MEMORIA PRÁCTICA 1 METODOLOGÍA DE LA PROGRAMACIÓN PARALELA

Alberto Gálvez Gálvez – alberto.galvezg@um.es
Antonio Martínez Fernández – antonio.martinezf@um.es
Mario Rodríguez Béjar – mario.rodriguezb1@um.es

ÍNDICE

JUSTIFICAR ZONAS PARALELIZABLES

GA.C

Como primer ejemplo, podemos ver el bucle de la función *"init_imagen_aleatoria"* de *"ga.c"* (vid *ilustración 1*), en la cual se podría paralelizar la inicialización de una imagen aleatoria.

Sin embargo, esta paralelización podría no obtener una mejora considerable debido a la poca computación presente en el bucle.

```
void init_imagen_aleatoria(RGB *imagen, int max, int total)
{

    for (int i = 0; i < total; i++)
    {

        imagen[i].r = aleatorio(max);
        imagen[i].g = aleatorio(max);
        imagen[i].b = aleatorio(max);
    }
}
```

Ilustración 1. Función init_imagen_aleatoria en fichero ga.c

Luego, en la función *"crear_imagen"* de *"ga.c"* encontramos el bucle correspondiente a la *ilustración 2*, el cual es el resultado de unir dos bucles inicialmente separados que tenían el mismo número de iteraciones y, por tanto, se podían juntar.

En este caso la mejora esperada es mucho mayor, al contar con una gran cantidad de computación, y por supuesto, sin dependencias entre iteraciones.

Un apunte importante es que como ya se hace de la paralelización dentro de la función *"fitness"* deberemos de indicarlo con un *"omp_set_nested(1)"*.

```
for (i = 0; i < tam_poblacion; i++)
{

    poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
    poblacion[i]->imagen = imagen_aleatoria(max, num_pixels);

    fitness(imagen_objetivo, poblacion[i], num_pixels);
}
```

Ilustración 2. Función crear_imagen en fichero ga.c

Otro caso en el que se podría paralelizar sería el cruce de los individuos (vid *ilustración 3*), en el que encontramos dos bucles los cuales no tienen dependencias entre ellos, aunque no cuentan con una gran cantidad de computación.

```
void cruzar(Individuo *padre1, Individuo *padre2, Individuo *hijo1, Individuo *hijo2, int num_pixels)
{
    int random_number = aleatorio(num_pixels);

        for (int i = 0; i < random_number; i++)
        {
            hijo1->imagen[i] = padre1->imagen[i];
            hijo2->imagen[i] = padre2->imagen[i];
        }

        for (int i = random_number; i < num_pixels; i++)
        {
            hijo1->imagen[i] = padre2->imagen[i];
            hijo2->imagen[i] = padre1->imagen[i];
        }

}
```

Ilustración 3. Función cruzar en fichero ga.c

Además, en la función *"crear_imagen"* se llama a la función *"cruzar"* dentro de un bucle para aplicar el operador sobre toda la población (vid *ilustración 4*), sin embargo, aquí no sería posible debido a la existencia de dependencias entre iteraciones.

```
for (i = 0; i < (tam_poblacion / 2) - 1; i += 2)
{
    //! Hay problemas al paralelizar por choques de la poblacion[i] y poblacion[i+1]
    cruzar(poblacion[i], poblacion[i + 1], poblacion[tam_poblacion / 2 + i],
    poblacion[tam_poblacion / 2 + i + 1], num_pixels);
}
```

Ilustración 4. Extracto de la función crear_imagen en fichero ga.c

Continuando, esta vez con la función *"fitness"* (vid *ilustración 5*), se podrían agilizar los cálculos de la diferencia existente entre una imagen y la objetivo.

```
for (int i = 0; i < num_pixels; i++)
{
    diff +=abs(objetivo[i].r - individuo->imagen[i].r) + abs(objetivo[i].g - individuo->imagen[i].g)
    + abs(objetivo[i].b - individuo->imagen[i].b);
}

individuo->fitness=diff;
```

