Metodología de la Programación Paralela

Silvia Perez Ruiz
E-mail: silvia.perezr@um.es, DNI: 49215974E
Raúl Hernández Martínez
E-mail: raul.hernandezm@um.es, DNI: 24423648V
Eduardo Gallego Nicolás
E-mail: eduardo.gallegon@um.es, DNI: 48844607J

Práctica 1. Paralelización con OpenMP

Grado en Ingeniería Informática
Metodología de la Programación Paralela
Profesor: Jose Matias Cutillas Lozano (josematias.cutillas@um.es)

UNIVERSIDAD DE
MURCIA

Índice

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

1 Cuestión 1.

Enunciado: Justificar qué funciones y/o zonas del código son paralelizables. No es necesario indicar las directivas OpenMP que sería necesario utilizar.

Empezamos con el fichero `ga.c`, dentro del código `aplicar_ga`, podemos aplicar en varias partes el paralelismo. La primera en el bucle de inicialización de la población y las demás son los tres bucles que están dentro del bucle que itera las generaciones. El bucle externo no se podría paralelizar ya que dentro de él hay operaciones que afectan a la población completa.

Las funciones `search_element` y `find_element` podríamos decir que son paralelizables, pero considerando que si el bucle es de un tamaño moderado y que una vez que encuentra el elemento parar la ejecución paralela o que garantice encontrar la posición mínima en un entorno paralelo, puede ser mas ineficiente que eficiente, por eso consideramos que no se puede paralelizar estas dos funciones.

Sin embargo la función `cruzar` si que se podría paralelizar ya que los dos bucles se pueden ejecutar en paralelo ya que la i siempre va a ser distinta y no se solaparía nunca la escritura.

Por último, en este fichero tenemos la función `fitness`. Para poder paralelizarlo se podría dividir el individuo en segmentos y calcular parcialmente el valor para cada segmente en paralelo para posteriormente sumarlo y obtener así el total.

En el fichero `io.c`, dentro del código en `generar_matriz_distancias(int n)` se podría paralelizar ya que lo que hay dentro del bucle se calcula de forma independiente y al dividirse el trabajo en partes, la carga de trabajo es claramente divisible entre varios hilos.

2 Cuestión 2.

Enunciado: Con el fin de obtener un código más eficiente en la parte de memoria compartida y poder maximizar el proceso de mejora de la población de individuos: reemplazar las llamadas a la función rand por rand_r (que es "thread safe") para evitar, por un lado, que se genere la misma secuencia de valores aleatorios en diferentes hilos y, por otro, que se pierda paralelismo durante la generación de dichos valores debido a la secuencialidad que introduce la función rand.

Para poder realizar este ejercicio primero hemos buscado en cada parte del código del fichero `ga.c` y hemos sustituido `rand()` por `r_rand()`. A continuación vamos a mostrar en como se quedaría el código cambiado.

La primera función donde hemos tenido que cambiarlo es en la función `int aleatorio(int n`:

```
int aleatorio(int n) {
  return (rand_r(&seed) % n);  // genera un numero aleatorio entre 0 y n-1
}
```

Listing 1: Función aleatorio(int n).

La otra función que hemos tenido que cambiar es la función `void mutar(Individuo *actual, int n, double m_rate)` (vamos a mostrar solo las partes cambiadas):

```
void mutar(Individuo *actual, int n, double m_rate) {
    int i, j;

    // Seleccionar dos posiciones aleatorias (i, j) tal que i < j
    i = rand_r(&seed) % (n - 1) + 1; // i debe ser > 0 y < n
    j = rand_r(&seed) % (n - i) + i + 1;

    // Realizar la mutacion: invertir el orden de los elementos entre i y j
    while (i < j) {

  // Determinar si mutar o no basado en la tasa de mutacion
    double random_value = ((double)rand_r(&seed) / RAND_MAX);

    ...
}
```

Listing 2: Función mutar(...).

Con esto hecho ya tendríamos la primera parte del ejercicio. A continuación nos pide crear una variable global llamada `seed` que se hará `threadprivate` y dicha variable se le pasará como referencia a la función `r_rand()` tal y como se puede ver en los códigos anteriores. Esto se implementaría tal que así en el fichero `ga.c`:

```
unsigned int seed;
#pragma omp threadprivate(seed)
```

Listing 3: Creación de la variable seed.

