Blai Ras, Albert Morales i Alex Campos

Programació Paral·lela

Pràctica 1

Objectius

1. Mantingueu sempre una copia intacta de la funció “convertBGR2RGBA” per a poder comparar temps d’execució.
   Hem creat una funció anomenada “permaConvertBGR2RGBA” la qual no ha estat modificada.

2. Experimenteu amb diferents maneres d’accedir a la memòria a la funció “convertBGR2RGBA” per explotar millor la localitat de les dades.
   Hem vist dos tipus de localitat de dades, la temporal i la espacial. Quan parlem de temporal ens referim a com de probable és que una posició de memòria torni a ser “visitada” en un futur, és a dir, una bona localitat temporal és aquella que estalvia al màxim possible l’accés a memòria, fet que comporta més temps d’execució. Per altra banda, la localitat espacial fa referència a com de probable es que es visitin en un futur direccions de memòria “properes” a una que ja ha estat visitada. Per tant, direm que un codi amb una bona localitat espacial és aquell que referencia direccions de memòria properes, estalviant així temps d’execució.
   Ambdues localitats han estat testejades a classe i als nostres ordinadors amb diferents modificacions que hem cregut oportunes. Les pots veure al apartat de Resultats.

3. Paral·lelitzeu la funció “convertBGR2RGBA” amb OpenMP.
   La funció “convertBGR2RGBA” consta de dos *fors*, un dins de l’altre. Per tant, paral·lelitzar-la es pot dur a terme de diferents formes:
   a. Creant un bloc que engloba els dos fors i paralel·litzar-lo amb *#pragma omp parallel*

```
void convertBGR2RGBA(uchar3* bgr, uchar4* rgba, int width, int height) {

#pragma omp parallel
    {
    for (int y=0; y<height; ++y) {

        for (int x=0; x<width; ++x) {
            rgba[width * y + x].x = bgr[width * y + x].z;
            rgba[width * y + x].y = bgr[width * y + x].y;
            rgba[width * y + x].z = bgr[width * y + x].x;
            rgba[width * y + x].w = 255;
        }
    }
    }
}
```

Il·lustració 1: Creació d'un bloc paral·lelitzat que engloba els fors

b. La opció a) però paral·lelitzant a la vegada el primer *for* amb *#pragma omp for*
c. La opció b) però paral·lelitzant també el segon for niuat, amb *#pragma omp parallel for*

```
void convertBGR2RGBA(uchar3* bgr, uchar4* rgba, int width, int height) {

#pragma omp parallel
    {
#pragma omp for
    for (int y=0; y<height; ++y) {
#pragma omp parallel for
        for (int x=0; x<width; ++x) {
            rgba[width * y + x].x = bgr[width * y + x].z;
            rgba[width * y + x].y = bgr[width * y + x].y;
            rgba[width * y + x].z = bgr[width * y + x].x;
            rgba[width * y + x].w = 255;
        }
    }
    }
}
```

Il·lustració 2: Parel·lització adicional del segon for

En totes les opcions estem paral·lelitzant, però mirant temps d'execució, vam optar per la opció b). Triga menys que la a), i, fent proves amb la c), vam veure que trigava gairebé sempre el mateix, així que vam suposar que estàvem pràcticament fent el mateix tan a b) com a c).

4. Imprimiu de forma correcta l'id de cada un dels threads que executen el codi, utilitzant les funcions proporcionades per OpenMP per conèixer aquest id, i per executar parts de codi de forma ordenada (regió crítica).
Tal i com va comentar un alumne a classe, un *cout* en una funció o bloc paral·lelitzat no ens serveix per testejar com funciona el flux del programa ja que molt probablement en algun moment del programa hi haurà un o més fils executant aquest *cout*. Ho veiem, per exemple, quan s'imprimeix per pantalla l'*string* del *cout* concatenat amb ell mateix:

```
Soc el fil numero 4
Soc el fil numero 4
Soc el fil numero 5Soc el fil numero 4


Soc el fil numero 4


0Soc el fil numero 1
Soc el fil numero 6Soc el fil numero 1
Soc el fil numero 7


Soc el fil numero 0
Soc el fil numero Soc el fil numero Soc el fil numero 3
Soc el fil numero 25Soc el fil numero Soc el fil numero 3
```

