# Laboratori 1 : Open MP

Autors: Joan Travé i Johnny Núñez.

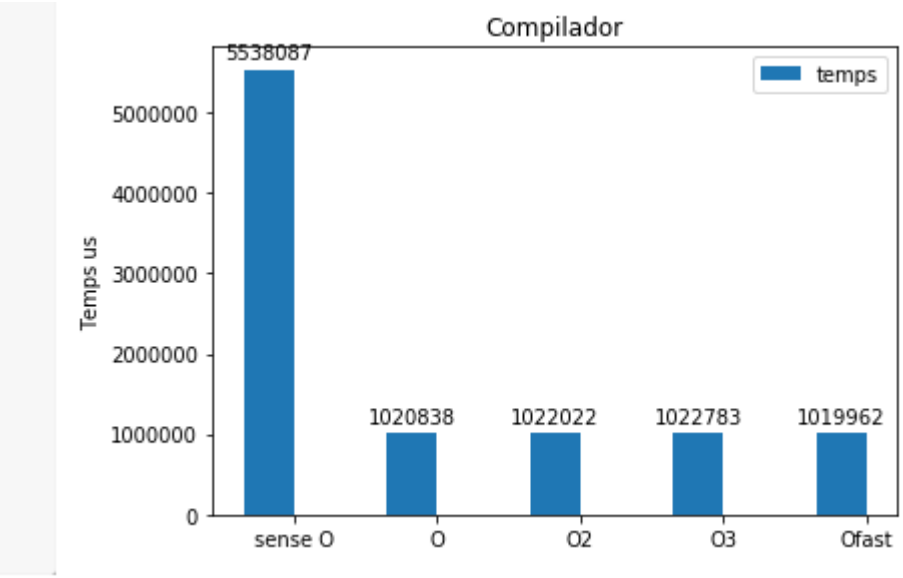
**IMPORTANT:**
**********
**Totes les proves han estat executades amb 10 repeticions.**
**Per executar: ./main <Exercici> <Repeticions>**
**Exemple: ./main 1 10**
**Scheduling ho hem fet en una altra carpeta i s´ha d'anar descomentant la linea del scheduling que vols provar.**
**********

**Exercici 1.**
Temps d'execució: 13320219us.

**Exercici 2.**

**Taula de comparacions entre compiladors.**

| **Compilador** | **Temps en us** |
|---|---|
| **sense O** | 5538087us |
| **O** | 1020838us |
| **O2** | 1022022us |
| **O3** | 1022783us |
| **Ofast** | 1019962us |




**Explicació de Compilador i hi ha algun risc en utilitzar-los?**

Parlarem ara de la variable -O. Aquesta variable controla el nivell d'optimització de tot el codi. En canviar aquest valor, la compilació de codi prendrà una mica més de temps, i utilitzarà molta més memòria, especialment en incrementar el nivell d'optimització.

- -O0: Aquest nivell (que consisteix en la lletra "O" seguida d'un zero) desconnecta per complet l'optimització i és el predeterminat si no s'especifica cap nivell -O en CFLAGS o CXXFLAGS. El codi no s'optimitzarà. Això, normalment, no és el que es desitja.
- -O1: El nivell d'optimització més bàsic. El compilador intenta produir un codi ràpid i petit sense prendre molt de temps de compilació. És bàsic, però aconseguirà realitzar correctament el treball.
- -O2: Un pas davant de -O1. És el nivell recomanat d'optimització, tret que el sistema tingui necessitats especials. -O2 activará algunes opcions afegides a les quals s'activen amb -O1. Amb -O2, el compilador intentarà aumentar el rendiment del codi sense comprometre la grandària i sense prendre molt més temps de compilació. Es pot utilitzar SSE o AVX en aquest nivell però no s'utilitzaran registres YMM tret que també s'habiliti ftree-*vectorize.
- -O3: El nivell més alt d'optimització possible. Activa optimitzacions que són cares en termes de temps de compilació i ús de memòria. El fet de compilar amb -O3 no garanteix una manera de millorar el rendiment i, de fet, en molts casos pot tenir un sistema degut a l'ús de binaris de gran grandària i molt ús de la memòria. També se sap que -O3 pot trencar alguns paquets. No es recomana utilitzar -O3. No obstant això, també habilita -ftree-*vectorize de manera que els bucles dins del codi es vectoriza i s'utilitzaran els registres AVX YMM.
- -Ofast: Nou en GCC 4.7. Consisteix en l'ajust -O3 més les opcions -ffast-*math, -fno-*protect-*parens i -fstack-*arrays. Aquesta opció trenca el compliment d'estàndards estrictes i no es recomana la seva utilització.