Por último, es necesario inicializar la variable `seed` y llamar a dicha inicialización justo antes de de generar la población inicial que esto sería en la función `aplicar_ga()`. Lo hacemos así ya que se necesita que todos los hilos estén inicializados y además necesita el mayor número de hilos que hay en el programa ahora mismo. Aclarar también que no hace falta meter la asignación de la variable seed en un `#pragma omp critical` ya que al crearse con `threadprivate` nos garantiza que no se crea una condición de carrera.

```
void inicializar_seed() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        unsigned int thread_seed = (unsigned int)(time(NULL) + thread_id);
        seed = thread_seed;
    }
}

...

double aplicar_ga(const double *d, int n, int n_gen, int tam_pob, int *sol, double m_rate)
{
  int i, g, mutation_start;

  // crea poblacion inicial (array de individuos)
  Individuo **poblacion = (Individuo **) malloc(tam_pob * sizeof(Individuo *));
  assert(poblacion);

  inicializar_seed();

 ...
}
```

Listing 4: Inicializacion de la variable seed.

3 Cuestión 3.

Enunciado: Paralelizar la función fitness utilizando, por un lado, los constructores critical y atomic (de forma independiente) y, por otro, la cláusula reduction, indicando en cada caso qué variables serían privadas y cuáles compartidas. Ejecutar el código variando el número de hilos y comparar los resultados obtenidos, en términos de tiempo de ejecución, respecto a versión secuencial.

Para poder ejecutarlo con varios hilos, hemos puesto un omp_set_num_threads() al principio de la función `aplicar_ga()` para que establezcamos dentro de los parentesis el número de hilos que queramos usar en el código.

Siendo `n_hilos` la cantidad de hilos con la que se va ejecutar el programa. Una vez aclarado esto, ya podemos pasar a comentar las diferentes formas de paralelizar la función fitness.

3.1 Critical.

El código realizado para este caso se nos quedaría así:

```
1
2  void fitness(const double *d, Individuo *individuo, int n)
3  {
4      double t_ini = mseconds();
5
6      double temp_fitness = 0.0;
7
8      // Determina la calidad del individuo calculando la suma de la distancia entre cada par
       de ciudades consecutivas en el array
9
10     // Realiza la suma de todas las ciudades consecutivas almacenado el valor en 'fitness'
11     #pragma omp parallel for
12     for (int i = 0; i < n-1; i++)
13     {
14         double distancia = distancia_ij(d, individuo->array_int[i], individuo->array_int[i +
        1], n);
15         #pragma omp critical
16         temp_fitness += distancia;
17     }
18
19     // Finalmente, se cierra el circulo sumando la distancia de la  ltima  ciudad con la
       ciudad inicial
20     double  distancia_ult = distancia_ij(d, individuo->array_int[n-1], individuo->array_int
       [0], n);
21     temp_fitness += distancia_ult;
22
23     // Se establece el valor fitness calculado al individuo
24     individuo->fitness = temp_fitness;
25
26     double t_fin = mseconds();
27     t_total_time += (t_fin - t_ini) / 1000;
28 }
```

Listing 5: Función fitness con critical.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	0,832 seg	1	1
2500	50	800	0.15	4	8,041 seg	0,1034	0,0258
2500	50	800	0.15	6	11,53 seg	0,0721	0,0120
2500	50	800	0.15	8	13,90 seg	0,0598	0,0074
2500	50	800	0.15	12	16,255 seg	0,0511	0,0042

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	16	18,475 seg	0,0450	0,0028

