Programmation en mémoire partagée (shared memory)

Informatique scientifique pour le Calcul
Ecoles Doctorales 2012/2013

Vincent Miele

CNRS - Biométrie & Biologie Evolutive

Février 2013

Document disponible sur `http://lyoncalcul.univ-lyon1.fr`

Plan

Parallélisme - Principe

Le **design** et l'**implementation** d'algorithmes parallèles répond à l'omniprésence des architectures multiprocesseurs/coeurs. Des questions impliquant un mélange d'**algorithmique** et d'**ingénierie**

- "comment faire faire UNE tâche par plusieurs travaileurs ?" **algorithmique**
- "si ils sont dans la même pièce ? dans des pièces différentes ?" **ingénierie**
- "les faire communiquer ? Se coordonner ?" **ingénierie**
- "qu'ils soient toujours occupés ?" **algorithmique**
- "qui fait quoi quand où ?" **algorithmique**

THIS IS PARALLEL COMPUTING
papier+machine
algorithmique+ingénierie

Parallélisme - Principe

Les principes du design d'algorithmes parallèles sont :
(d'après ▸ Introduction to parallel computing, Gramma et al, Addison Wesley)

1. decomposition into *tasks* (indivisible units executed in parallel) of various size

NB : ici, un process est une unité de calcul abstraite

Parallélisme - Principe

Les principes du design d'algorithmes parallèles sont :
(d'après ▸ Introduction to parallel computing, Gramma et al, Addison Wesley)

1. decomposition into *tasks* (indivisible units executed in parallel) of various size
2. listing the dependencies between tasks and evaluate their size

NB : ici, un process est une unité de calcul abstraite

Parallélisme - Principe

Les principes du design d'algorithmes parallèles sont :
(d'après ▸ Introduction to parallel computing, Gramma et al, Addison Wesley)

1. decomposition into *tasks* (indivisible units executed in parallel) of various size
2. listing the dependencies between tasks and evaluate their size
3. mapping tasks to *process* : scheduling/synchronizing to respect dependencies, avoid waiting time and minimize total time

NB : ici, un process est une unité de calcul abstraite

Parallélisme - Principe

Les principes du design d'algorithmes parallèles sont :
(d'après ▸ Introduction to parallel computing, Gramma et al, Addison Wesley)

1. decomposition into *tasks* (indivisible units executed in parallel) of various size
2. listing the dependencies between tasks and evaluate their size
3. mapping tasks to *process* : scheduling/synchronizing to respect dependencies, avoid waiting time and minimize total time
4. distributing the input/output

NB : ici, un process est une unité de calcul abstraite

Parallélisme - Principe

Les principes du design d'algorithmes parallèles sont :
(d'après ▸ Introduction to parallel computing, Gramma et al, Addison Wesley)

1. decomposition into *tasks* (indivisible units executed in parallel) of various size
2. listing the dependencies between tasks and evaluate their size
3. mapping tasks to *process* : scheduling/synchronizing to respect dependencies, avoid waiting time and minimize total time
4. distributing the input/output
5. managing access on common data

NB : ici, un process est une unité de calcul abstraite

Mémoire partagée

Sur une machine à *mémoire partagée*, toute la mémoire est accessible par tous les processes.
Les communications nécessaires à la synchronisation, à la gestion des tâches, des dépendences, etc... sont implicites.

Les paradigmes de programmation en mémoire partagée se focalisent sur :

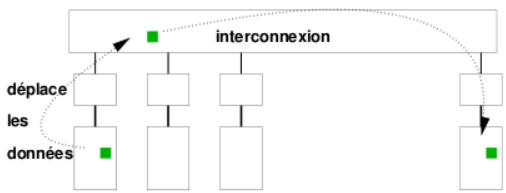
- le lancement de processes parallèles
- le partage des données (lecture et écriture)
- la synchronisation (l'ordre de réalisations des tâches,...)

Du parallélisme en mémoire partagée depuis le bas (`thread`) vers le medium (`openMP`, `Intel TBB`) vers le haut niveau (`Python multiprocessing`, `R multicore`)

Rappel

Mémoire Partagée

- Tous les processeurs accèdent à toute la mémoire globale avec un même espace d'adressage global.
- Chaque processeur travaille indépendamment des autres, mais les modifications effectuées par un processeur à un emplacement mémoire (en mémoire globale) sont visibles par tous les autres.
- Les processeurs ont leur propre mémoire locale (cache).



