TD/TP5 : Analyse de performances de configurations de microprocesseurs multicoeurs pour des applications parallèles.

Le parallélisme au niveau instructions *(ILP : Instruction Level Parallelism)* est un niveau de parallélisme qui a ses limites.

Une autre approche consiste à utiliser des architectures capables d'exécuter plusieurs flots d'instructions indépendants simultanément sur plusieurs flots de données, en fonction de la disponibilité des ressources internes. Citons par exemple les architectures superscalaires de type SMT (Simultaneous Multithreading). Ces architectures exploitent le parallélisme de l'application au niveau des taches *(TLP : Thread Level Parallelism).* Le Pentium4 HyperThreading est un exemple commercial de ce type de processeur [1].

Pourtant, la recherche de performances supplémentaires sur la base d'un processeur unique ne peut plus être poursuivie pour des raisons thermiques et de consommation d'énergie. Cela signifie la fin de l'augmentation des fréquences d'horloge.

L'alternative consiste à multiplier le nombre de processeurs sur une même puce en utilisant une fréquence d'horloge plus basse. La Figure 16 montre les architectures de processeurs (a) monoprocesseur scalaire in-order (b) monoprocesseur superscalaire out-of-order (c) monoprocesseur SMT (Simultaneous Multithreading) (d) CMP (Chip MultiProcessor/multicore).

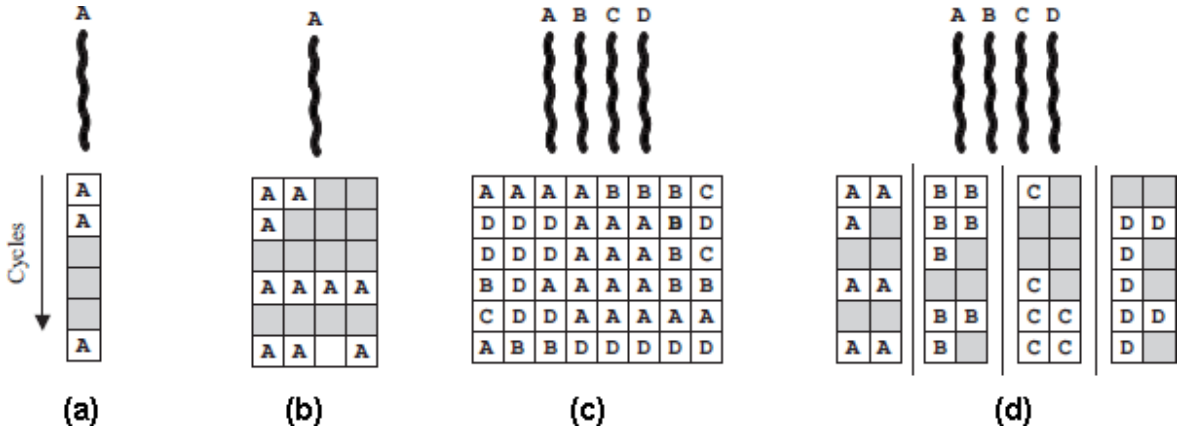

Figure 16 a) scalaire b) superscalaire c) SMT Simultaneous Multithreading d) CMP Chip MultiProcessor

IBM POWER6 [2], SUN UltraSPARC T2 [3], et Intel Ivy Bridge [4] sont des exemples commerciaux parmi plusieurs qui existent sur le marché.

Par contre, en multipliant le nombre de processeurs, des nouvelles questions et de nouveaux problèmes se posent :

- Type des cœurs de processeur (scalaire, superscalaire)
- Hiérarchie mémoire
- Synchronisation et communication entre les cœurs de processeur
- Cohérence de cache

Le but de ce TD/TP est d'explorer les architectures multiprocesseurs CMP en exécutant une application parallèle. En particulier, les performances par rapport aux nombre de cœurs et le type de chaque cœur seront étudiées. Pour ce TD/TP, on va utiliser le simulateur gem5 (Annexes 2 et 3) pour générer un simulateur de type SMP (Symmetrical MultiProcessing) afin de simuler des architectures multiprocesseurs de type CMP.

