UNIVERSITAT DE BARCELONA

Ingeniería Informática

Programación Paralela

Práctica 1

OpenMP

Nombre	NIUB
Manuel Ernesto Martínez Martín	20081703
José Manuel López Camuñas	18079924

Índice

Guía de ejecución

- Hemos modificado ligeramente el código para poder ejecutar todos los ejercicios con el mismo main.

  El modo de ejecución sería:

  ./main <modo> <ejercicio>

  Por ejemplo:

  ./main -e 1
  ./main --exercise 1

- Hemos creado un script en bash runTest.sh para hacer n iteraciones y quedarnos con la media

  El modo de ejecución sería:

  ./runTest.sh <file> <option> <exercise> <ntimes>

  (del ejercicio 1 al 6)

  Por ejemplo:

  ./runTest.sh ./main -e 6 4
  ./runTest.sh ./main -exercise 6 4

Cuestiones planteadas

Punto 1

c) Tiempo de ejecución en micro segundos

Mac i5-6287U dual core: **12192954us**
Windows 10 i7-10710U hexa core: **9204367us**

Punto 2

b) Hay algún riesgo al utilizar los flags (-O, -O2, -O3, Ofast)?

Son Flags para asignar el nivel de optimización.

Las optimizaciones pueden hacer cambios en el código, se consigue velocidad de ejecución pagando el precio en tiempo de compilación, pero para algunas tareas que requieren un orden concreto que puede ser cambiado no es buena idea.

También pueden algunos cálculos ser menos precisos.

-0: Disminuye el tiempo de ejecución y el tamaño del código pero aumenta el uso de memoria y el tiempo de compilación

> Mac i5-6287U dual core: **9387449us**
> Windows 10 i7-10710U hexa core: **7590183us**

-02: Disminuye aún más el tiempo de ejecución pero aumenta la memoria y el tiempo de compilación, este último aun más que el -O

> Mac i5-6287U dual core: **9463217us**
> Windows 10 i7-10710U hexa core: **7586546us**

-03: Igual que O2 pero con tiempo de ejecución más rápido y tiempo de compilación más lento

> Mac i5-6287U dual core: **1048885us**
> Windows 10 i7-10710U hexa core: **7590936us**

-Ofast: Igual que O3 pero los cálculos matemáticos son más rápidos y menos precisos

> Mac i5-6287U dual core: **1059854us**
> Windows 10 i7-10710U hexa core: **7577719us**

Aunque en alguna optimización superior tarde más se debe a que el programa éste no es de tamaño significativo para apreciarlo bien

e) Tiempo y Speedup respecto el Punto 1, ¿Qué está pasando?

Porcentajes

Mac i5-6287U dual core				
Optimización	**-O**	**-O2**	**-O3**	**-Ofast**
% Mejora	23.09%	23%	91%	91.30%

Windows 10 i7-10710U hexa core				
Optimización	**-O**	**-O2**	**-O3**	**-Ofast**
% Mejora	17.51%	17.57%	17.52%	17.67%

Speedup

Mac i5-6287U dual core				
Optimización	**-O**	**-O2**	**-O3**	**-Ofast**
Speedup	1.29	1.28	11.62	11.50

Windows 10 i7-10710U hexa core				
Optimización	**-O**	**-O2**	**-O3**	**-Ofast**
Speedup	1.212	1.213	1.212	1.214

Hay mejora en la ejecución porque la optimización del compilador reordena y modifica el código como hemos explicado antes.

Punto 3

b) Explicar cambios y los Speedups obtenidos al mejorar la localidad de datos

Mac i5-6287U dual core: **1109857us**
Windows 10 i7-10710U hexa core: **989390us**

Hemos intercambiado **x** e **y** en los fors de manera que en vez de leer y modificar columnas leemos y modificamos filas, entonces hay más probabilidad de tener un hit en la caché y nos evitamos tener que ir yendo a memoria y traer los datos.