Mémoire Partagée

- UMA, Uniform Memory Access :
  - Des processeurs identiques, ayant tous le même temps d'accès à la mémoire. Plus couramment appelées SMP (Symmetric MultiProcessor).
  - Gestion de la cohérence des caches. Certains systèmes sont CC-NUMA : si un processeur modifie un emplacement mémoire partagé, la modification est visible par tous les autres processeurs.
- NUMA (Non Uniform Memory Access) :
  - Conçue pour pallier aux problèmes d'accès mémoire concurrents via un unique bus.
  - En général fabriquées à base de plusieurs blocs SMP interconnectés.
  - Des temps d'accès à la mémoire diffèrent suivant la zone accédée. Le temps d'accès via le lien d'interconnexion des blocs est plus lent.
  - Gestion de la cohérence des caches. CC-NUMA : Cache Coherent NUMA.

Plan

Idée : saupoudrer un programme de directives pour aider le compilateur à paralléliser.

Standard faisant suite à de nombreux dialectes “constructeurs”
Langages supportés : Fortran et C/C++
OpenMP 1.0 : 1997
OpenMP 2.5 : 2005
OpenMP 3.0 : 2008

Un jeu de directives de compilation confortable et standard, vu comme des commentaires par les compilateurs qui ne supportent pas openMP :
un programme contenant de l'openMP est aussi un programme séquentiel valide (portabilité !).

```
#pragma omp directive-name [clause[ [,] clause]...]
```

Juste compiler avec `-fopenmp` (supporté nativemment par la majorité des compilateurs actuels) et inclure le header

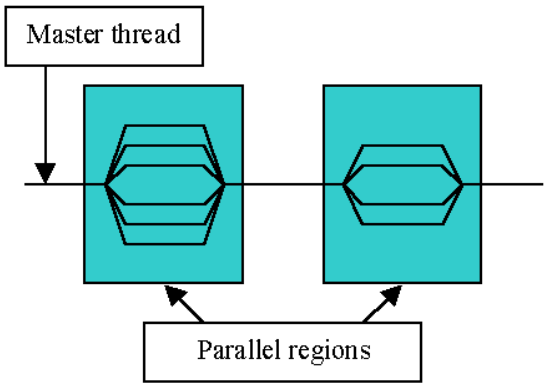
```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Il existe également des fonctions *run-time*, dont l'utilisation n'est pas possible par le code séquentiel (à utiliser avec `#ifdef _OPENMP`)

```
int/void omp_get/set_...()
```

└Execution parallèle

L'entité centrale d'un programme avec openMP n'est plus le process mais le *thread*. Le thread *master* (identifiant = 0) est immédiatement actif et dans des *régions parallèles* une équipe de threads execute les instructions en parallèle. On parle de modèle *fork-join*.



Pour sélectionner le nombre de threads :

```
export OMP_NUM_THREADS=10
```

```
#ifdef _OPENMP
omp_set_num_threads(10)
#endif;
```

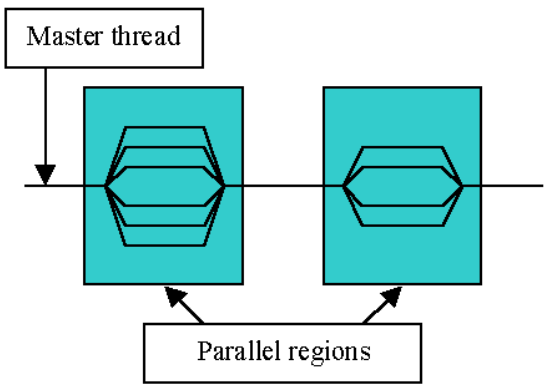
Le programmeur demande l'ouverture de régions parallèles avec la directive `parallel` :

```
#pragma omp parallel
{
    do_work_package(omp_get_thread_num(), omp_get_num_threads());
}
```

DEMO1

Execution parallèle

Le modèle *fork-join* conduit les performances à suivre la loi d'Amdawl : le speedup théorique maximum est donné par $\frac{1}{\alpha+(1-\alpha)/c}$, où α est la fraction du temps total concernant la partie séquentielle and c le nombre de processes (threads ici).



Si $\alpha = 10\%$ alors le speedup maximum est 10 même avec $c \to \infty$

Portée des données

Le partage du travail en parallèle implique de travailler sur des données communes/partagées ou privées à chaque thread.

Mécaniquement en C++, les données/variables déclarées en dehors des régions parallèles sont connues par tous les threads (`shared` par défaut), mais les données déclarées à l'intérieur sont privées (`private`) pour chaque thread.