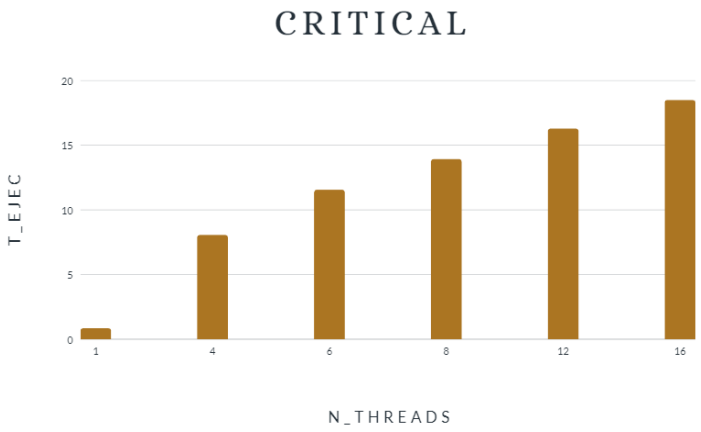

Figura 1: Critical

Como vemos, con la directiva critical no mejora el tiempo con el paralelismo. Esto se puede deber a que como se tiene que hacer esa parte en exclusión mutua, todos los hilos deben esperar a que termine otro hilo de ejecutarla creando asi un embudo que a cuantos más hilos mayor será dicho embudo.

3.2 Atomic.

El código realizado para este caso se nos quedaría así:

```
1
2  void fitness(const double *d, Individuo *individuo, int n)
3  {
4      double t_ini = mseconds();
5
6      double temp_fitness = 0.0;
7
8      // Determina la calidad del individuo calculando la suma de la distancia entre cada par
       de ciudades consecutivas en el array
9
10     // Realiza la suma de todas las ciudades consecutivas almacenado el valor en 'fitness'
11     #pragma omp parallel for
12     for (int i = 0; i < n-1; i++)
13     {
14         double distancia = distancia_ij(d, individuo->array_int[i], individuo->array_int[i +
        1], n);
15         #pragma omp atomic
16         temp_fitness += distancia;
17     }
18
19     // Finalmente, se cierra el circulo sumando la distancia de la  ltima  ciudad con la
       ciudad inicial
20     double  distancia_ult = distancia_ij(d, individuo->array_int[n-1], individuo->array_int
       [0], n);
21     temp_fitness += distancia_ult;
22
23     // Se establece el valor fitness calculado al individuo
24     individuo->fitness = temp_fitness;
25
26     double t_fin = mseconds();
27     t_total_time += (t_fin - t_ini) / 1000;
28 }
```

Listing 6: Función fitness con critical.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	0,556 seg	1	1
2500	50	800	0.15	4	4,614 seg	0,1205	0,0301
2500	50	800	0.15	6	4,549 seg	0,1222	0,0203
2500	50	800	0.15	8	4,627 seg	0,1201	0.0150
2500	50	800	0.15	12	7,754 seg	0,0717	0.0059
2500	50	800	0.15	16	18,475 seg	0,0300	0,0018

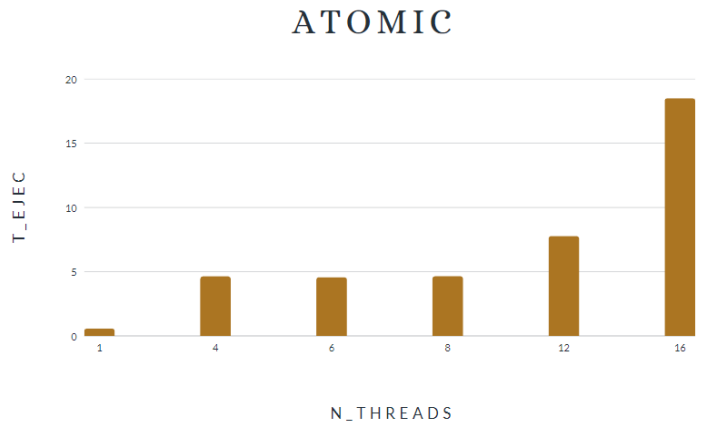

Figura 2: Atomic

Vemos que con esta directiva no mejora debido a que solo un hilo puede acceder a la seccion atomic creando un poco de embudo en esa operacion haciendo que haya grandes esperas viendose asi reflejado en el tiempo de la gráfica y la tabla.

3.3 Reduction.

El código realizado para este caso se quedaría asi:

```
void fitness(const double *d, Individuo *individuo, int n)
{
  double t_ini = mseconds();

    double temp_fitness = 0.0;


  // Se suma la distancia entre todas las ciudades para calcular su valor fitness
  #pragma omp parallel for reduction(+:temp_fitness)
    for(int i = 0; i < n-1; i++) {
    double distancia = distancia_ij(d, individuo->array_int[i], individuo->array_int[i + 1],
     n);
        temp_fitness += distancia;
    }

  // Se realiza a parte la ultima iteracion para unir la ultima ciudad con la primera.
  double distancia_ult = distancia_ij(d, individuo->array_int[n-1], individuo->array_int[0],
     n);
  temp_fitness += distancia_ult;


  // Se guarda el fitness en el fitness del individuo
  individuo->fitness = temp_fitness;

  double t_fin = mseconds();
    t_total_time += (t_fin - t_ini) / 1000;
}
```

Listing 7: Función fitness con reduction.