Ilustración 5. Extracto de la función fitness en ga.c

Finalmente, sería posible paralelizar la función *"mutar*" (vid *ilustración 6*), al igual que hacíamos con la función *"fitness"*. Sin embargo, dentro de la función, no sería demasiado interesante paralelizar debido a la poca computación existente, sería más lógico hacerlo en la función "*crear_imagen"*.

```
void mutar(Individuo *actual, int max, int num_pixels, float prob_mutacion)
{
    for (int i = 0; i < num_pixels; i++)
    {
        if (aleatorio(1500)<= 1)
        {
            actual->imagen[i].r = aleatorio(max);
            actual->imagen[i].g = aleatorio(max);
            actual->imagen[i].b = aleatorio(max);
        }
    }
}
```

Ilustración 6. Función mutar en fichero ga.c

IMAGEN.C

En este fichero, simplemente se podría hacer uso de la paralelización en la función *"suavizar"* (vid *ilustración 7*), en la cual se debe recorrer toda la matriz para ir calculando los nuevos valores mediante un suavizado de media.

De esta paralelización, se deberá de obtener una mejora significativa, ya que se reparte mucha carga entre los distintos hilos, sin embargo, como veremos posteriormente, el peso del tiempo de esta función no es significativo con respecto a la de la ejecución y cálculo de las distintas generaciones.

```
void suavizar(int ancho, int alto, RGB *imagen)
{

    RGB *imagen_auxiliar = (RGB *) malloc(size*sizeof(RGB));

    for(int i=0;i<size;i++){

        imagen_auxiliar[i].r=imagen[i].r;
        imagen_auxiliar[i].g=imagen[i].g;
        imagen_auxiliar[i].b=imagen[i].b;
    }

    for(int i=0;i<alto;i++){

        for(int j=0;j<ancho;j++){

            suavizar_pixel(i,j,ancho,alto,imagen_auxiliar, imagen);

        }
    }


}
```

Ilustración 7. Función suavizar en fichero imagen.c

PARALELIZACIÓN DE LA FUNCIÓN FITNESS

En este ejercicio se nos pedía la paralelización de la función *"fitness"* a través del uso de los pragmas:

1. *Critical*
2. *Reduction*
3. *Atomic*

Comparando sus tiempos de ejecución y mostrando las concusiones obtenidas.

IMPLEMENTACIÓN

Primero, vamos a mostrar las implementaciones llevadas a cabo para cada uno de estos pragmas.

Las variables indicadas en la directiva *firstprivate (diff)* son privadas a cada hilo. La variable *diff* en caso de *reduction* es usada por todos los hilos para juntar el resultado por lo que es global. La variable *individuo->fitness* es global ya que está protegida de ser accedida por varios hilos.

CRITICAL

Para esta implementación (vid *ilustración 8*) hemos creado una región paralela en la cual cada uno de los hilos cuente con una variable privada a ellos, *"diff"*.

En esta región paralela se ha repartido el trabajo del bucle *"for"* entre los distintos hilos, los cuales han calculado la diferencia existente entre dos matrices en la parte correspondiente a ese hilo en su variable privada *"diff"*.

Luego, una vez realizados estos cálculos se han volcado sobre la variable compartida *"individuo->fitness"*, a la cual solo debe poder acceder un hilo a la vez para evitar resultados no deseados, por lo que lo protegemos con un *"pragma omp critical"*.

```
#pragma omp parallel firstprivate(diff)
{
    #pragma omp for
    for(int i=0;i<num_pixels;i++){

        diff += abs(objetivo[i].r - individuo->imagen[i].r) + abs(objetivo[i].g - individuo->imagen[i].g)
        + abs(objetivo[i].b - individuo->imagen[i].b);

    }
    #pragma omp critical

        individuo->fitness += diff;

}
```

Ilustración 8. Extracto de la función fitness en fichero ga.c con el uso del pragma critical

