Ces fils d'exécution partageront le même espace d'adressage dans la mémoire.
- ☺ Les variables globales sont partagées par les threads (C/C++: Variables globales, static)



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Environnement de données

### Mais TOUT n'est pas partagé...

- ☺ Les variables sur la pile dans des fonctions appelées dans des régions parallèles sont **PRIVATE**
- ☺ Les variables automatiques dans une phrase sont **PRIVATE**
- ☺ Les indices de boucles sont **PRIVATE** mais il existe des exceptions
- ☺ C/C+: L'indice de la première boucle dans des boucles imbriquées suite à une **#pragma omp for** est **PRIVATE** alors que les indices des boucles imbriquées sont partagés.



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## Attributs de portée de données

Le statut par défaut peut être modifié avec :

☺ Clause d'attributs de portée partagée par défaut

**Default (shared | none)**

☺ Définition des variables partagées

**Shared (varname,...)**

☺ Définition des variables privées

**Private (varname,...)**

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● La clause private

Reproduit les variables pour chaque thread :

- ☺ Les variables sont non-initialisées; les objets C++ sont construits par défaut
- ☺ Toute valeur en dehors de la région parallèle est indéfinie.

ACCES INTERDIT AUX THREADS  
NON AUTORISES

```
void* work(float* c, int N)
{
    float x y; int i;
    #pragma omp parallel for private(x,y)
        for(i=0; i<N; i++) {
            x = a[i]; y = b[i];
            c[i] = x + y;
        }
}
```

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● La clause shared

- ☺ Définit explicitement les variables dont les données seront partagées par tous les threads.
- ☺ En définissant une variable comme étant **SHARED** (partagée), vous garantissez que chaque thread peut utiliser la variable, pas nécessairement en toute sécurité.
- ☺ Les variables sont partagées par défaut.

C[] est partagé par défaut mais il y a un risque de conflit

```
void* work(float* c, int N) {  
    float x, y; int k = calculK(N);  
    #pragma omp parallel for private(x,y) shared(k)  
        for(int i=0; i<N; i++) {  
            x = a[i]; y = b[i];  
            c[i] = x + y + k;  
        }  
}
```

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Exemple : produit vectoriel

```
float point_produit (float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
        for(int i=0; i<N; i++)
            { sum += a[i] * b[i] }
    return sum
}
```



Qu'est ce qui  
ne va pas ?

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Protéger les données partagées

```
float point_produit (float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++)
    {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Il est impératif de protéger l'accès aux données partagées



# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● La clause OpenMP Critical

```
#pragma omp critical [(lock_name)]
```

Définit une zone critique dans un bloc structuré

☺ Chaque thread attend son tour, un seul appel `consum()` ce qui protège RES de conflits d'accès.

☺ Le fait de nommer la zone critique `RES_lock` est optionnel

```
float RES;  
#pragma omp parallel  
{ float B;  
  #pragma omp for  
  for(int i=0; i<niters; i++)  
  {  
    B = big_job(i);  
    #pragma omp critical (RES_lock)  
    consum (B, RES);  
  }  
}
```

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● La clause OpenMP Reduction

reduction (op : list)

☺ Les variables dans “*list*” doivent être partagées dans la zone parallèle

☺ A l'intérieur de zones parallèles ou de partage de travail :

- ▒ Une copie privée de chaque variable dans la liste est créée et initialisée selon la nature de “op”
- ▒ Ces copies sont mises à jour localement par le thread
- ▒ En sortie de la zone concernée par la clause, les copies sont regroupées en une seule valeur via “op” et combinées avec la variable partagée.

# Injecter du parallélisme

Step 1

Step 2

Step 3

Step 4

## ● Exemple de Reduction