A continuación pasamos a realizar una tabla comparativa de los distintos de ejecuciones. Solo vamos a tener en cuenta el tiempo que nos ha tardado de la propia función y no con todo el código.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	0,3110 seg	1	1
2500	50	800	0.15	4	0,1280 seg	2,4296	0,6074
2500	50	800	0.15	6	0,1190 seg	2,6134	0,4355
2500	50	800	0.15	8	0,1190 seg	2,6134	0,3266
2500	50	800	0.15	12	0,1140 seg	2,7280	0.2273
2500	50	800	0.15	16	0,7470 seg	0,41633	0.0260

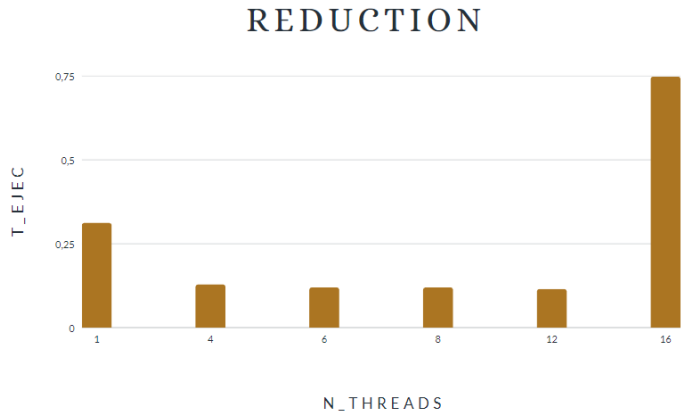

Figura 3: Reduction

Como podemos ver a partir de la tabla y el gráfico, el código con la paralelización ofrece beneficio hasta 6 hilos que es cuando alcanza su punto máximo de speedup. Sin embargo, la eficiencia disminuye con cada incremento de hilos, ya que no se ve una mejora proporcional al aumento de hilos. Aún así vemos que si mejora con el paralelismo con reduction debido a que este permite que que cada hilo trabaje con su propia variable local.

4 Cuestión 4.

Enunciado: Probar diferentes formas de scheduling (static, dynamic, guided) variando el tamaño de asignación (chunk_size) en aquellos bucles en los que se considere necesario. Ejecutar el código variando el número de hilos y justificar los tiempos de ejecución obtenidos con cada política.

Al tener tantas zonas de código que podemos paralelizar, hemos optado por hacerlo en aquellas zonas de código en las que su computación sea mayor, así nos evitamos realizar las pruebas donde la diferencia entre los tiempos de ejecución sean prácticamente insignificantes.

Los bucles que hemos elegido han sido 4. Estos se encuentran todos en la función `aplicar_ga.c` en el fichero `ga.c`. El primer bucle es donde es el de `crear_individuo` y los otros tres son los bucles que llaman a las funciones `cruzar`, `mutar` y `fitness`.

Vamos a probar cada una por separado con cada schedule, con tamaño 0, 8 y 12 y con 1, 4 y 8 hilos.

4.1 Primer bucle.

Para poder medir bien el tiempo vamos a medir solo el tiempo que tarda en ejecutar esa bucle. El código con los pragmas quedaría tal que así:

```
double t_ini = mseconds();

  #pragma omp parallel
  #pragma omp for schedule(TIPO_SCHEDULE, TAM)
  for(i = 0; i < tam_pob; i++) {
    poblacion[i] = (Individuo *) malloc(sizeof(Individuo));
    poblacion[i]->array_int = crear_individuo(n);

    // calcula el fitness del individuo
    fitness(d, poblacion[i], n);
  }

  double t_fin = mseconds();
    t_total_time += (t_fin - t_ini) / 1000;
  printf("Tiempo total for=%.4lf\n", t_total_time);
```

Listing 8: Primer For de aplicar_ga.

Ahora pasamos a analizar cada caso con este bucle.