Il·lustració 3: Exemple d'output d'un cout sense regió crítica

Per tant, si volem imprimir l'identificador de cada fil que executen el codi, continuarem fent servir un *cout*, però aquest estarà dins d'una secció crítica. Entenem com a secció crítica el tros/bloc de codi en el que s'assegura que un i només un fil hi accedeix a dur a terme les instruccions de dins del bloc. Amb OpenMP, podem crear una secció crítica amb la directiva *#pragma omp critical* aplicada a un bloc.
Així doncs, hem aconseguit l'objectiu encapsulant el *cout* en una secció crítica:

```
void convertBGR2RGBA(uchar3* bgr, uchar4* rgba, int width, int height) {

#pragma omp parallel
    {
#pragma omp for
    for (int y=0; y<height; ++y) {
        for (int x=0; x<width; ++x) {
#pragma omp critical
            {
            cout << "Soc el fil numero " << omp_get_thread_num() << endl;
            }
            rgba[width * y + x].x = bgr[width * y + x].z;
            rgba[width * y + x].y = bgr[width * y + x].y;
            rgba[width * y + x].z = bgr[width * y + x].x;
            rgba[width * y + x].w = 255;
        }
    }
    }
}
```

Il·lustració 4: Encapsulació del cout dins d'una regió crítica

```
Soc el fil numero 1
Soc el fil numero 3
Soc el fil numero 4
Soc el fil numero 0
Soc el fil numero 4
Soc el fil numero 0
Soc el fil numero 4
Soc el fil numero 1
```

Il·lustració 5: Output d'un cout encapsulat

5. Experimenteu amb diferents polítiques *d'scheduling*. Justifiqueu quina és més eficient i per què.

   Breument, a teoria hem vist 3 tipus de *scheduling*:

   - *Static*: es dona a cada fil un espai concret, fix en tota l'execució.
   - *Dynamic*: els trossos o "espais" es van donant als fils de manera dinàmica, s'assignen en temps d'execució
   - *Guided*: va assignant tros als fils tenint en compte les iteracions restants i el nombre de fils amb els que estem paral·lelitzant. Per tant, aquest tros disminueix a mesura que avancen les iteracions.

   D'aquí podem deduir que una planificació *static* ens convé quan el procediment que es dur a terme dins del *for* sempre és el mateix (no hi ha condicionals, accedim les mateixes vegades a memòria, etc). Una planificació *dynamic*, en canvi, potser va millor quan pot variar el que fem dintre d'un bucle, i una *guided* més o menys va pel mateix camí però en certes ocasions ens pot donar un rendiment millor que la dinàmica, segurament, amb funcions més complexes o llargues, que requereixin molts fils.
   Dins l'apartat de resultats podem veure les diferents proves que hem fet. Un cop comprovat que la millor planificació era la estàtica, hem trobat el valor de *chunk* més òptim per ella. De totes maneres, hem investigat també les altres dues planificacions, fins ajuntar-les totes per analitzar i comparar.

Resultats

1. Millores de localitat de dades
    a. Millores temporals: les millores que hem introduït en la funció "convertBGR2RGBA" són dues. La més destacable és la creació de dues variables temporals anomenades "tempbgr" i "temprgba". La primera és una copia d'un tros del vector "bgr", mentre que la segona és un vector del mateix tipus que el vector "rgba" (uchar4). La realització d'aquesta "duplicació" de variables és una millora temporal d'accés a memòria perquè són copies dels vectors reals però que es troben "a la memòria cache", una memòria que com sabem és de ràpid accés i d'emmagatzemat de dades temporal.
    Dit i fet, creem primer la còpia del vector bgr i la igualem al vector original en la posició que toqui segons les iteracions del *for*. En segon lloc, declarem el que serà el nou tros del vector rgba.
    Un cop tenim les variables creades, procedim a fer el mateix que fa "convertBGR2RGBA" però usant aquestes variables temporals, és a dir, assignem els valors "x", "y", "z" del vector "tempgbr" a la variable instanciada "temprgba". En aquesta última també li assignem el valor 255 corresponent a la *alpha*.
    Ara el que tenim és una variable temporal amb el resultat de la conversió, així que només falta igualar el vector original a la posició en qüestió a ella.
    La segona millora temporal realitzada fa referència a la posició on entrem de cadascun dels vectors. Aquesta es calcula mitjançant la *width* i les variables "x" i "y" dels *fors*. Així doncs, aquesta posició es calcula en cada iteració. No podem estalviar-nos el seu càlcul, però sí que podem no repetir-lo quan assignem cada valor del vector bgr al rgba. Així doncs, creem una variable entera anomenada "pos" que es calcula una vegada i no 4, de manera que el programa només a d'accedir a ella (a cache) en comptes d'accedir al valor de *width*, "x" i "y".