**Per què és més ràpid un compilador que un altre?**
Si optimitzem molt el codi, pot ser molt més lent el temps de compilació perquè utilitza més recursos, com memoria o procesador. En canvi si utilitzem un compilador que no optimitzi tant el codi, la compilació serà més ràpida però la execució serà més lenta, per això s'ha de buscar un equilibri, encara que normalment el -Ofast és el que anirà més ràpid en l'execució. Dit això, hi ha risc d'utilitzar un o l'altre, perquè a partir de O2 pot corrompre algún paquet en el temps de compilació.

**Exercici 3.**
Modificació dels fors. Hem invertit els fors per poder recorrer la imatge per files i no per columnes, així podem millorar la localitat de les dades.
Hem vist dos tipus de localitat de dades.
La temporal, estalviem el màxim d'accesos a memòria, és a dir, quantifiquem la probabilitat de ser visitada, fet que comporta més temps d'execució, perquè el accés a memòria es molt lent comparat amb la caché.
La espacial, es la probabilitat de visitar direccions de memòries propera altre ja visitada, una vegada que hem accedit a memòria, que la següent direcció de memòria estigui proper a la direcció actual.

**Versió original:**

```
void convertBRG2RGBA(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    for (int x = 0; x < width; ++x)
    {
        for (int y = 0; y < height; ++y)
        {
            rgba[width * y + x].x = brg[width * y + x].y;
            rgba[width * y + x].y = brg[width * y + x].z;
            rgba[width * y + x].z = brg[width * y + x].x;
            rgba[width * y + x].w = 255;
        }
    }
}
```

**Versió modificada:**

```
void convertBRG2RGBA_2(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    /* Per optimitzar el codi accedim de manera lineal al vector, de manera que */
    /* cada accés estigui a una posició contigua a la memoria de l'accés anterior */
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            uchar3 tmp_3 = brg[width * y + x];
            uchar4 tmp_4;

            tmp_4.x = tmp_3.y;
            tmp_4.y = tmp_3.z;
            tmp_4.z = tmp_3.x;
            tmp_4.w = 255;
            rgba[width * y + x] = tmp_4;
        }
    }
}
```

Amb 1000 iteracions:

- Temps d'execució: 994497us.

**Exercici 4.**
**a. Segons el que hem vist a teoria, quin problema pot tenir l'struct uchar3?**
A l'hora d'accedir i guardar les dades. Ja que per com està estructurada la memòria és més eficient accedir i segmentar en potencia de 2.

**Temps**:

- 100 iteraciones: 425561us

- 1000 iteraciones: 4162810us

**d. Busqueu a google que es __attribute__ i que fa concretament aligned(4). Pregunteu a classe. Expliqueu a la memoria que heu entès.**

- **__attribute__**
  Permet especificar atributs especials de tipus struct i union quan definim aquests tipus. Per exemple delimitar el struct a x bytes.

- **aligned(4)**
  Això alinea la memòria assignada al buffer en un límit de 4 bytes. Això pot accelerar les transferències de memòria entre la CPU i la memòria principal entre altres coses.

**Exercici 5.**

**a)**

```
void convertBRG2RGBA_3(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    #pragma omp parallel for
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            uchar3 tmp_3 = brg[width * y + x];
            uchar4 tmp_4;

            tmp_4.x = tmp_3.y;
            tmp_4.y = tmp_3.z;
            tmp_4.z = tmp_3.x;
            tmp_4.w = 255;

            rgba[width * y + x] = tmp_4;
        }
    }
}
```

- **Temps: 3859489us**

```
void convertBRG2RGBA_3(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    //#pragma omp parallel for
    for (int y = 0; y < height; ++y)
    {
        #pragma omp parallel for
        for (int x = 0; x < width; ++x)
        {
            uchar3 tmp_3 = brg[width * y + x];
            uchar4 tmp_4;

            tmp_4.x = tmp_3.y;
            tmp_4.y = tmp_3.z;
            tmp_4.z = tmp_3.x;
            tmp_4.w = 255;

            rgba[width * y + x] = tmp_4;
        }
    }
}
```