ATOMIC

En esta implementación (vid *ilustración 9*) se ha usado la misma estrategia que en el *"critical"*, pero intercambiando esa directiva por la de *"atomic"*.

```
#pragma omp parallel firstprivate(diff)
{
    #pragma omp for
    for(int i=0;i<num_pixels;i++){

        diff += abs(objetivo[i].r - individuo->imagen[i].r) + abs(objetivo[i].g - individuo->imagen[i].g)
        + abs(objetivo[i].b - individuo->imagen[i].b);

    }
    #pragma omp atomic
    individuo->fitness += diff;
}
```

Ilustración 9. Extracto de la función fitness en fichero ga.c con el uso del pragma atomic

REDUCTION

En este caso la implementación (vid *ilustración 4*) ha sido mucho más simple, solo hemos indicado la directiva "*reduction"* indicando la operación a realizar, la suma, y el lugar donde almacenar el resultado, *"diff"*.

```
#pragma omp parallel for reduction(+: diff)
for (int i = 0; i < num_pixels; i++)
{
    diff +=abs(objetivo[i].r - individuo->imagen[i].r) + abs(objetivo[i].g - individuo->imagen[i].g)
    + abs(objetivo[i].b - individuo->imagen[i].b);
}

individuo->fitness=diff;
```

Ilustración 10. Extracto de la función fitness en fichero ga.c con el uso del pragma reduction

COMPARANDO RESULTADOS

Hilos	Atomic	Critical	Reduction
4	73,506(s)	71,336(s)	73.102(s)
12	70.853(s)	83,420(s)	65,603(s)
24	84.233(s)	110,926(s)	75,654(s)

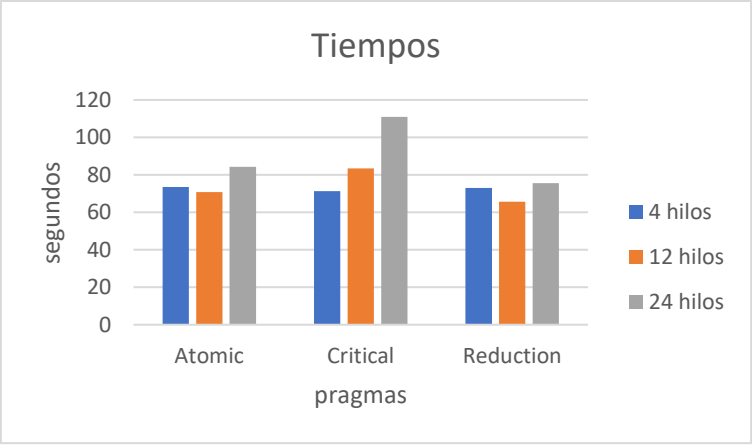
Tabla 1. Tiempos

Hilos	Atomic	Critical	Reduction
4	1,1485729	1,18351183	1,15492052
12	1,1915797	1,01207145	1,2869381
24	1,0023031	0,76111101	1,11596214

Tabla 2. SpeedUp obtenido

Hilos	Atomic	Critical	Reduction
4	0,28714323	0,29587796	0,28873013
12	0,09929831	0,08433929	0,10724484
24	0,04176263	0,03171296	0,04649842

Tabla 3. Eficiencia



La primera aclaración que hacer sobre estos resultados es el número de generaciones y tamaño de población usado, siendo 2000 y 256 respectivamente.

Luego, en cuanto a los resultados obtenidos podemos ver que la directiva *"critical"* obtiene buenos resultados a bajos números de hilos, pero a medida que van subiendo el rendimiento se va a pique.

Esto se debe a que esta directiva consta de un gran *overhead* a la hora de que los hilos entren y salgan de la sección critica, por lo que a mayor número de hilos mayor *overhead*, el cual no se ve compensado debido a una carga de computación a los hilos insuficiente.

"Atomic" ofrece unos resultados mucho mejores en general, ya que esta directiva no aporta un *overhead* tan grande, por lo que podemos ver que en los 12 hilos alcanza su mejor rendimiento, sin embargo, acaba pasando lo mismo al seguir aumentando el número de hilos debido a la falta de carga computacional.