b. Millora espacials: la funció "convertBGR2RGBA" posseeix una millora espacial. Aquesta sorgeix al veure que tal i com està la funció original, anem accedint a la matriu de la fotografia fila per fila, és a dir, primer píxel per primer píxel. Això es tradueix en un salt de direccions de memòria de 3480 en 3480!
Si en canvi es recorres la matriu fent una fila sencera i després una altre, estaríem escurçant molt més la distància entre adreces de memòria. Aquest recorregut es pot aconseguir simplement "girant" els *fors*, és a dir, el *for* gran recorre la variable "x" (que correspon a la *width*) i el segon for recorre la "y", que correspon a la *heigth*.

```
51  void convertBGR2RGBA(uchar3* bgr, uchar4* rgba, int width, int height) {
52
53  #pragma omp parallel
54      {
55  #pragma omp for
56      for (int y=0; y<height; ++y) {
57          for (int x=0; x<width; ++x) {
58          #pragma omp critical
59              {
60              cout << "Soc el fil numero " << omp_get_thread_num() << endl;
61              }
62              int pos = width * y + x;
63              uchar3 tempbgr = bgr[pos];
64              uchar4 temprgba;
65
66              temprgba.x = tempbgr.z;
67              temprgba.y = tempbgr.y;
68              temprgba.z = tempbgr.x;
69              temprgba.w = 255;
70
71              rgba[pos] = temprgba;
72          }
73      }
74      }
75  }
```

Il·lustració 6: Línies 56 i 57: "gir" del for, opt. espacial. Línia 62 fins 71: millores temporals

2. Test de planificacions

A l'hora d'analitzar el temps d'execució del programa, hem decidit fer ús de la funció *multitime* (https://tratt.net/laurie/src/multitime/) la qual és una extensió de la funció *time* indicada a teoria on li pots indicar el número d'execucions que vols realitzar de cert executable (en aquest cas, el nostre codi). El resultat són 5 columnes: la mitjana de temps d'execució, la seva desviació mitjana, la execució que menys ha trigat, la mediana i la execució que més ha trigat, tan per temps "real" com d'usuari o de sistema. Aquí un exemple:

#pragma omp for schedule(static,1)	Mean	Std.Dev.	Min	Median	Max
real	0.057	0.014	0.037	0.061	0.086
user	0.135	0.041	0.060	0.122	0.241
sys	0.017	0.009	0.004	0.016	0.049