```
#pragma omp parallel for reduction(+:somme)
for(i=0; i<N; i++) {
    somme += a[i] * b[i];
}
```

- ☺ Un copie locale de *somme* pour chaque thread
- ☺ Toutes les copies locales de *somme* son regroupées puis stockées dans une variable “globale”

# Injecter du parallélisme

Step 1

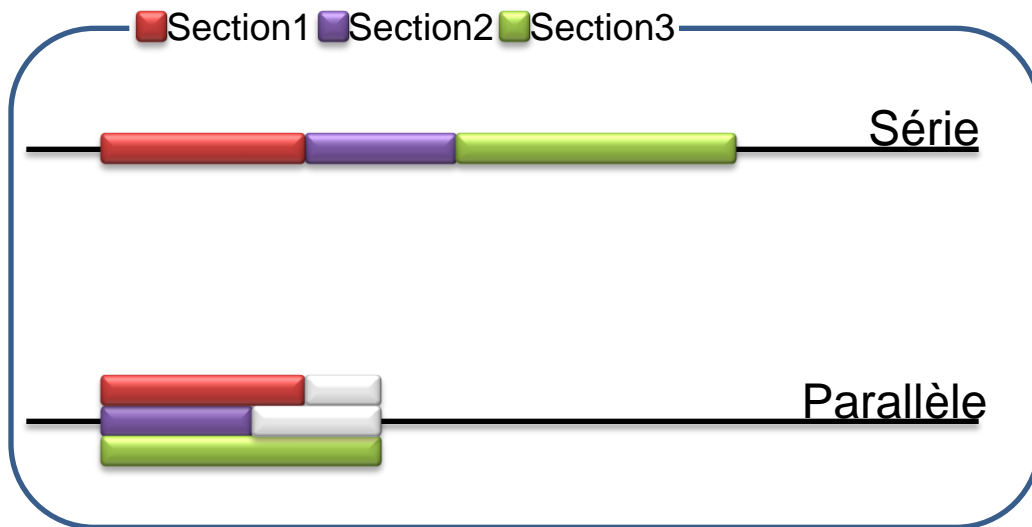
Step 2

Step 3

Step 4

## Sections parallèles

☺ Des sections de code indépendantes peuvent s'exécuter en parallèle



```
#pragma omp parallel sections  
{  
    #pragma omp section  
    phase1();  
  
    #pragma omp section  
    phase2();  
  
    #pragma omp section  
    phase3();  
}
```

# Injecter du parallélisme

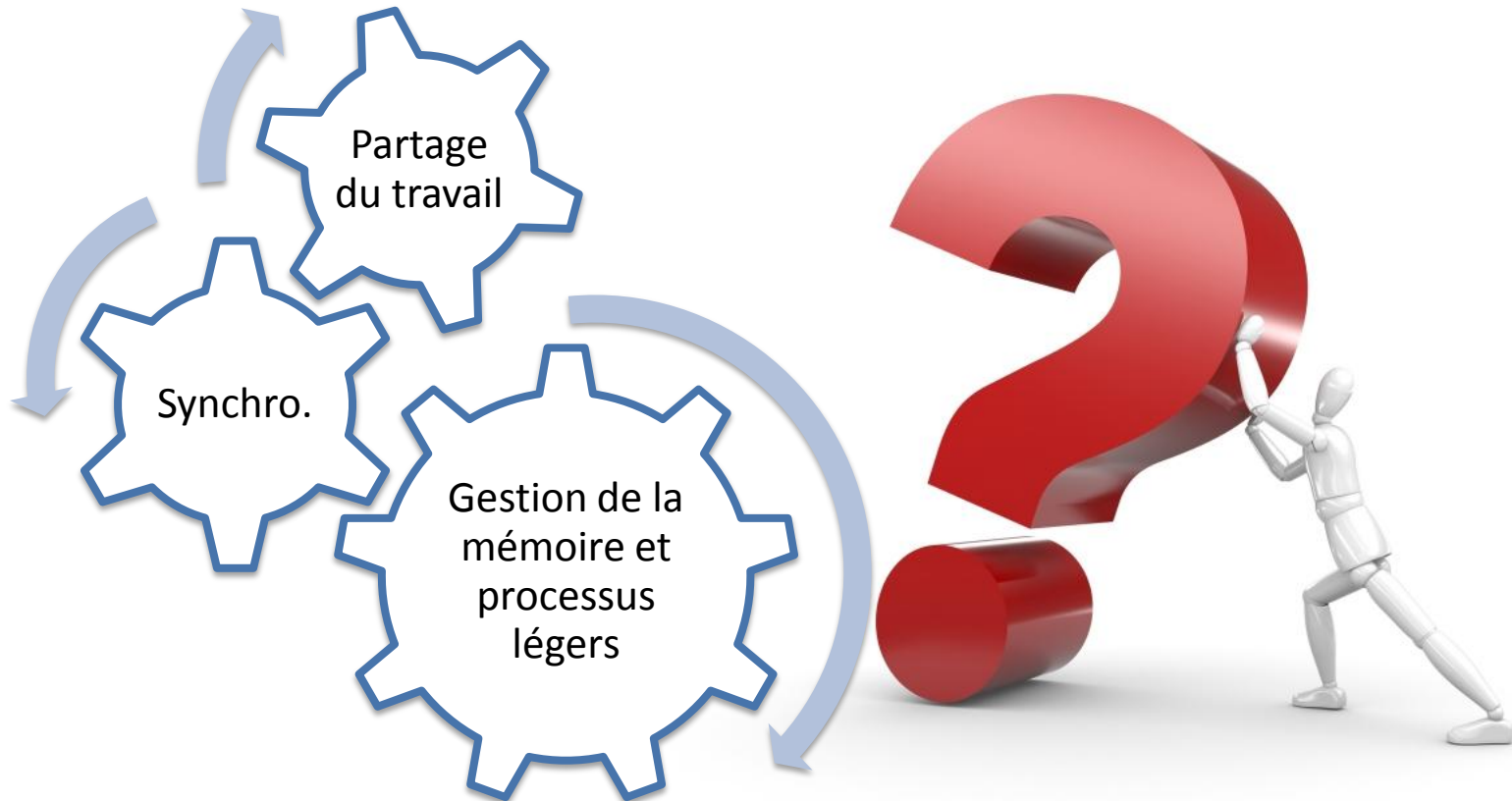
Step 1

Step 2

Step 3

Step 4

- Il reste beaucoup de chose à découvrir !



# Applications



# Evaluer les performances

Step 1

Step 2

Step 3

Step 4

## ● Outils d'évaluation des performances

- ☉ Déterminer le temps nécessaire pour exécuter chaque partie (procédure, fonction, bloc) du code
- ☉ Déterminer les sections critiques du code (sections qui posent problème)
- ☉ Pour analyser un code :
  - ▒ Rapport ou listing des compilateurs
  - ▒ Profiling (timers & profilers)
  - ▒ Hardware performance counters

# Evaluer les performances

Step 1

Step 2

Step 3

Step 4

## ● Rapport et listing des compilateurs

- Les compilateurs peuvent générer éventuellement des rapports d'optimisation et la liste des fichiers.
- Utilisez le « Loader Map » pour déterminer les bibliothèques chargées

### IA32/EM64T:

```
– <compiler> -opt-report {optimization, (Intel)}  
– <compiler> -S {listing (Intel)}
```

# Evaluer les performances

Step 1

Step 2

Step 3

Step 4

## ● Profiling : classification des analyses

### Informations :

Wall clock / CPU : temps passé en chaque fonction.  
HW counters : Caches misses, nombre d'instructions FLOPS ...

### Outils :

Timers  
Profilers  
Profile visualiser  
API to read/display HW counters info.

### Informations :

Trace file (raw)  
Timeline ( état de thread / process, communication, événement utilisateur prédéfini)

### Outils :

Trace generateur  
API instrumentation  
Outils pour lire / interpréter les trace files et visualiser

# Evaluer les performances

Step 1

Step 2

Step 3

Step 4

## Timers

☉ time :

```
$ gcc test.c -o testout  
$ time ./testout
```

Le résultat de testout .....

```
real 0m0.122s  
user 0m0.110s  
sys 0m0.010s
```

Routine	Type	Résolution	OS / compi.
<b>times</b>	<b>user/sys</b>	<b>1000</b>	<b>Linux/AIX IRIX/UNICOS</b>
<b>getrusage</b>	<b>wall/user/sys</b>	<b>1000</b>	<b>Linux/AIX/IRIX</b>
<b>gettimeofday</b>	<b>wall clock</b>	<b>1</b>	<b>Linux/AIX IRIX/UNICOS</b>
<b>rdtsc</b>	<b>wall clock</b>	<b>0.1</b>	<b>Linux</b>
<b>read_real_time</b>	<b>wall clock</b>	<b>0.001</b>	<b>AIX</b>
<b>system_clock</b>	<b>wall clock</b>	<b>(System)</b>	<b>Fortran 90 Intrinsic</b>
<b>system_clock</b>	<b>wall clock</b>	<b>(System)</b>	<b>MPI Library (C &amp; Fortran)</b>

# Evaluer les performances

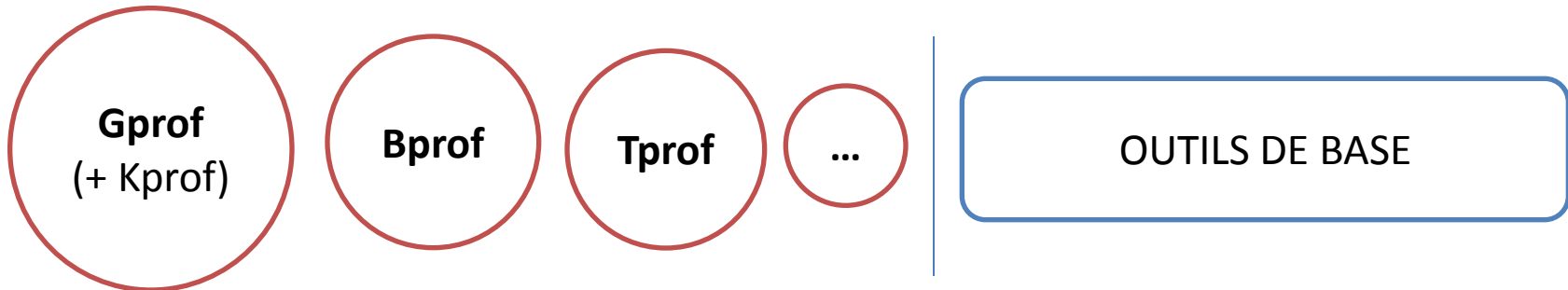
Step 1

Step 2

Step 3

Step 4

## ● Profilers