4.1.1 Static con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,054 seg	1	1
2500	50	800	0.15	4	2,685 seg	3,372	0,843
2500	50	800	0.15	8	1,6050 seg	5,641	0,705

4.1.2 Static con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,4570 seg	1	1
2500	50	800	0.15	4	2,468 seg	3,831	0,957
2500	50	800	0.15	8	1,29 seg	7,331	0,916

4.1.3 Static con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,3550 seg	1	1
2500	50	800	0.15	4	2,4040 seg	3,891	0,972
2500	50	800	0.15	8	1,4860 seg	6,295	0,786

4.1.4 Dynamic con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,372 seg	1	1
2500	50	800	0.15	4	2,426 seg	3,863	0,965
2500	50	800	0.15	8	1,257 seg	7,455	0,931

4.1.5 Dynamic con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,280 seg	1	1
2500	50	800	0.15	4	2,447 seg	3,792	0,948
2500	50	800	0.15	8	1,543 seg	6,014	0,751

4.1.6 Dynamic con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,265 seg	1	1
2500	50	800	0.15	4	2,435 seg	3,804	0,951
2500	50	800	0.15	8	1,337 seg	6,929	0,866

4.1.7 Guided con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,332 seg	1	1
2500	50	800	0.15	4	2,414 seg	3,865	0,966
2500	50	800	0.15	8	1,333 seg	7,000	0,875

4.1.8 Guided con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,2230 seg	1	1
2500	50	800	0.15	4	2,4460 seg	3,770	0,942

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	8	1,572 seg	5,867	0,733

4.1.9 Guided con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,443 seg	1	1
2500	50	800	0.15	4	2,429 seg	3,887	0,971
2500	50	800	0.15	8	1,295 seg	7,291	0,911