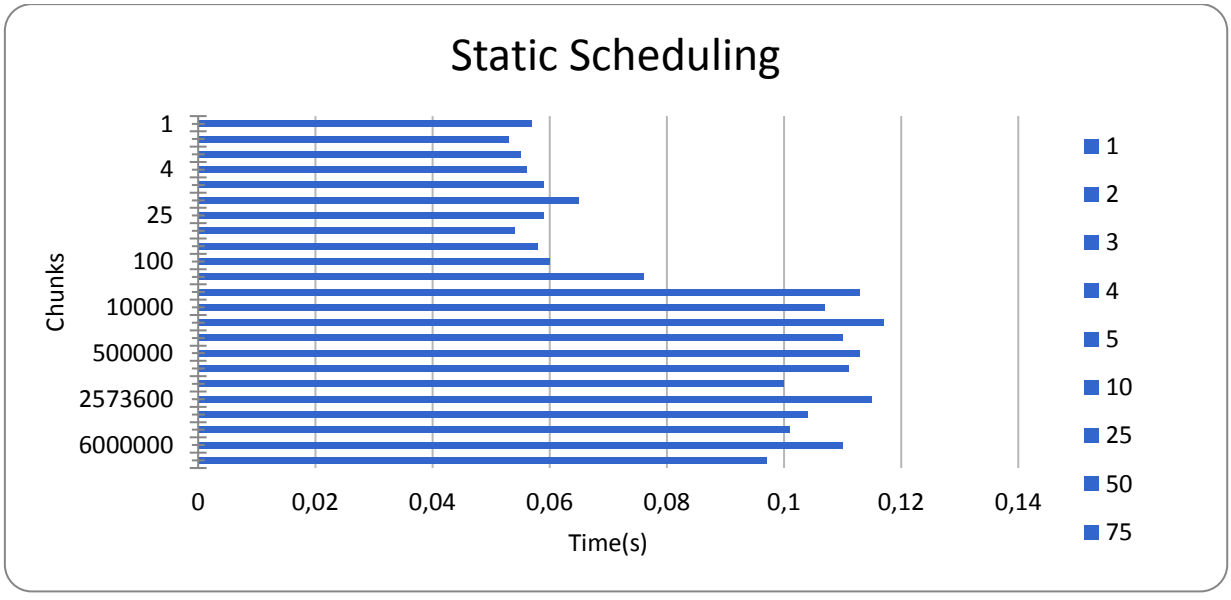
Il·lustració 7: Exemple d'output de multitime

D'aquesta manera, podem tenir una millor aproximació del temps mitjà que triga el nostre codi sense haver d'executar-lo manualment. Concretament, totes les mitjanes de temps d'execució que veurem a continuació són extretes executant el programa 50 vegades. A més a més, respecte l'arpartat 6) de la pràctica, afegir que totes aquestes execucions i temps han estat calculats amb l'apartat de *checkResults* comentat. Aquests són els tests realitzats amb les seves conclusions.

a. Planificació estàtica

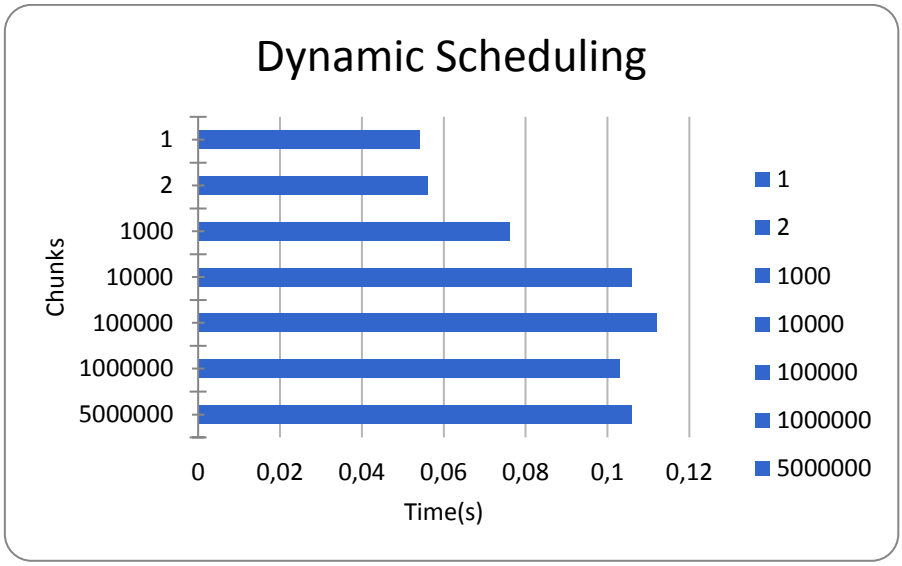
   Sobre el paper, aquesta planificació és la més òptima per la nostra funció ja que en els seus *fors* sempre es realitza el mateix i sempre s'accedeix a memòria el mateix nombre de vegades. Per tant, podem planificar i administrar els trossos que tindran cada fil en iniciar la execució sense perill de "mal-gestionar" recursos. En conseqüència, hem realitzat una gran multitud de proves seguint alguns criteris.