Description de l'application multiplication de matrice parallèle

Pour ce TD/TP, on va utiliser l'application de multiplication de 2 matrices A et B, qui est parallélisée pour être exécutée par **m** threads. A et B sont 2 matrices de taille **n * n**, et le résultat est stocké dans une matrice C.

La Figure 18 montre une description de cet algorithme pour 2 threads et une matrice de taille **4 * 4**

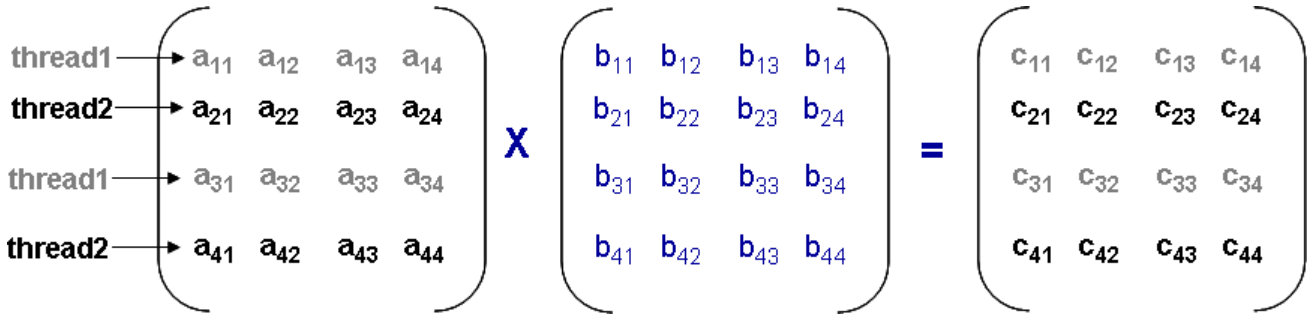

Figure 17 Multiplication de matrice 4x4 avec 2 threads

Le thread principal créé ('fork') **m-1** threads secondaires indépendants. Ce pool de m threads exécute alors le calcul matriciel comme visible sur la Figure 19.

Puis, le thread principal attend la fin de l'exécution de tous les threads secondaires et considère que la multiplication des matrices a réussi.

Cette génération de threads secondaires est gérée dans la boucle **for** principale du programme **test_omp.cpp** grâce à une librairie de programmation parallèle très utilisée dans la communauté scientifique, OpenMP [6].

Dans le programme **test_omp.cpp** (compilé vers l'exécutable **test_omp**), le nombre de threads **m** ainsi que la taille des matrices **n** sont paramétrables.

Les paramètres d'entrée de l'application sont donc :

./test_omp *<nthreads> <size>*

<nthreads> : nombre de threads en parallèles (de 1 à m<n)

<size > : nombre de lignes et colonnes de la matrice carrée (**Note :** *pour conserver des temps de simulations raisonnables, garder n<256)*

Pour récupérer le binaire **test_omp** :

$cp –v /home/c/**nom-mdc**/ES201/tools/TP5/test_omp /your/target/directory/

Attention : La librairie de programmation pthreads fournie pour votre TD/TP pour le mode « SE » de gem5 (**voir Annexes 2 et 3**) est toujours en cours de développement et impose pour le moment de conserver l'équivalence suivante : **n threads = n cœurs**.

Attention 2 : Se référer au fichier : /home/c/**nom-mdc**/ES201/tools/TP5/README expliquant la procédure (ainsi qu'aux annexes).

export GEM5=/home/c/nom-mdc/ES201/tools/TP5/gem5-stable

L'exécution du programme **test_omp** se fera donc de la manière suivante (avec ncores = nthreads) :

$GEM5/build/ARM/gem5.fast $GEM5/configs/example/se.py -n *<ncores>* **-c test_omp -o "***<nthreads> <size >***"**