Finalmente, la directiva *"reduction"* ofrece el mayor rendimiento entre las distintas directivas, probablemente debido al uso de optimizaciones internas y que parece decirnos que de forma interna parezca que haga uso de *"atomic"*. Cabe destacar que encontramos el mismo problema con un número de hilos mayor (24).

DIFERENCIAS EN SCHEDULING

En la ilustración, podemos observar qué tiempo nos ha llevado ejecutar los diferentes bucles con la directiva schedule con diferentes tamaños de chunk. En la primera columna se ha indicado si el bucle sobre el que se itera depende del tamaño de población del algoritmo genético o del número de pixeles de la imagen de la entrada.

1)

```
#pragma omp parallel
{
    randomSeed = omp_get_thread_num() * time(NULL);
    omp_set_nested(1);
    //#pragma omp for schedule(dynamic)
    //#pragma omp for schedule(dynamic,16)
    //#pragma omp for schedule(dynamic,8)
    //#pragma omp for schedule(guided)
    //#pragma omp for schedule(guided,16)
    //#pragma omp for schedule(guided,8)
    for (i = 0; i < tam_poblacion; i++)
    {
        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
        poblacion[i]->imagen = imagen_aleatoria(max, num_pixels);
        fitness(imagen_objetivo, poblacion[i], num_pixels);
    }
}
```

Ilustración 11. Uso de schedules en inicialización de población y cálculo del fitness

2)

```
//#pragma omp for schedule(dynamic)
//#pragma omp for schedule(dynamic,16)
//#pragma omp for schedule(dynamic,8)
//#pragma omp for schedule(guided)
//#pragma omp for schedule(guided,16)
//#pragma omp for schedule(guided,8)
for (i = mutation_start; i < tam_poblacion; i++)
{
    mutar(poblacion[i], max, num_pixels, prob_mutacion);
}
```

Ilustración 12.Uso de schedules rodeando la función mutar

3)

```
//#pragma omp for schedule(dynamic)
//#pragma omp for schedule(dynamic,16)
//#pragma omp for schedule(dynamic,8)
//#pragma omp for schedule(guided)
//#pragma omp for schedule(guided,16)
//#pragma omp for schedule(guided,8)
for (i = 0; i < tam_poblacion; i++)
{
    fitness(imagen_objetivo, poblacion[i], num_pixels);
}
```

Ilustración 13.Uso schedule en cálculo fitness

4)

```
//#pragma omp parallel for reduction(+: diff) schedule(static)
//#pragma omp parallel for reduction(+: diff) schedule(dynamic)
//#pragma omp parallel for reduction(+: diff) schedule(dynamic,128)
//#pragma omp parallel for reduction(+: diff) schedule(dynamic,256)
//#pragma omp parallel for reduction(+: diff) schedule(guided)
//#pragma omp parallel for reduction(+: diff) schedule(guided,128)
//#pragma omp parallel for reduction(+: diff) schedule(guided,256)
for (int i = 0; i < num_pixels; i++)
{
    diff +=abs(objetivo[i].r - individuo->imagen[i].r)
        + abs(objetivo[i].g - individuo->imagen[i].g)
        + abs(objetivo[i].b - individuo->imagen[i].b);
}

individuo->fitness=diff;
```

Ilustración 14.Uso schedule en cálculo de fitness

5)

```
void mutar(Individuo *actual, int max, int num_pixels, float prob_mutacion)
{

    //#pragma omp for schedule(dynamic)
    //#pragma omp for schedule(dynamic,256)
    //#pragma omp for schedule(dynamic,128)
    //#pragma omp for schedule(guided)
    //#pragma omp for schedule(guided,256)
    //#pragma omp for schedule(guided,128)
    for (int i = 0; i < num_pixels; i++)
    {
        if (aleatorio(1500)<= 1)
        {
            actual->imagen[i].r = aleatorio(max);
            actual->imagen[i].g = aleatorio(max);
            actual->imagen[i].b = aleatorio(max);
        }
    }
}
```

Ilustración 15.Uso schedule en mutar

6)