   - En primer lloc, hem provat amb la *chunk_size* mínima, 1.
   - Després, vam limitar el número de fils a 4, i de mida de tros vam especificar 2073600, el resultat de dividir la mida de la imatge (3840*2160) entre quatre.
   - El resultat no va millorar respecte d'1. Després, per assegurar-nos de que estava planificant correctament, vam especificar de mida 8294400, el resultat de la multiplicació de la mida de la imatge, i vam comprovar que efectivament el temps incrementava significativament.
   - Aleshores, en busca del millor valor, vam anar provant valors de potència de 10. Primer 100, després 1.000 i finalment 100.000. Al allunyar-nos molt del valor mínim fins aquell moment trobat (1), vam provar amb potències de 5 (5.000, 50.000, 500.000). El resultat va ser nefast (500.000 és el valor de *chunk* amb el que més temps va trigar)
   - Aleshores, vam provar amb valors semblants a 1, concretament, 2, 3, 4 i 5, trobant finalment el valor mínim (i que considerem més eficient): 2



b. Planificació dinàmica

De la planificació dinàmica hem dit que convé quan allò que realitzem dins d'un bucle pot variar en temps d'execució o en accessos a memòria, de manera que anem assignant dinàmicament a cada fil trossos de diferents mides. Ara que ja hem fet testos per la planificació estàtica, procedim a repetir-los per la dinàmica:

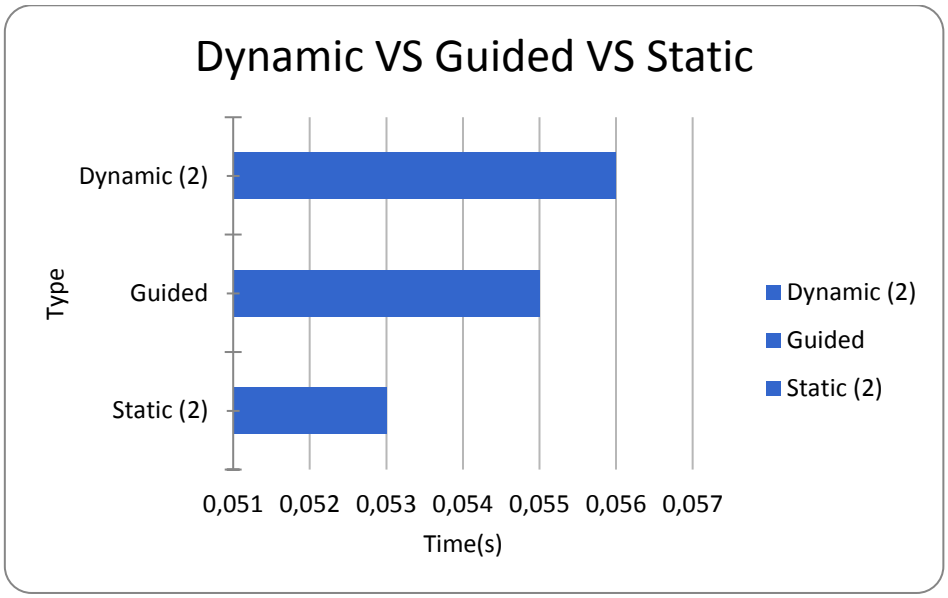


c. Planificació guiada

Una planificació *guided* és aquella que té en compte quantes iteracions restants ha de realitzar i amb quants fils ho ha de fer. Amb aquesta informació, va adjudicant trossos de mida variable a cada fil. A una planificació *guided*, per tant, no se li pot especificar manualment una mida de *chunk*, i convé paral·lelitzar processos amb iteracions de temps o recursos variables amb un volum de fils considerable.