Mac i5-6287U dual core		Windows 10 i7-10710U hexa core	
Porcentaje	**Speedup**	**Porcentaje**	**Speedup**
90.89%	10.98	89.25%	9.30

c) Si el tiempo < 1s, argumenta EXPERIMET_ITERATIONS a 1000

1000 Iteraciones sin Ofast

Mac i5-6287U dual core: **163628044us**

Windows 10 i7-10710U hexa core: **90607474us**

1000 Iteraciones con Ofast

Mac i5-6287U dual core: **10937280us**

Windows 10 i7-10710U hexa core: **9811910us**

Mac i5-6287U dual core		Windows 10 i7-10710U hexa core	
Porcentaje	**Speedup**	**Porcentaje**	**Speedup**
93.31%	14.96	89.17%	9.23

Punto 4

a) ¿Qué problema puede tener el struct uchar3?

La memoria se gestiona peor a no ser un número par potencia de 2, afecta a la caché.

c) Medidas de la nueva ejecución

Mac i5-6287U dual core: **2636056us**
Windows 10 i7-10710U hexa core: **6092779us**

Mac i5-6287U dual core		Windows 10 i7-10710U hexa core	
Porcentaje	**Speedup**	**Porcentaje**	**Speedup**
98.38%	62.07	93.27%	9.23

d) ¿Qué es __attribute__?, ¿Qué hace aligned(4)?

__atribute__ es un mecanismo que ayuda al desarrollador a añadir características a la declaración de una función para dejar al compilador hacer más comprobaciones de errores

aligned fuerza a que ocupe un cierto tamaño fijo del número indicado de bits y si es más pequeño funciona como un padding y añade los ceros que falten.

Punto 5

a) ¿Qué bucle va mejor para paralelizar?

```
void convertBRG2RGBA3(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)
        {
            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;
        }
    }
}
```

Mac i5-6287U dual core: 2040924us
Windows 10 i7-10710U hexa core: 5251015us

```
void convertBRG2RGBA3(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    for (int y = 0; y < height; ++y)
    {
        #pragma omp parallel for
        for (int x = 0; x < width; ++x)
        {
            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;
        }
    }
}
```

Mac i5-6287U dual core: **80425420us**
Windows 10 i7-10710U hexa core: **10033013us**

```
void convertBRG2RGBA3(uchar3 *brg, uchar4 *rgba, int width, int height)
{
    #pragma omp parallel
    for (int y = 0; y < height; ++y)
    {
        #pragma omp for
        for (int x = 0; x < width; ++x)
        {
            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;
        }
    }
}
```

Mac i5-6287U dual core: **71623987us**
Windows 10 i7-10710U hexa core: **13334297us**

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

Mac i5-6287U dual core: **2305483us**
Windows 10 i7-10710U hexa core: **5618477us**

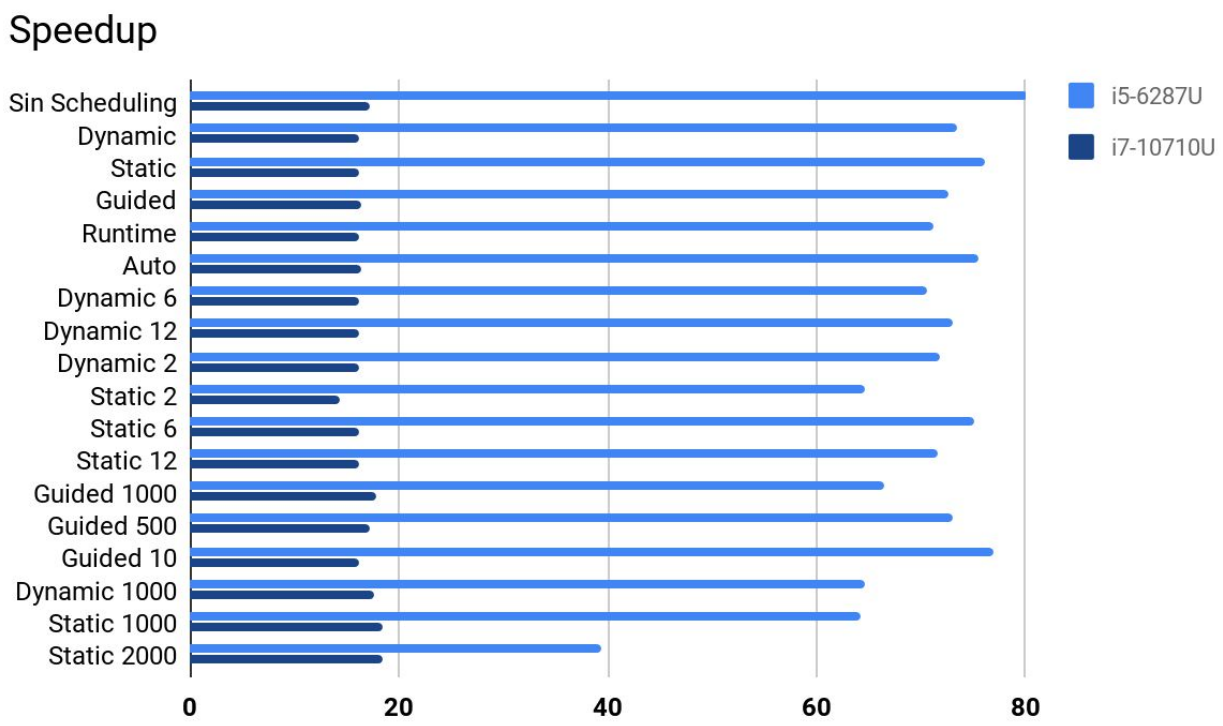
```
#pragma omp parallel for
    for (int y = 0; y < height; ++y)
    {
#pragma omp task
        {
            for (int x = 0; x < width; ++x)
            {

                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;
            }
        }
    }
```