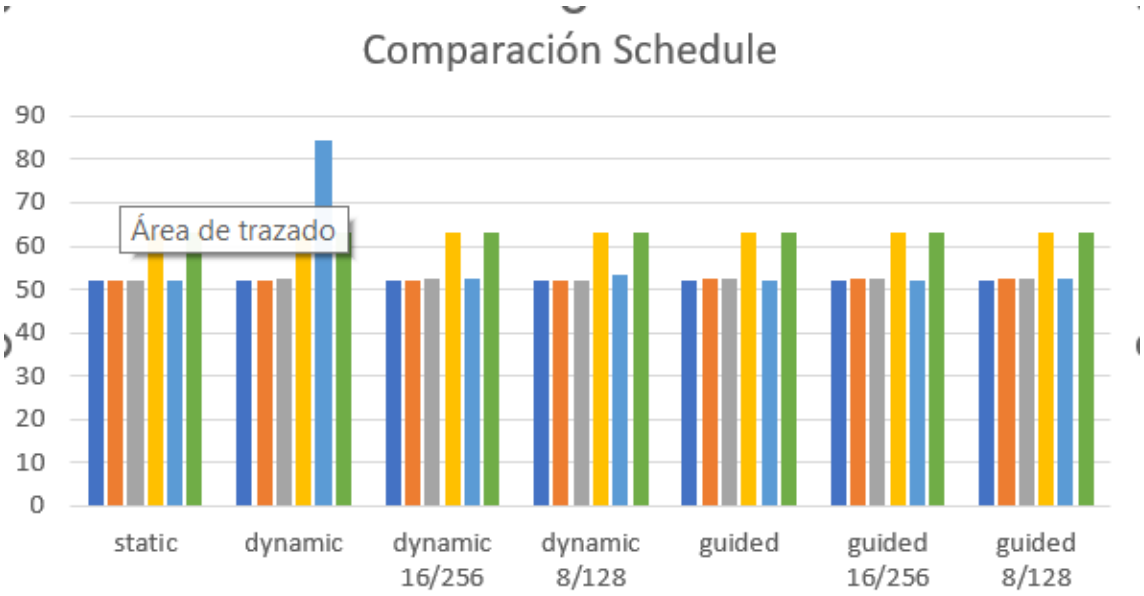
```
//#pragma omp for schedule(static)
//#pragma omp for schedule(dynamic)
//#pragma omp for schedule(dynamic,16)
//#pragma omp for schedule(dynamic,8)
//#pragma omp for schedule(guided)
//#pragma omp for schedule(guided,16)
//#pragma omp for schedule(guided,8)
for(int i=0;i<alto;i++){
    for(int j=0;j<ancho;j++){
        suavizar_pixel(i,j,ancho,alto,imagen_auxiliar, imagen);
    }
}
```

Ilustración 16.Uso schedule en suavizar_pixel

Sobre qué itera	Id bucle	static	dynamic	dynamic 16/256	dynamic 8/128	guided	guided 16/256	guided 8/128
Tam pob	1	52,165	52,117	52,219	52,176	52,211	52,000	51,953
Tam pob	2	52,175	52,053	52,147	52,229	52,278	52,303	52,500
Tam pob	3	52,139	52,296	52,270	52,247	52,430	52,326	52,328
Num pix	4	63,186	63,196	63,091	63,187	63,173	63,229	63,249
Num pix	5	52,135	84,600	52,547	53,275	52,244	52,211	52,482
Num pix altura	6	63,132	63,133	63,193	63,147	63,089	63,065	63,083

El tiempo viene dado en segundos.

Tabla 4. Tiempos de aplicar independientemente distintos schedules a diferentes bucles



Nota: los tamaños de chunk 16 y 8 corresponden a los bucles que iteran sobre el tamaño de población, mientras que los tamaños 128 y 256 son aquellos que lo hacen sobre el número de píxeles.