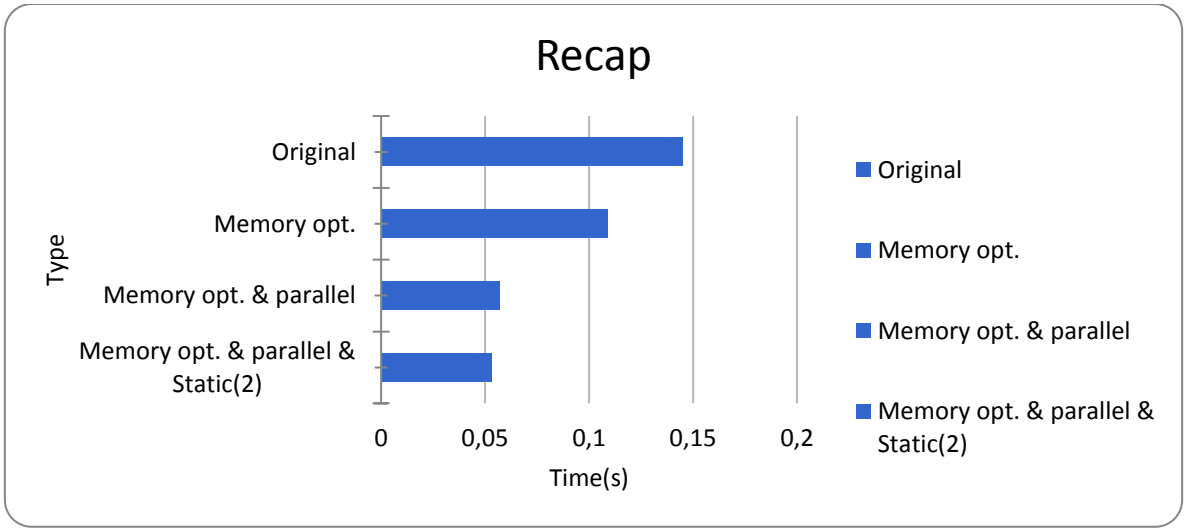
Efectivament, al comprovar-ho ens surt de mitjana un temps d'execució de 0,55 milisegons, on la execució que menys ha trigat ho ha fet en 0,35 ms i la que més 0,105 ms.

Un cop analitzat el comportament de cada planificació, podem comparar-les entre sí per poder esbrinar quina pot ser la més adequada o veure les seves diferències més enllà de la teoria de classe.



Aquesta gràfica mostra la competició entre les planificacions estàtica, dinàmica i *guided.* Aquesta última no té paràmetre *chunk_size*, però els resultats d'estàtica i dinàmica son amb *chunk_size* de dos, el mateix valor per a poder fer una comparació justa. Hem triat de mida dos per què en la estàtica és la *size* que millor resultat dona. Els resultats, però, no varien a gran escala: estem parlant de diferències de mil·lèsimes!

On si trobem grans diferències és en el resultat final. Hem analitzat el temps d'execució de la funció original, el temps d'execució de la funció un cop optimitzada i el temps d'execució de la funció optimitzada i paral·lelitzada, però sense especificar cap tipus de planificació. Finalment, li hem afegit la que per nosaltres és la millor planificació (estàtica de *chunk_size* 1) obtenint el següent gràfic:



**Atenció: pots trobar tots els tests amb tots els temps dintre d'un fitxer adjuntat anomenat "tests".*

Conclusions

Hem vist que la funció convertBGR2RGBA original funciona correctament i dintre d'un temps moderat. No obstant, a teoria hem vist certs mètodes per incrementar el rendiment d'un procés a gran escala.

El primer d'ells ha estat optimitzacions de memòria. Tal i com hem pogut veure a l'apartat de resultats, amb dues petites modificacions la diferència entre l'original i la optimitzada ja es força gran.

A continuació, hem aplicat conceptes de paral·lelització amb OpenMP. Sense dur a terme planificacions de cap tipus, especificacions sobre la compartició de variables o la creació de tasques, la funció ja millora en rendiment simplement per paral·lelitzar-la.

Finalment, hem provat diferents tipus de planificacions vistes a teoria i hem vist el seu rendiment aplicant mitjanes de temps d'execució. Hem arribat a la conclusió de que la millor planificació és la estàtica amb una *chunk_size* de dos.