Université Sidi Mohamed Ben Abdellah

Faculté des sciences Dhar El Mahrez Fès

Master Big Data Analytics & Smart Systems

Traitement Parallèle

TP7 : Open MP

Introduction à l'utilisation d'Open MP (fait)..... (de 3 a 6)

Calcul de produit des matrices (fait)..... (de 7 a 10)

Réalisé par Grp 6 :

Mohamed EL BAGHDADI

Ayoub BERRAG

Encadré par :

Pr. Mohamed MEKNASSI

SOMMAIRE

Open MP

Définition

Open MP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est prise en charge par de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

La programmation parallèle hybride peut être réalisée par exemple en utilisant à la fois OpenMP et MPI.

Le développement de la spécification OpenMP est géré par le consortium OpenMP Architecture Review Board.

Les threads avec la bibliothèque pthreads

La bibliothèque standard pthread.h fournit de nombreuses méthodes pour créer, manipuler, détruire, ... des threads dans un programme. La bibliothèque est plutôt orientée bas-niveau, la majorité des tâches de gestion ou de synchronisation est laissée à l'utilisateur. Développer une application parallèle avec cette bibliothèque demande une certaine maîtrise et beaucoup de débogages...

La bibliothèque intel threading building blocks

Intel distribue (sous 2 versions de licence, GPL ou commerciale) une bibliothèque pour le calcul parallèle : Intel Threading Building Blocks. Cette bibliothèque écrite pour le C++ utilise les templates, compliquant un peu la programmation de tâches simples.

La bibliothèque openmp

OpenMP est une bibliothèque supportée par plusieurs langages (C,C++ et Fortran) disponible sur plusieurs plate-formes (Linux, Windows, OS X, ...). OpenMP regroupe des directives de compilation et des fonctions. Le compilateur gcc supporte OpenMP depuis la version 4.2, en ajoutant simplement une option sur la ligne de commande et en

incluant le fichier d'en-têtes omp.h. La gestion des différentes fonctions est assurée par la libraire libgomp.

Dans la suite de cet article, nous présenterons différents exemples pour découvrir OpenMP. Seule la parallélisation des boucles sera abordée dans cet article. OpenMP permet d'accélérer les calculs sans devoir gérer les threads « à la main ». Il est bien sûr possible de gérer finement les threads avec OpenMP.

Utilisation de la bibliothèque openmp

Pour commencer, un programme élémentaire va être pris comme exemple pour montrer l'utilisation générale d'OpenMP. Ce programme est présenté ci-dessous, la boucle représente le traitement de différents éléments de manière séquentielle.

```
#include<stdio.h>

#include<stdlib.h>

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

  int n;

  for(n=0;n<8;n++){

    printf("Element %d traité\n",n);

  }

  return EXIT_SUCCESS;

}
```

La compilation se fait avec la ligne :

```
gcc -Wall -oPremierProgramme PremierProgramme.c
```

Rien de bien surprenant ne se produit à l'exécution (normalement...). Les éléments sont traités de manière séquentielle, l'un après l'autre.

Nous allons maintenant modifier ce programme pour que le traitement des différents éléments se fasse de manière simultanée. La machine utilisée pour tester ce programme est un Core 2 Duo. Elle est donc capable de traiter deux tâches de manière simultanée. Le programme modifié est le suivant

```
#include<stdio.h>
```

```
#include<stdlib.h>

#include<omp.h>

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

  int n;

  #pragma omp parallel for

  for(n=0;n<8;n++){

    printf("Element %d traité par le thread %d \n",n,omp_get_thread_num());

  }

  return EXIT_SUCCESS;

}
```

Les modifications apportées sont les suivantes :

- Inclusion de la librairie omp.h qui contient différentes fonctions dont omp_get_thread_num qui permet de connaître le numéro du thread actuel.

- Ajout d'une directive de compilation omp parallel for avant la boucle. OpenMP parallélisera automatiquement la boucle qui suit la directive.

Le programme est compilé avec gcc en ajoutant l'option -fopenmp, la ligne de compilation est :

```
gcc -Wall -fopenmp -oPremierProgramme PremierProgramme.c
```

L'exécution du programme donne l'affichage suivant (il peut varier selon votre système) :

```
Element 0 traité par le thread 0
Element 4 traité par le thread 1
Element 1 traité par le thread 0
Element 5 traité par le thread 1
Element 2 traité par le thread 0
Element 6 traité par le thread 1
Element 3 traité par le thread 0
Element 7 traité par le thread 1
```

Deux threads ont été automatiquement créés. L'ensemble des indices de la boucle a été séparé en deux parties : les indices allant de 0 à 3 sont traités par le premier thread, les indices allant de 4 à 7 sont traités par le second thread.