```
$ gcc -pg test.c -o testout
$ ./testout
$ gprof ./testout | less
```

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
100.00	0.05	0.05	1	50000.00	50000.00	calcul
0.00	0.05	0.00	1	0.00	0.00	affiche
0.00	0.05	0.00	1	0.00	50000.00	main

# Evaluer les performances

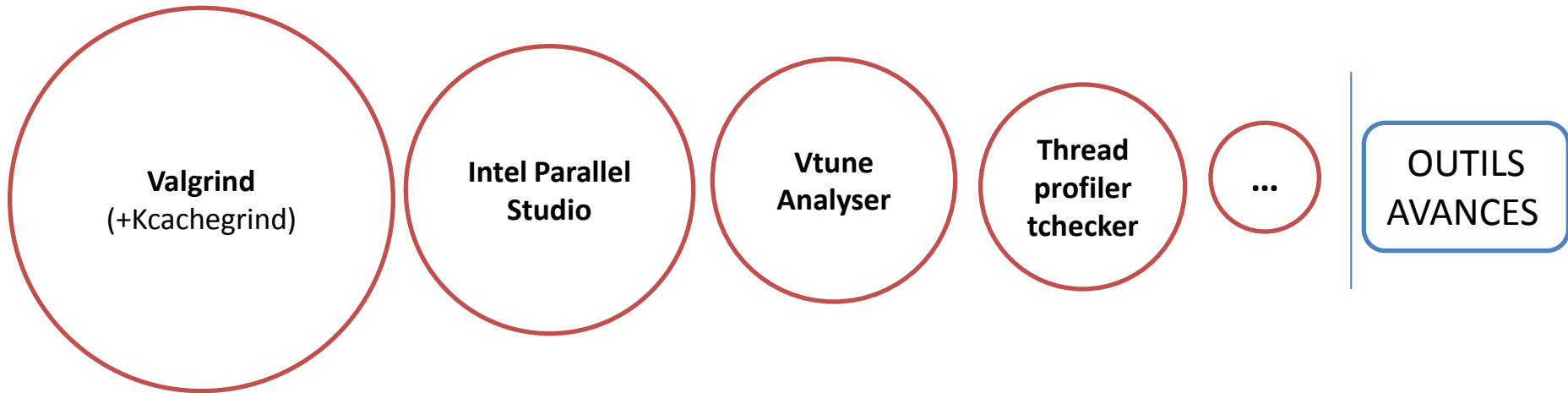
Step 1

Step 2

Step 3

Step 4

## ○ Profilers



# Optimiser

Step 1

Step 2

Step 3

Step 4

« L'optimisation prématurée est la source de tout les maux » Donald Knuth

- Niveau algorithmique (diminuer l'ordre de complexité , structure de données adaptée...)
- Niveau langage de développement (réorganisation du code, boucle, section critique, partage ...)
- Niveau assembleur (intégration instructions MMX, SSE4 ...)
- ...

Quelque chose  
pour vous ouvrir  
l'appétit ...



● **Questions?**

● **Idées?**

● **Applaudissement?**

Merci pour votre attention

**Prof. Mohamed AKIL & Ramzi MAHMOUDI**

**AMINA Workshop 2010**