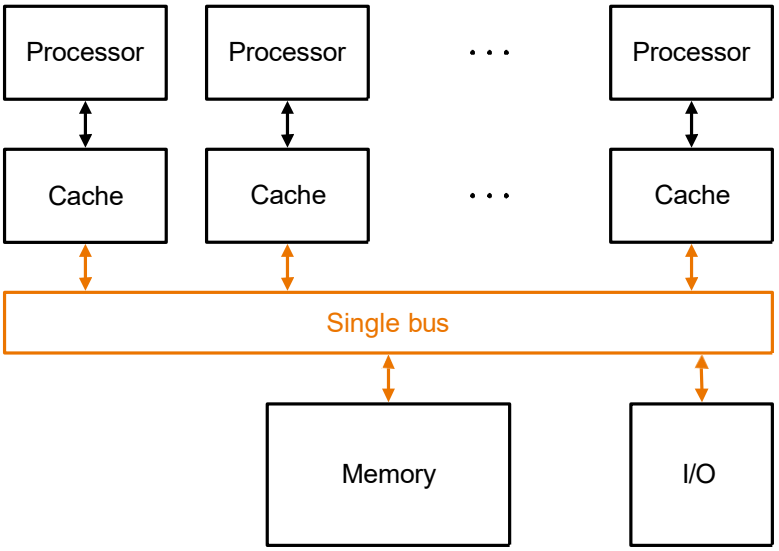

Figure 18 Architecture multicœurs à base de bus

Note : Pensez à vous référer aux Annexes 2 et 3 ainsi qu'à l'aide de l'exécutable pour accéder à toutes les options d'exécution possibles (type de cpu, configuration des caches, etc.) :

$GEM5/build/ARM/gem5.opt $GEM5/configs/example/se.py --help

Questions

1. **Analyse théorique de cohérence de cache**

Q1 : En considérant que chaque thread s'exécute sur un processeur dans une architecture de type multicoeurs à base de bus et 1 niveau de cache (comme décrit Figure 21), décrivez le comportement de la hiérarchie mémoire et de la cohérence des caches pour l'algorithme de multiplication de matrices. On supposera que le thread principal se trouve sur le processeur d'indice 1.

2. **Paramètres de l'architecture multicoeurs**

Le simulateur gem5 permet de faire varier de très nombreux paramètres des éléments architecturaux (types de processeur et paramètres associés à un processeur donné, mémoires caches L1, L2, cohérence de cache, etc.). Tous ces paramètres ne sont pas « accessibles » depuis les options de configuration système **se.py**, mais il est pourtant possible de les faire varier.

Q2 : Examinez le fichier de déclaration d'un élément de type « processeur superscalaire out-of-order », et présentez sous forme de tableau cinq paramètres configurables de ce type de processeur avec leur valeur par défaut. Choisissez de préférence des paramètres étudiés lors des séances TD/TP précédentes. Le fichier à consulter est le suivant :

$GEM5/src/cpu/o3/O3CPU.py

Q3 : Examinez le fichier d'options de la plateforme **se.py,** puis déterminez et présentez sous forme de tableau les valeurs par défaut des paramètres suivants :

- Cache de données de niveau 1 : associativité, taille du cache, taille de la ligne
- Cache d'instructions de niveau 1 : associativité, taille du cache, taille de la ligne
- Cache unifié de niveau 2 : associativité, taille du cache, taille de la ligne

Le fichier d'options à consulter est le suivant :

$GEM5/configs/common/Options.py

3. **Architecture multicoeurs avec des processeurs superscalaires in-order (Cortex A7)**

On se propose dans cette partie d'étudier une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A7 étudié dans le TD/TP5. Dans notre simulateur gem5, le modèle de CPU associé à ce type de processeur est le modèle **arm_detailed (--cpu-type=arm_detailed)**.

En fixant la taille de la matrice à **m**, et en faisant varier le nombre de threads parallèles de l'application (nombre de threads = 1, 2, 4, 8, 16, …, **m**), répondre aux questions suivantes :

Q4 : Déterminez quel est le processeur exécutant toujours le plus grand nombre de cycles. Expliquez pourquoi. Expliquez également pourquoi l'analyse du nombre de cycles sur ce processeur revient à analyser le nombre total de cycles d'exécution de l'application.

Q5 : Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 2 axes

Q6 : Déduire le speedup par rapport à la configuration à 1 thread.

Q7 : En utilisant le nombre total d'instructions simulées, déterminez quelle est la valeur maximale de l'IPC pour chaque configuration ?

Q8 : Discussion et interprétation *(max. 10 lignes)*.