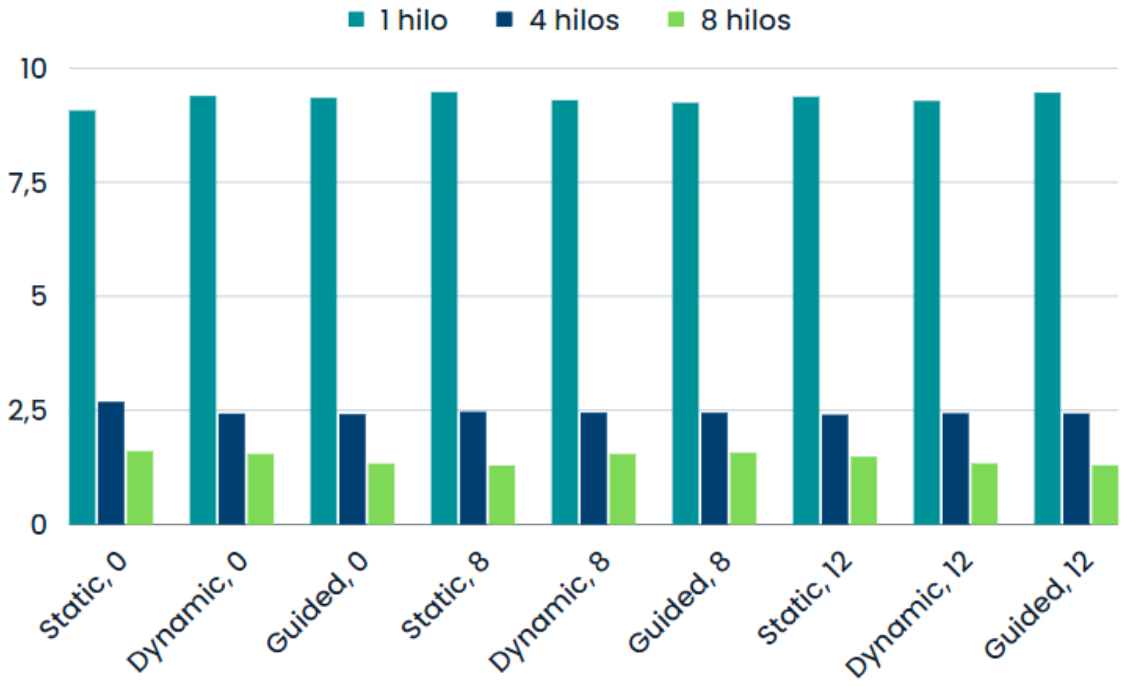

Figura 4: Primer bucle

4.2 Segundo bucle.

Para poder medir bien el tiempo vamos a medir solo el tiempo que tarda en ejecutar ese bucle. El código con los pragmas quedaría tal que así:

```
double t_ini = mseconds();

    #pragma omp parallel
    #pragma omp for schedule(TIPO_SCHEDULE, TAM)
    for(i = 0; i < (tam_pob/2) - 1; i += 2) {
      cruzar(poblacion[i], poblacion[i+1], poblacion[tam_pob/2 + i], poblacion[tam_pob/2 + i
     + 1], n);
    }
    double t_fin = mseconds();
      t_total_time += (t_fin - t_ini) / 1000;
    temp_total += t_total_time
```

Listing 9: Segundo For de aplicar_ga.

4.2.1 Static con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,286 seg	1	1
2500	50	800	0.15	4	9,469 seg	0,980	0,245
2500	50	800	0.15	8	10,106 seg	0,918	0,114

4.2.2 Static con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,233 seg	1	1
2500	50	800	0.15	4	8,9750 seg	1,140	0,285
2500	50	800	0.15	8	8,552 seg	1,196	0,149

4.2.3 Static con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	8,831 seg	1	1
2500	50	800	0.15	4	8,484 seg	1,040	0,260
2500	50	800	0.15	8	8,634 seg	1,022	0,127

4.2.4 Dynamic con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,045 seg	1	1
2500	50	800	0.15	4	8,917 seg	1,014	0,253
2500	50	800	0.15	8	10,024 seg	0,902	0,112

4.2.5 Dynamic con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	7,853 seg	1	1
2500	50	800	0.15	4	8,8940 seg	0,882	0,220
2500	50	800	0.15	8	8,901 seg	0,882	0,110

4.2.6 Dynamic con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	8,547 seg	1	1
2500	50	800	0.15	4	8,667 seg	0,986	0,246
2500	50	800	0.15	8	9,79 seg	0,873	0,109

4.2.7 Guided con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	8,809 seg	1	1
2500	50	800	0.15	4	8,277 seg	1,064	0,266
2500	50	800	0.15	8	9,378 seg	0,939	0,117

4.2.8 Guided con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	8,420 seg	1	1
2500	50	800	0.15	4	8,764 seg	0,960	0,240
2500	50	800	0.15	8	9,636 seg	0,873	0,109

4.2.9 Guided con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	8,994 seg	1	1
2500	50	800	0.15	4	8,9210 seg	1,008	0,252
2500	50	800	0.15	8	10,53 seg	0,854	0,106