Fixer le nombre de threads

Dans l'exemple précédent, le nombre de threads a été automatiquement fixé par OpenMP. Il est possible de fixer le nombre de threads de plusieurs manières. Il peut être configuré en utilisant la variable système OMP_NUM_THREADS

```
$ setenv OMP_NUM_THREADS 8
```

La bibliothèque propose plusieurs fonctions pour gérer les threads. La fonction omp_set_num_threads permet de fixer le nombre de threads dans le programme. D'autres fonctions sont utiles comme :

- omp_get_num_procs qui permet de connaître le nombre de processeurs équipant la machine.

- omp_get_num_threads qui retourne le nombre de threads actuellement gérés par la librairie.

Il est ainsi possible d'adapter le nombre de threads aux processeurs équipant la machine.

On peut modifier le nombre de threads à l'intérieur de la directive parallel en utilisant la clause num_threads(nb). Pour tester le programme avec 4 threads, on modifie la directive de la manière suivante : #pragma omp parallel for num_threads(4). Le nombre de threads peut être une constante ou une variable du programme.

CALCUL DU PRODUIT DE MATRICES

Le calcul du produit de deux matrices est une application typique (mais un peu matheuse...) des problèmes de parallélisation. Chaque élément peut être calculé de manière indépendante des autres, la tâche doit donc pouvoir se traiter avec OpenMP. Le calcul du produit de matrices fait appel à trois boucles les unes dans les autres (pour l'algorithme naïf, des versions optimisées existent).

Partage des variables entre les threads

Avant de présenter le détail du programme, nous allons nous attarder sur une option de la directive omp parallel for utilisée dans le programme précédent.

Par défaut, toutes les variables présentes dans la boucle sont partagées entre les différents threads, sauf le compteur de boucles (chaque thread dispose d'une copie qu'il est libre de modifier, sans conséquence pour les autres threads). Il est souvent nécessaire de protéger des variables contre des accès simultanés (un thread écrit pendant qu'un autre lit). Les variables à protéger sont listées dans la clause private.

Exemple de programme réalisant le produit de matrices avec openmp

Ces différentes notions sont utilisées dans le programme de calcul du produit de matrices. Le programme est présenté ci-dessous. Il permet de mesurer les performances d'OpenMP avec un nombre de threads variable (de 1 à 12). Le temps de calcul est mesuré avec la fonction gettimeofday.

```
#include <stdio.h>

#include <stdlib.h>
```

```
#include <omp.h>

#include <sys/time.h>

#define SIZE 2000

double get_time() {

  struct timeval tv;

  gettimeofday(&tv, (void *)0);

  return (double) tv.tv_sec + tv.tv_usec*1e-6;

}

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

  int nb, i , j, k;

  double t,start,stop;

double* matrice_A;

  double* matrice_B;

  double* matrice_res;

  // Allocations

  matrice_A = (double*) malloc(SIZE*SIZE*sizeof(double)) ;

  matrice_B = (double*) malloc(SIZE*SIZE*sizeof(double)) ;

  matrice_res = (double*) malloc(SIZE*SIZE*sizeof(double)) ;

  for(i = 0; i < SIZE; i++){

    for(j = 0; j < SIZE; j++){

      matrice_A[i*SIZE + j] = (double)rand()/(double)RAND_MAX;

      matrice_B[i*SIZE + j] = (double)rand()/(double)RAND_MAX;
```

```
   }
  }
  printf("Nb.threads\tTps.\n");
  for(nb=1;nb<=12;nb++){
   start = get_time();
   #pragma omp parallel for num_threads(nb) private(j,k)
   for(i = 0; i < SIZE; i++){
    for(j = 0; j < SIZE; j++){
     matrice_res[i*SIZE + j] = 0.0;
     for(k = 0; k < SIZE; k++){
      matrice_res[i*SIZE + j] += (matrice_A[i*SIZE + k]*matrice_B[k*SIZE + j])
     }
    }
   }
   stop=get_time();
   t=stop-start;
printf("%d\t%f\n",nb,t);
  }
  // Libérations
  free(matrice_A);
  free(matrice_B);
  free(matrice_res);
```

```
return EXIT_SUCCESS;}
```

Résultats

Pour tester les différents programmes qui vont suivre, nous avons utilisé une machine équipée de deux processeurs Xeon 5060. Chaque processeur est un double cœur capable de gérer l'hyperthreading. En théorie (ou selon les commerciaux...), cette machine peut donc exécuter 8 threads simultanément.

Le système d'exploitation est Ubuntu Server 8.10 avec un compilateur gcc dans sa version 4.3. Pour tous les essais, le système est utilisé sans interface graphique pour minimiser le nombre de tâches liées au système d'exploitation.