```
int A[10], B[10];
#pragma omp parallel
{
    int bound = omp_get_thread_num()*5;  // bound est private
    for (int i=bound; i<bound+5; i++)
        A[i] = B[i]*B[i];
}
```

└Portée des données

Les clauses `private, shared, firstprivate,copyin` permettent de privatiser ou de copier vers des variables privées.

```
int i = 100;
#pragma omp parallel firstprivate(i)
{
    i = i + omp_get_thread_num(); // i = 100 or 101 or 102....
}
// here still i == 100
```

Attention, “concurrent write access to a shared variable is evil ! #1”

└Partage de travail des boucles

Dans beaucoup de codes de calcul, les boucles sont omniprésentes. Si les itérations sont indépendantes, elles sont candidates pour la parallélisation.
Les itérations sont distribuées entre les threads (ci-dessous, chaque thread a un jeu de valeurs `i`).
Le thread master reprend la main quand tous les threads ont finis leur itérations.

```
#pragma omp parallel
{
#pragma omp for
    for (int i = 1; i <10; i++)
        A[i] = B[i]*A[i];
}
```

```
#pragma omp parallel
{
#pragma omp for
    for (int i = 1; i <10; i++)
        A[i] = B[i]*A[i-1];  // CATASTROPHE
}
```

Marche également depuis la version 3.0 avec certains itérateurs

```
#pragma omp parallel for
{
    for (it = v.begin(); v != v.end(); v++)
}
```

NB1 : on peut combiner les 2 directives (cf. ci-dessus)
NB2 : si une boucle parallèle est rencontrée dans une fonction appelée dans une région parallèle, elle sera effective.

DEMO2

└Partage de travail des boucles

Attention à la *race condition*.
Le programmeur ne peut maîtriser la façon dont une variable `shared` est utilisée en écriture.

```
int sum = 0
#pragma omp parallel
{
#pragma omp for
    for (int i = 0; i <1000000; i++)
        sum += i;  // CATASTROPHE
}
```

DEMO3

Et `firstprivate(sum)` ne résout pas le problème...

“Concurrent write access to a shared variable is evil ! #2“
Les regions critiques résolvent le problème : une thread après l'autre execute une partie du code (ordre inconnu).
Mais attention : les threads risquent d'attendre leur tour... ! !

```
int sum = 0, tsum = 0;
#pragma omp parallel firstprivate(tsum)
{
#pragma omp for
    for (int i = ; i <10; i++)
        tsum += i;
}
#pragma omp critical
{
    sum += tsum
}
}
```

DEMO4

└Synchronisation

Pour éviter les *deadlocks*, on donne des noms aux sections critiques

```
#pragma omp critical (sum_critical)
{
    sum += func(i)
}
....
void func(){....
#pragma omp critical (random_critical)
    {
        result = random_func()
        // utilise une graine/seed partagée
    }
```

(sans les noms, une thread peut s'attendre elle-même. . .).

Il peut être nécessaire de synchronizer les threads (attention au *load imbalance*...)

```
#pragma omp barrier
```

└Réduction

A la fin d'une boucle, les contributions de chaque thread sont accumulées : on parle alors d'opération de `reduction`

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

Les opérateurs sont +, *, -, /, &, ^, |, &&, ||

└Ordonnancement de boucle

Le *mapping* des itérations de boucle est configurable :

- le plus simple, diviser la boucle en *chunks* contigus de taille donnée :
  `#pragma omp parallel for schedule(static)`
  Si la taille de la tâche par itération est variable, sous-optimal.

- des chunks sont assignés dynamiquement à chaque thread disponible
  `#pragma omp parallel for schedule(dynamic)`
  Pour un taille de chunk petite (1), risque d'*overhead* (accés concurrent à la liste des itérations à réaliser).
  Pour une grande taille, risque de *load imbalance*.

└Ordonnancement de boucle

- il est aussi possible depuis la version 3.0 que le thread master genère des tâches qui seront dynamicament executées

```
Family fam[100];
#pragma omp parallel
{
#pragma omp single
{
    for (i=0; i<100; i++)
        if (fam[i].size() > 3)
#pragma omp task
{
            do_something(fam[i])
}
}
}
```

Plan

"thread-safe"

Les entrées/sorties ne sont pas *thread-safe* : on utilise des régions `critical` mais attention à l'overhead.

DEMO1

Certaines fonctions de la librarie standard C++ ne sont pas thread-safe (exple : rand).

Placement des régions paralléles

Optimiser le nombre de 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 *scaling* sauf si 10 est multiple du nombre de threads.