Mac i5-6287U dual core: **2247048us**
Windows 10 i7-10710U hexa core: **5984614us**

El bucle que va mejor para paralelizar es el for más externo respecto a las pruebas realizadas, aunque no se nota mucho la diferencia si se lo ponemos a ambos fors, donde si que se nota es si lo ponemos en el for interno.

b) ¿Qué scheduling es mejor?, añade tabla de resultados, modificando el número de iteraciones y el chunk size del scheduling.

Sin Collapse(Media de 10 loops)

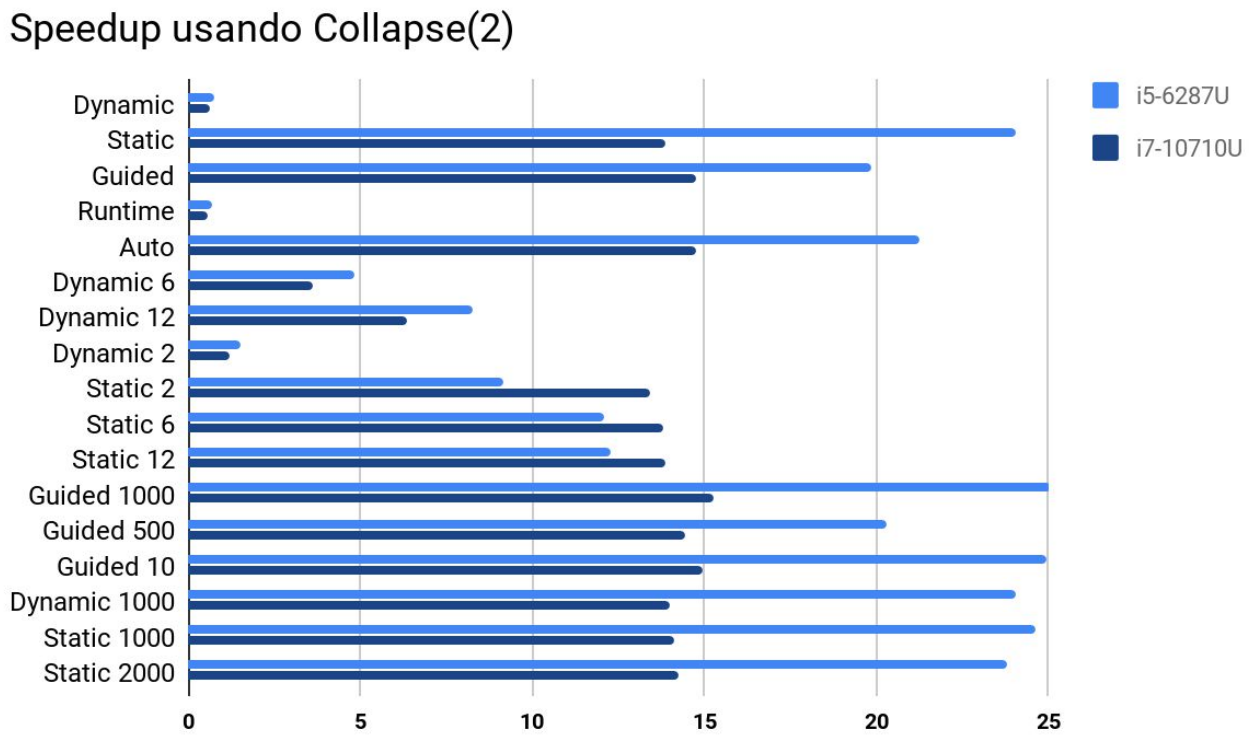


Speedup	i5-6287U	i7-10710U
Sin Scheduling	80,17351161	17,25523046
Dynamic	73,40849266	16,12508654
Static	76,19019513	16,29450884
Guided	72,6994066	16,40394292
Runtime	71,16341992	16,10252195
Auto	75,49031181	16,31510586
Dynamic 6	70,66584842	16,28169574
Dynamic 12	73,05482191	16,23828561
Dynamic 2	71,85199445	16,09536517
Static 2	64,57506314	14,43609536
Static 6	75,00980736	16,11390524
Static 12	71,54449012	16,27508919
Guided 1000	66,40112554	17,82173675
Guided 500	73,12419822	17,24614268
Guided 10	76,89905721	16,14681088
Dynamic 1000	64,6013481	17,56435351

Static 1000	64,1045854	18,40582429
Static 2000	39,41510019	18,38664866

Hay más margen de mejora respecto al tiempo original en el i5-6287U

Con Collapse(Media de 10 loops)



Speedup	i5-6287U	i7-10710U
Dynamic	0,7812579464	0,6213226971
Static	24,03418438	13,89113773
Guided	19,81950193	14,78492467
Runtime	0,7089624488	0,5947826413
Auto	21,26404574	14,75689596
Dynamic 6	4,847912561	3,614372448
Dynamic 12	8,268643416	6,34708845
Dynamic 2	1,553974492	1,207633813
Static 2	9,188935946	13,42757458
Static 6	12,06829098	13,83187808
Static 12	12,26547013	13,89289492
Guided 1000	25,07197672	15,27259142
Guided 500	20,28378851	14,42793023
Guided 10	24,94831187	14,95456876