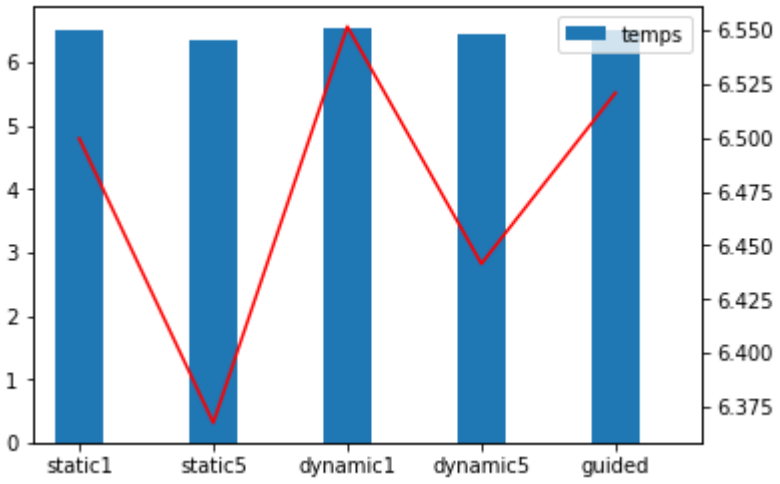
- **Temps: 5320674us**

Va millor paral·lelitzar el primer bucle, ja que si paral·lelitzar el bucle anidat, estem construint i destruint tasques per cada iteració del primer for, però més adalt del codi, aquesta funció és cridada dintre de un altre for en el main, i en aquest for es on hem obtingut un millor rendiment.

**b. Quin scheduling es millor? Feu taules amb resultats, modificant la mida de la imatge, el numero de iteraciones, i el chunk size del scheduling. Justifiqueu el raonament amb aquestes taules.**

**Aquets resultats es e+06us.**

| **Sheduling** | **Temps en us** |
|---|---|
| **Static 1** | 6.49968 |
| **Static 5** | 6.36754 |
| **Dynamic 1** | 6.55167 |
| **Dynamic 5** | 6.44142 |
| **Guided** | 6.52085 |




**Segons la teoria vista a classe, la planificació scheduling:**
“””

A. **Planificació estàtica**

La divisió entre tasques i fils es determina en iniciar la computació i no varia al llarg d’aquesta.

B. **Planificació dinàmica**

L’assignació de tasques a fils es realitza a mesura que es realitza la computació.

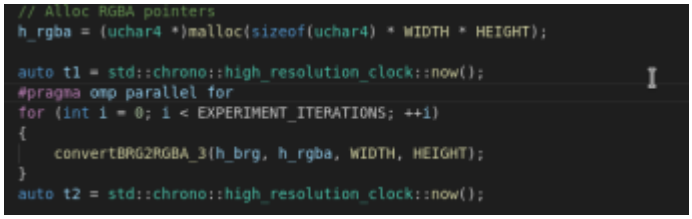
C. **Planificació guiada**

La mida del tros assignada al fil es proporcional al nombre d’iteracions que queden per processar dividit pel nombre de fils.
“””

Donat els resultats, hem pogut observar que el rendiment és relativament semblant, però el millor dels casos l'hem obtingut en el estàtic modificant el chunk_size, amb diferents paràmetres, tot i que el rendiment era molt aproximat entre els diferents valors.

**c. Canvia alguna cosa definir variables private i shared?**
En aquest cas no afecta posar les variables private o shared, ja que les variables son privades per defecte.

**Exercici 6.**



```
// Alloc RGBA pointers
h_rgba = (uchar4 *)malloc(sizeof(uchar4) * WIDTH * HEIGHT);

auto t1 = std::chrono::high_resolution_clock::now();
#pragma omp parallel for
for (int i = 0; i < EXPERIMENT_ITERATIONS; ++i)
{
    convertBRG2RGBA_3(h_brg, h_rgba, WIDTH, HEIGHT);
}
auto t2 = std::chrono::high_resolution_clock::now();
```

- Temps:1286845us.

Aquest for es el que millor treballa la paral·lelització, perquè aquest for crida contínuament a la funció convertBRG2RGBA  que conté els fors anidats, per tant, es crea les tasques d'una sola vegada i es reparteix la feina.

**Exercici 7**

**Exemple:**

```
(base) johnny@iMac-de-Johnny OpenMP % ./main 3 10
Soc el fil numero 0
Soc el fil numero 1
Soc el fil numero 2
Soc el fil numero 3
convertBRG2RGBA time for 1000 iterations = 2.42434e+06us
Executed 10 times
```