Nota: para la elección de los tamaños de chunk se ha llevado a cabo el siguiente procedimiento (ejemplo de un bucle de 1000 iteraciones con 16 cores):

1. Dividir el número total de iteraciones a paralelizar entre el número de cores (1000/16 = 62.6)
2. Obtener el de las dos potencias de dos más cercanas por abajo (32, 16)

Comparando resultados

Sin paralelizar:

Hilos	Secuencial
1	84,427(s)

Tabla 5. Tiempo secuencial

Paralelizando con 4 hilos:

Id bucle	T con mejor schedule	SpeedUp	Eficiencia
1	52,117(s)	1,6199	0,4050
2	52,503(s)	1,6080	0,402
3	52,139(s)	1,6192	0,4048
4	63,091(s)	1,3381	0,3345
5	52,135(s)	1,6193	0,4048
6	63,065(s)	1,3387	0,3346

Tabla 6. Tiempo del mejor schedule de cada bucle y su correspondiente SpeedUp y eficiencia

Como podemos observar en los resultados de la tabla, no se aprecian notables diferencias en los resultados de las ejecuciones realizadas con un schedule u otro. Esto último, con la excepción del bucle con id 4, cuyo tiempo con un schedule de tipo dynamic sin tamaño de chunk especificado es extremadamente lento en comparación a cualquier otra configuración.

La justificación de estas anecdóticas diferencias se debe al hecho de que la carga de computación que realiza cada hilo es prácticamente la misma, lo cual vuelve casi indiferente el tipo de schedule usado. Recordemos que las diferencias entre un schedule estático y uno dinámico o guiado se dan cuando los distintos hilos que se usan en un for paralelo tardan tiempos significativamente diferentes. Así pues, como todos los hilos tardan prácticamente el mismo tiempo (debido a que realizan computación prácticamente igual) y debido a la sobrecarga que supone para el computador la gestión de un schedule no estático, podemos determinar que lo mejor sería usar un schedule estático en todos los bucles analizados.

NECESIDAD DE CONSTRUCTORES

Tras haber programado nuestro código para la práctica 0 y habiendo hecho las modificaciones para la práctica 1 no hemos encontrado partes de nuestro código donde

sea necesario usar dichos constructores, mediante barreras incluidas en las propias directivas o simplemente por la organización del código (en suavizar copiamos en una matriz) conseguimos que el código sea paralelizable sin incluir esas directivas extras.

PARALELIZACIÓN CRUZAR CON SECCIONES

```
int random_number = aleatorio(num_pixels);

#pragma omp parallel sections
{
    #pragma omp section
    {
        for (int i = 0; i < random_number; i++)
        {
            hijo1->imagen[i] = padre1->imagen[i];
            hijo2->imagen[i] = padre2->imagen[i];
        }
    }
    #pragma omp section
    {
        for (int i = random_number; i < num_pixels; i++)
        {
            hijo1->imagen[i] = padre2->imagen[i];
            hijo2->imagen[i] = padre1->imagen[i];
        }
    }
}
```

Ilustración 17. Paralelización con sections del cruce

En este caso no podríamos usar la directiva *"nowait"* en *"sections"*, ya que, si lo usáramos pudiéramos obtener resultados no deseados debido a que no esperarían a que todos los hilos acaben, pudiendo ocasionar fallos en posteriores funciones, como "mutar" que hace uso de los nuevos individuos generados, al trabajar con individuos que aún no han sido cruzados.

La única forma de hacerlo sería que separásemos el *"pragma omp parallel*" del *"sections"* y que al incluir la cláusula *"nowait",* el *"parallel"* se encargase de sincronizar a los hilos, pero los resultados obtenidos serían los mismos que los del código de la *ilustración 17*.