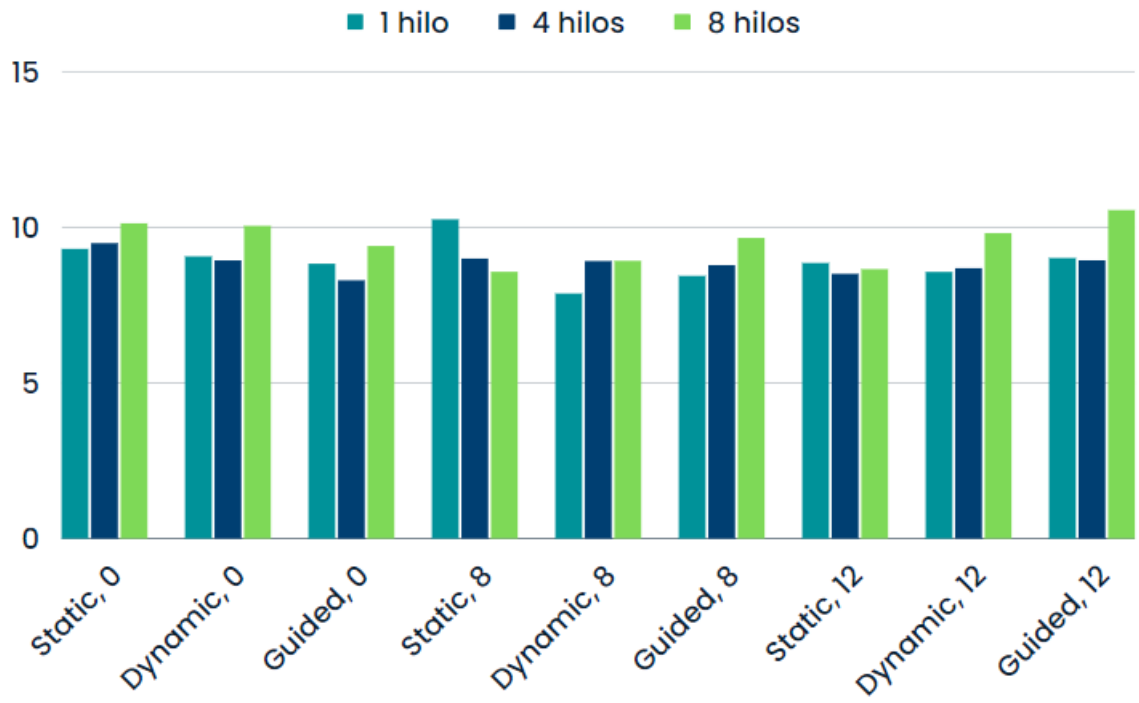

Figura 5: Segundo bucle

4.3 Tercer bucle.

Para poder medir bien el tiempo vamos a medir solo el tiempo que tarda en ejecutar esa bucle. El código con los pragmas quedaría tal que así:

```
double t_ini = mseconds();

    #pragma omp parallel
    #pragma omp for schedule(TIPO_SCHEDULE, TAM)
    for(i = mutation_start; i < tam_pob; i++) {
      mutar(poblacion[i], n, m_rate);
    }
    double t_fin = mseconds();
      t_total_time += (t_fin - t_ini) / 1000;
    temp_total += t_total_time;
```

Listing 10: Tercer For de aplicar_ga.

4.3.1 Static con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,518 seg	1	1
2500	50	800	0.15	4	9,297 seg	1,131	0,282
2500	50	800	0.15	8	10,208 seg	1,030	0,128

4.3.2 Static con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,957 seg	1	1
2500	50	800	0.15	4	9,497 seg	1,048	0,262
2500	50	800	0.15	8	9,8090 seg	1,015	0,126

4.3.3 Static con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,595 seg	1	1
2500	50	800	0.15	4	10,337 seg	0,928	0,232
2500	50	800	0.15	8	11,037 seg	0,8696	0,108

4.3.4 Dynamic con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,165 seg	1	1
2500	50	800	0.15	4	8,746 seg	1,162	0,290
2500	50	800	0.15	8	10,492 seg	0,968	0,121

4.3.5 Dynamic con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,106 seg	1	1
2500	50	800	0.15	4	10,197 seg	0,991	0,247
2500	50	800	0.15	8	10,847 seg	0,931	0,116

4.3.6 Dynamic con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,485 seg	1	1
2500	50	800	0.15	4	8,9370 seg	1,173	0,293
2500	50	800	0.15	8	10,567 seg	0,992	0,124

4.3.7 Guided con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	10,011 seg	1	1
2500	50	800	0.15	4	10,345 seg	0,967	0,241
2500	50	800	0.15	8	10,899 seg	0,918	0,114

4.3.8 Guided con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,686 seg	1	1
2500	50	800	0.15	4	8,962 seg	1,080	0,270
2500	50	800	0.15	8	9,092 seg	1,065	0,133

4.3.9 Guided con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,765 seg	1	1
2500	50	800	0.15	4	10,2180 seg	0,955	0,238
2500	50	800	0.15	8	11,2740 seg	0,866	0,108

