Projet 4MOC

Février 2023

Open MP

Module associé : **Programmation Parallèle**

Enseignant : **Thierry JOUBERT**

Un rendu par groupe en soutenance

Documents à fournir : **Fichier Zip contenant tous les sources ainsi q'un makefile ou script de compilation. En plus du code conserver (mais ne pas envoyer) les fichiers BMP des différents tests pour les présenter et les commenter en soutenance.**

Pré-requis :

- Programmation en C/C++
- Librairie *Open MP*

Librairie complémentaire :

- EasyBmp

Objectif :

Analyser la façon dont la librairie *OpenMP* répartit les calculs entre différents threads en utilisant un marqueur de couleur pour chaque thread dans un fichier d'image que l'on visualisera après exécution.

SOMMAIRE

Introduction

On se propose de représenter la contribution de chaque thread d'une application OMP par des zones de couleur dans un fichier BMP. On utilisera pour cela le code C++ open source de la librairie **EasyBMP** avec une application **OmpTrace** compilée avec OpenMP.

L'application crée une image BMP vide puis colorie tous les pixels avec deux boucles imbriquées LIGNES – COLONNES. La couleur de chaque pixel sera basé sur le numéro du thread OMP qui colorie, obtenu avec fonction omp_get_num_thread(). A la suite des boucles, l'image est sauvée dans un fichier passé en argument du programme (**`argv[1]`**) un second argument (**`argv[2]`**) servira à indiquer le nombre de threads OMP avec la fonction omp_set_num_threads().

On part du code mono-thread suivant (fourni dans OmpTrace.cpp)

```
// OmpTrace.cpp : painting OMP threads
// compilation:
// g++ -c EasyBMP.cpp
// g++ -c OmpTrace.cpp -fopenmp
// g++ OmpTrace.o EasyBmp.o -fopenmp
//
#include "EasyBMP.h"
#include "omp.h"
#define  HAUTEUR   1080
#define  LARGEUR   1920

int main(int argc, char* argv[])
{
  RGBApixel pixel;
  BMP img ;
#ifdef _OPENMP
    printf("*** Compiled with OPENMP ***\n");
#endif
  if (argc < 3)
      goto finally;
  img.SetSize(LARGEUR, HAUTEUR);
  for (int m = 0; m < LARGEUR; m++)  // Thread1 Thread2 etc.
  {
      for (int n = 0; n < HAUTEUR; n++)
      {
            pixel.Blue = 192;
            pixel.Green = 255 ;
            pixel.Red = 192;

            img.SetPixel(m, n, pixel);
      } // columns
  } // lines
  img.WriteToFile(argv[1]);

  printf( "...done\n");
  return 0;

finally:
  printf("syntaxe: %s  fichier_bmp  nombre_de_threads\n",argv[0]);
  printf("     ex: %s  trace.bmp 4\n",argv[0]);
  return -1;
}
```

Travaux à réaliser

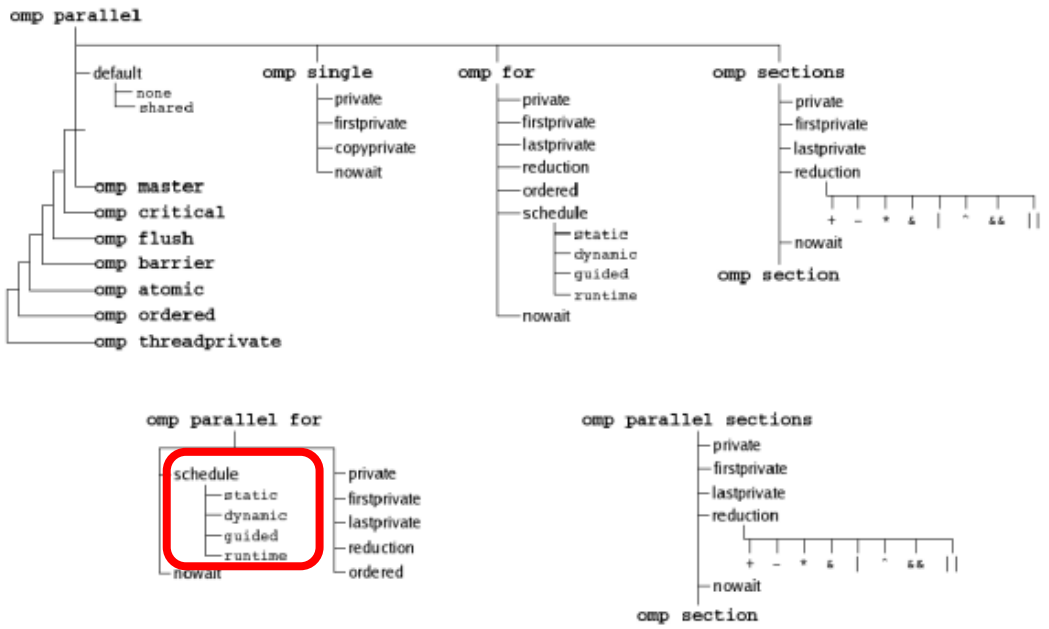
Gestion du parallélisme de base

Ajouter au code fourni des **`#pragma omp parallel for`** pour que la boucle de coloration se fasse en multi-threads. Modifier aussi l'alorithme de coloration pour y intégrer le numéro du thread dans une des composantes (R, G, ou B) en utilisant le résultat de omp_get_num_thread() on peut faire par exemple un `switch(numThread)` sur les 8 couleurs de base `0-Red, 1-Green, 2-Blue, 3-Yell., 4-Mag., 5-Cyan, 6-White, 7-Black.`

L'image produite en multi-thread ne devrait plus être mono-chrome, modifier le nombre de threads pour bien constater la répartition des threads OMP, sauver à chaque fois le fichier BMP comme trace de votre travail (accessoirement le convertir en JPG pour réduire sa taille).

Tests du Scheduler OMP FOR

La syntaxe OMP permet de contrôler comment la répartition du travail entre les threads se fait dans un #pragma omp parallel for. Avec la clause schedule on peut appliquer différentes stratégies (static, dynamic, etc.)



Mettre en œuvre ces quatre stratégies dans le code de **`OmpTrace.cpp`** et visualiser l'effet dans le fichier image. Sauvegarder ces fichiers comme trace de votre travail, cela donnera 5 fichiers pour un nombre de threads donné (ex. 8)

1. Pas de clause schedule
2. Schedule(static)
3. Schedule(dynamic)
4. Schedule(guided)
5. Schedule(runtime)