COMPARANDO RESULTADOS

Sin paralelizar:

Hilos	Secuencial
1	84,427(s)

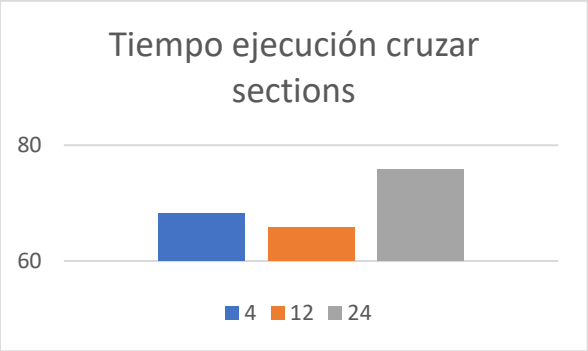
Tabla 7. Tiempo secuencial

Paralelizando:

Hilos	Paralelo	SpeedUp	Eficiencia
4	68,266(s)	1,23673571	0,3091839
12	65.807(s)	1,28294862	0,1069124

24	75,876(s)	1,11269703	0,0463624

Tabla 8. Tiempo ejecución, SpeedUp y Eficiencia Sections



Como podemos observar la paralelización con *"sections"* nos muestra mejoras respecto a la versión secuencial, sin embargo, vemos que los 24 hilos empiezan a empeorar el rendimiento, ya que la sincronización de tantos hilos aumenta el tiempo de ejecución.

En cuanto a los resultados obtenidos pensamos que el mejor resultado debería de ser el de 4 hilos, ya que en la zona *"sections"* solo hay dos *"section",* y por tanto solo trabajan dos hilos en paralelo mientras los demás no hacen nada, sin embargo, los resultados no apoyan nuestra hipótesis, quizás es debido a que los hilos se encontraban ocupados y se tuvo que esperar a la liberación de alguno de ellos.

SUAVIZAR CON COLLAPSE

En este apartado se nos pide paralelizar la función *"suavizar"* con *"collapse"* en caso de ser posible y comentar los resultados obtenidos.

```
#pragma omp for collapse(2)
for(int i=0;i<alto;i++){
    for(int j=0;j<ancho;j++){
        suavizar_pixel(i,j,ancho,alto,imagen_auxiliar, imagen);
    }
}
```

Ilustración 18. Utilización de collapse en suavizar

Lo primero de todo vamos a comentar de que se trata esta clausula
"collapse" permite convertir un bucle anidado en un único bucle, optimizando enormemente la paralelización de este, pero para poder realizar esto se deben cumplir unos requisitos, los cuales son:

1. Que para cada iteración del bucle exterior haya el mismo número de ejecuciones del interior.
2. Que no haya dependencias.

COMPARANDO RESULTADOS

Sin paralelizar:

Hilos	Secuencial
1	84,427(s)

Tabla 9. Tiempo secuencial

Paralelizando:

Hilos	Sin collapse(solo parallel for)	SpeedUp	Eficiencia
12	84,459	0,99962112	0,08330176

Hilos	Con collapse	SpeedUp	Eficiencia
12	84,321	0,98514586	0,08209549

Tabla 10. Tiempo ejecución, SpeedUp y Eficiencia Sections

Como podemos observar no obtenemos ningún tipo de diferencia, esto se puede deber a que el tiempo del suavizar, al realizarse una única vez no es significativo para el tiempo global con respecto al tiempo dedicado en calcular las 2000 generaciones.

MERGESORT

En este ejercicio se nos pide intercambiar el uso de *"qsort"* por *"mergesort"* para la ordenación de la población por fitness y luego paralelizar este código haciendo uso de tareas.

```
void mergeSort(Individuo **poblacion, int izq, int der)
{

    int med = (izq + der) / 2;
    if ((der - izq) < 8) {
        return;
    }

    #pragma omp taskgroup
    {
        #pragma omp task
        {
            mergeSort(poblacion, izq, med);
        }
        #pragma omp task
        {
            mergeSort(poblacion, med, der);
        }
    }

    mezclar(poblacion, izq, med, der);
}
```