4. **Architecture multicoeurs avec des processeurs superscalaires out-of-order (Cortex A15)**

On se propose dans cette partie d'étudier une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A15 étudié dans le TD/TP5. Dans notre simulateur gem5, on dispose d'un modèle de processeur superscalaire out-of-order : le modèle **o3** (--cpu-type=detailed)

En fixant la taille de la matrice à **m**, et en faisant varier le nombre de threads parallèles de l'application (nombre de threads = 1, 2, 4, 8, 16, …, **m**), et la largeur du processeur superscalaire (nombre de voies = 2, 4, 8), répondre aux questions suivantes :

Q9 : Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 3 axes.

Q10 : Déduire le speedup par rapport à la configuration à 1 thread.

Q11 : En utilisant le nombre total d'instructions simulées, déterminez quelle est la valeur maximale de l'IPC pour chaque configuration ?

Q12 : Discussion et interprétation *(max. 10 lignes)*.

5. **Configuration CMP la plus efficace**

Q13 : Proposez une configuration ou une gamme de configuration de l'architecture CMP (nombre de threads de l'application **test_omp**, nombre et type de cœurs) qui vous semble la plus appropriée si la contrainte recherchée par le concepteur du système est l'efficacité surfacique ? Discussion et interprétation *(max. 10 lignes)*.
N.B. : *Vous vous appuierez sur les résultats des deux TD/TP (4 et 5).*

6. **Facultatif :**

En utilisant des tailles de matrices relativement grandes (taille des lignes et des colonnes en octets supérieure à la taille du cache de données de niveau 1) ou des tailles de caches de données relativement petites, un grand nombre de threads peut parfois faire apparaître un comportement « supra-linéaire » (comportement pour lequel le speedup est supérieur à n, où n est le nombre de threads).

Q14 : Au regard de l'évolution théorique du speedup et son évolution constatée lors des questions précédentes, proposez une tentative d'explication *(max. 10 lignes)*.

Remettre un rapport de TP par **quadrinôme** en version électronique **au format PDF** (**TP5-nom1-nom2-nom3-nom4.pdf**) incluant l'ensemble des réponses aux questions précédentes pour le **24/03/2025** aux adresses emails : omar.hammami@ensta.fr et votre chargé de TD avec en sujet de message **ECE_4ES01_TA /TP5**.

La note du rapport comptera pour 15% de la note globale.

N.B. : *La lisibilité et le format font partie de la note de votre rapport.*

```
// test_omp.cpp
// Good old matrix multiply using openmp

#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int64_t* A;
int64_t* B;
int64_t* C;

int main(int argc, const char** argv) {

  if (argc != 3) {
    printf("Usage: ./test_omp <nthreads> <size>\n");
    exit(1);
  }

  int nthreads = atoi(argv[1]);

  if (nthreads < 1) {
    printf("nthreads must be 1 or more\n");
    exit(1);
  }

  int size = atoi(argv[2]);

  if (size < 1) {
    printf("size must be 1 or more\n");
    exit(1);
  }

  printf("Setting OMP threads to %d\n", nthreads);
  omp_set_num_threads(nthreads);

  A = (int64_t*) calloc(size*size, sizeof(int64_t));
  B = (int64_t*) calloc(size*size, sizeof(int64_t));
  C = (int64_t*) calloc(size*size, sizeof(int64_t));

  printf("Starting with row/col size=%d\n",size);

  for (int x = 0; x < size; x++) {
    for (int y = 0; y < size; y++) {
      A[x*size + y] = x*y;
    }
  }
  printf("A initialized\n");

  for (int x = 0; x < size; x++) {
    for (int y = 0; y < size; y++) {
      B[x*size + y] = x*y - y;
    }
  }
  printf("B initialized\n");
  printf("Computing A*B with %d threads\n", nthreads);

  #pragma omp parallel for
  for (int x = 0; x < size; x++) {
    for (int y = 0; y < size; y++) {
      int64_t tot;
      for (int m = 0; m < size; m++) {
        tot += A[x*size + m]*B[m*size + y];
      }
      C[x*size + y] = tot;
    }
  }

  printf("Done\n");
 return 0;
}
```