```
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] =....
```

Overhead.

Utiliser le code séquentielle quand le parallélisme ne peut être efficace car l'ouverture des régions parallèles coûte plus cher que le calcul (à benchmarker).

```
#pragma omp parallel for if (n>1000)
    for (i=0; i<n; i++)
        a[i] =....
```

"false sharing"

Le *false sharing* se produit quand plusieurs threads agisse sur la même ligne de cache, créant ainsi un fort ralentissement du programme dû aux mécanisme de cohérence de cache.

```
vector<int> v = .... (dans {0,..,9})
int count[nbthreads][10]
#pragma omp parallel for
for (unsigned int i=0; i<n; i++)
    count[numthread][v[i]]++;
```

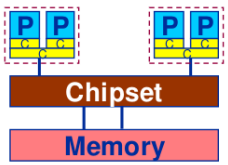
Utiliser des variables privées pour remédier au problème.

```
#pragma omp parallel
{
int count_t[10]
...
#pragma omp critical
{
...
}
}
```

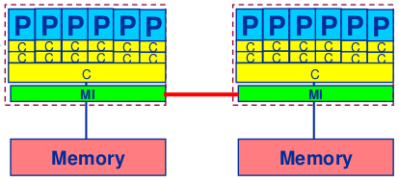
NB : L'accès en lecture ne crée pas du false sharing.

Effet ccNUMA

"EVEN Concurrent READ access to a shared variable MAY BE evil !"

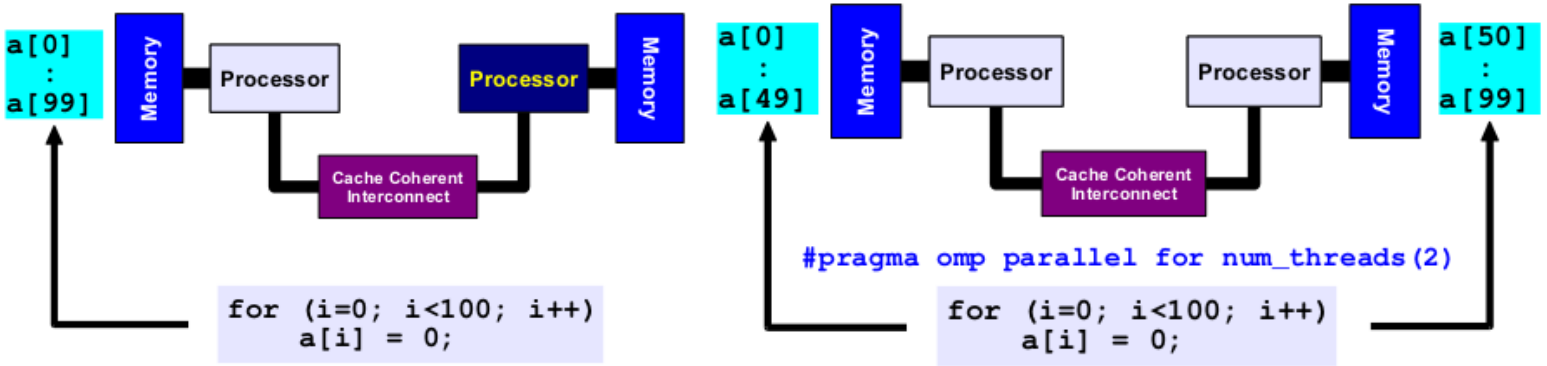


SMP OK (quad-core),



NUMA pas OK (bi hexa-core)

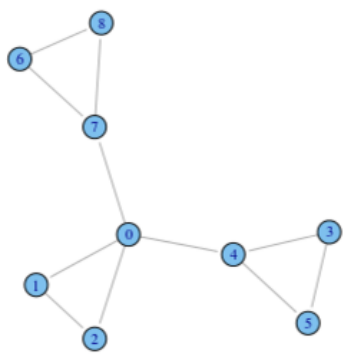
Solutions :
-placement de threads
-*first touch policy*

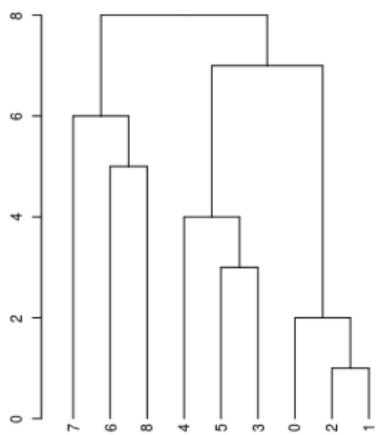


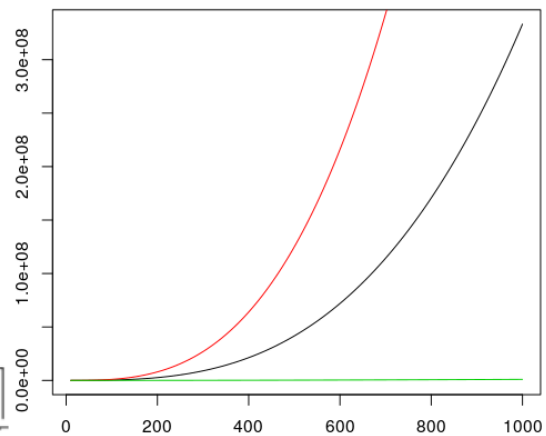
Classification ascendante hiérarchique (cf. précédent cours)

n entités, et on sait calculer la distance entre 2 groupes d'entités (ou singletons)