Ilustración 19. Función mergeSort en fichero ga.c

```c
void mezclar(Individuo **poblacion, int izq, int med, int der)
{
    int i, j, k;

    Individuo **pob = (Individuo **) malloc((der - izq)*sizeof(Individuo *));
    assert(pob);

    for(i = 0; i < (der - izq); i++) {
        pob[i] = (Individuo *) malloc(sizeof(Individuo));
    }

    k = 0;
    i = izq;
    j = med;
    while( (i < med) && (j < der) ) {
        if (poblacion[i]->fitness < poblacion[j]->fitness) {
            memmove(pob[k],poblacion[i],sizeof(Individuo));
            k++;
            i++;
        }
        else {
            memmove(pob[k],poblacion[j],sizeof(Individuo));
            k++;
            j++;
        }
    }

    for(; i < med; i++) {
        memmove(pob[k++], poblacion[i],sizeof(Individuo));
    }

    for(; j < der; j++) {
        memmove(pob[k++],poblacion[j],sizeof(Individuo));
    }

    i = 0;
    for(i = 0; i < (der - izq); i++) {
        memmove(poblacion[i+izq],pob[i],sizeof(Individuo));
        free(pob[i]);
    }
    free(pob);
}
```

Ilustración 20. Función mezclar en fichero ga.c

Lo primero que vamos a hacer es comentar el código, en el cual tuvimos que coger el código que se nos proporcionó y hacer unos leves cambios.

1. En la función "mezclar" tuvimos que hacer uso de *"memove"* para la copia de los valores de un array a otro, para evitar errores de *"double free"* y *"segmentation fault"*.
2. Sustituir en *"crear_imagen"*, *"qsort"* por *"mergesort"*.

Luego, nuestra tarea fue realizar la paralelización para ello tuvimos que hacer lo siguiente:

- En las llamadas a *"mergesort"* de *"crear_imagen"* lo llamamos desde una región paralela haciendo uso del *"pragma omp parallel"*, y le ponemos un *"pragma omp single"*, ya que solo debe de ejecutarse una vez y no por todos los hilos, como podemos ver en la *ilustracion 21*.

```c
// Ordenar individuos según la función de bondad (menor "fitness" --> más aptos)
#pragma omp parallel
{
    #pragma omp single nowait
    {
        mergeSort(poblacion,0,tam_poblacion);
    }
}
```

Ilustración 21. Llamada a la función mergeSort en crear_imagen en ga.c

- Dentro de *"mergesort"* incluimos como tareas las llamadas recursivas a *"mergesort"* e incluimos ambas tareas en un *"taskgroup"*, de tal forma que se espera a la ejecución de estas tareas para poder llamar a la función "mezclar, como podemos ver en la *ilustración 19.*

COMPARANDO RESULTADOS

Sin paralelizar:

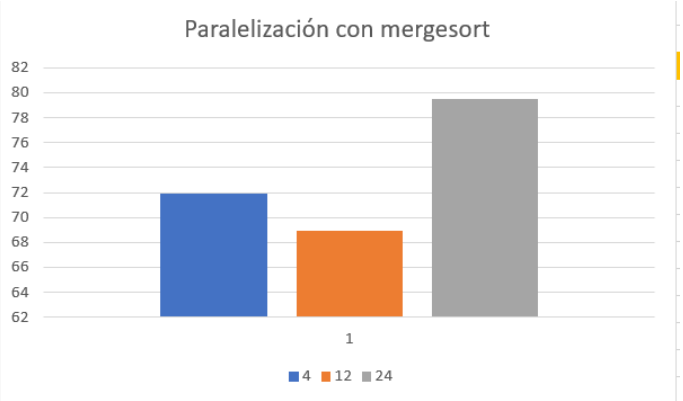
Hilos	Secuencial
1	84,427(s)

Tabla 11. Tiempo secuencial

Paralelizado:

Hilos		SpeedUp	Eficiencia
4	71,927(s)	1,17378731	0,29344683
12	68.890(s)	1,22553346	0,10212779
24	79.458(s)	1,06253618	0,04427234

Tabla 12. Tiempo ejecución, speedUp y eficiencia ,ergesort



buenos, ya que obtenemos una gran reducción del tiempo de ejecución debido al uso de tareas y la estrategia empleada, ya que al haber iniciado una región paralela los hilos están a la espera de que se le asigne trabajo, cosa que hará el propio sistema al añadirse tareas a la cola de tareas a través de las llamadas recursivas con la directiva *"pragma omp task"*.