Dynamic 1000	24,07083793	13,9789352

Static 1000	24,62009311	14,10868773
Static 2000	23,76098053	14,21947917

En el i5-6287U va mejor sin scheduling o con static por defecto, en el i7-10710U va mejor el static de 2000, aunque en ambos va peor usar collapse.

Normalmente el mejor método es aquel que hace que el tiempo de ejecución del peor thread(thread que más tiempo de ejecución tiene) sea mejor. Se busca pues distribuir la carga de una manera más inteligente.

c) ¿Cambia alguna cosa definir variables private y shared?

No tiene sentido porque todo se pasa por parámetro, la **i** de los bucles ya por defecto es privada

Punto 6

Paraleliza el bucle que itera EXPERIMENT_ITERATIONS_2, ejecutando la función “convertBRG2RGBA_2”. ¿Cómo compara con el Punto 5?, ¿Por qué crees que pasa?

Mac i5-6287U dual core: **3065771us**
Windows 10 i7-10710U hexa core: **7513731us**

Obtenemos mejores tiempos con la implementación del punto 5 ya que paralelizamos el acceso a la matriz con varias tareas, de la otra forma a cada tarea se le asigna una matriz entera lo que genera un mayor tiempo de ejecución.

Punto 7

Imprimimos el id con omp_get_thread_num() y utilizamos omp_critical

```
#pragma omp parallel
{
    #pragma omp critical
    {
    cout << "Thread ID: " << omp_get_thread_num() << endl;
    }
    #pragma omp for
    for (int i = 0; i < EXPERIMENT_ITERATIONS_2; ++i)
    {
        convertBRG2RGBA2(h_brg, h_rgba, WIDTH, HEIGHT);
    }
}
```

Creamos los threads y marcamos la sección crítica (en este caso imprimir el número del thread) para asegurarnos que ese trozo de código se ejecuta de manera no concurrente. Una vez salimos de la sección crítica procedemos a paralelizar la ejecución como en los demás ejercicios.