```
n(n-1)/2 calculs de distance
puis
k <- 0
tantque k<n-2
     (n-k)(n-k-1)/2 recherche de min -> (i,j,min)
     (n-k-1) recalculs de distance après fusion(i,j)
```

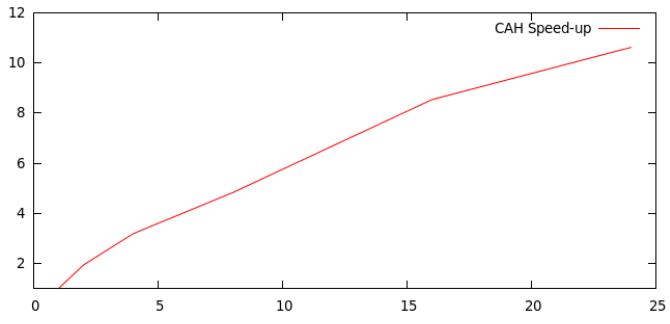






CAH $n = 8835$

```
#pragma omp parallel if (nbD>1000)
    { // page placement by first touch
#pragma omp for
      for(int ii=0; ii<nbD; ++ii)
        {
          *(_D[0]+ii) = 0.;
        }
    }
```




```
#pragma omp parallel if (nbD>1000) shared(dmin_shared, iimin_shared)
 private(iimin) firstprivate(dmin_p)
    {
#pragma omp for
      for(int ii=0; ii<nbD; ++ii)
        {
          double distij = *(_D[0]+ii);
          if(distij<=dmin_p){
            dmin_p = distij;
            iimin = ii;
          }
        }
#pragma omp critical
      {
        if(dmin_p<=dmin_shared){
          dmin_shared = dmin_p;
          iimin_shared = iimin;
        }
      }
    }
    dmin = dmin_shared;
```

Avec 24 coeurs sur un Quadri 12-core AMD 2.2 Ghz :
- 45 secondes avec first touch policy
- 157 secondes sans first touch policy

Gloups

oepnMP... pas si simple : difficile d'obtenir des performances sans effort

- problème de l'architecture des multicoeurs
- problème du coûts des actions internes à openMP relativement aux coûts des calculs
- difficultés de l'accés concurrent aux données
- ordonnancement des tâches non résolu

Se documenter

- cours de l'IDRIS
- formation du groupe Calcul
- reference sheet for C/C++
- 32 OpenMP Traps For C++ Developers

Plan

La bibliothèque Intel Threading Building Blocks : orienté-objet, orienté-STL orienté-tâche
La librairie thread du `C++11` : moderne et natif mais bas-niveau et technique

Le module multiprocessing : haut-niveau, efficace pour lancer des tâches en parallèle mises en file d'attente (différentes possibilités de scheduling : (un)ordered,etc..)

DEMO5

Le package multicore intégré dans le package `parallel` dans R-base permet de faire des boucles parallèles

```
for (i in 1:30){ rnorm(i)} # boucle explicite classique, à éviter

lapply(1:30, rnorm) # boucle implicite

library(parallel)
mclapply(1:30, rnorm) # boucle implicite parallèle
```

Utilise le mécanisme *copy-on-write* très efficace.

Plan

On en reparle...

Calcul flottant, salle C4 Nautibus, par Florent de Dinechin (LIP)