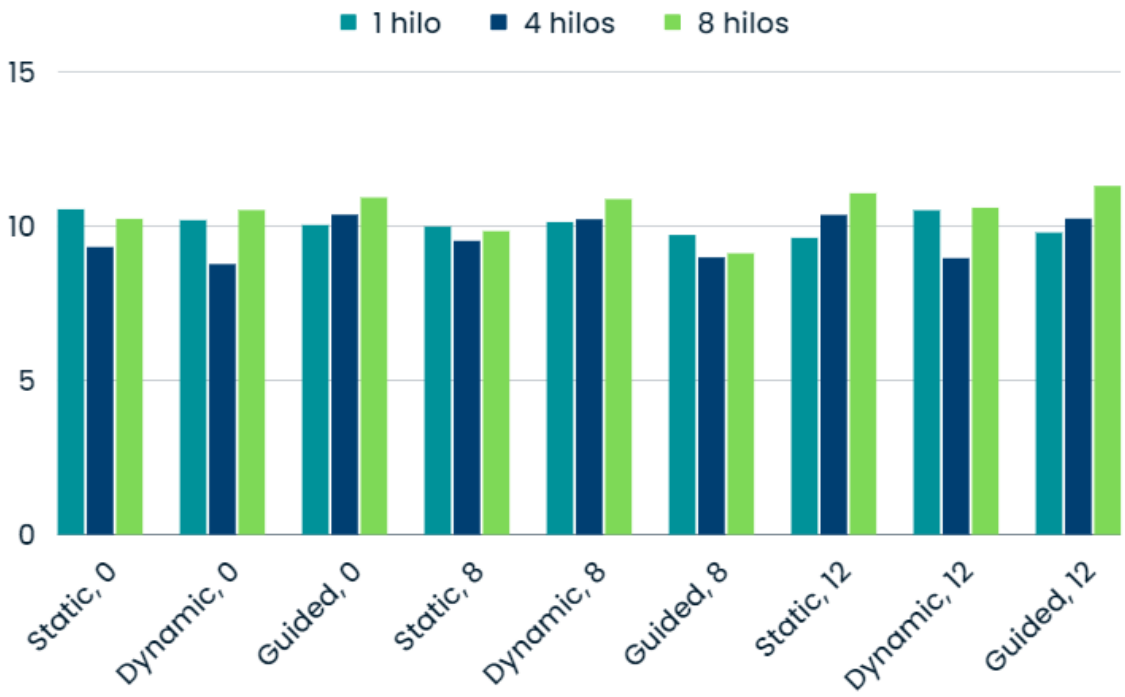

Figura 6: Tercer bucle

4.4 Cuarto bucle.

Para poder medir bien el tiempo vamos a medir solo el tiempo que tarda en ejecutar esa bucle. El código con los pragmas quedaría tal que así:

```
double t_ini = mseconds();

    #pragma omp parallel
    #pragma omp for schedule(TIPO_SCHEDULE, TAM)
    for(i = 0; i < tam_pob; i++) {
      fitness(d, poblacion[i], n);
    }
    double t_fin = mseconds();
      t_total_time += (t_fin - t_ini) / 1000;
    temp_total += t_total_time;
```

Listing 11: Cuarto For de aplicar_ga.

4.4.1 Static con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,891 seg	1	1
2500	50	800	0.15	4	9,632 seg	1,338	0,334
2500	50	800	0.15	8	8,670 seg	1,486	0,185

4.4.2 Static con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,136 seg	1	1
2500	50	800	0.15	4	9,359 seg	1,296	0,324
2500	50	800	0.15	8	10,567 seg	1,148	0,143

4.4.3 Static con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,840 seg	1	1
2500	50	800	0.15	4	8,8060 seg	1,458	0,364
2500	50	800	0.15	8	9,527 seg	1,347	0,168

4.4.4 Dynamic con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,385 seg	1	1
2500	50	800	0.15	4	10,015 seg	1,236	0,309
2500	50	800	0.15	8	10,373 seg	1,193	0,149

4.4.5 Dynamic con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,849 seg	1	1
2500	50	800	0.15	4	10,018 seg	1,282	0,320
2500	50	800	0.15	8	10,934 seg	1,175	0,146

4.4.6 Dynamic con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	13,3640 seg	1	1
2500	50	800	0.15	4	9,644 seg	1,385	0,346
2500	50	800	0.15	8	9,800 seg	1,363	0,170

4.4.7 Guided con tamaño 0.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	13,305 seg	1	1
2500	50	800	0.15	4	10,465 seg	1,271	0,317
2500	50	800	0.15	8	8,902 seg	1,494	0,1186

4.4.8 Guided con tamaño 8.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	13,154 seg	1	1
2500	50	800	0.15	4	11,036 seg	1,191	0,297
2500	50	800	0.15	8	10,481 seg	1,255	0,156

4.4.9 Guided con tamaño 12.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	12,705 seg	1	1
2500	50	800	0.15	4	8,778 seg	1,447	0,361
2500	50	800	0.15	8	9,953 seg	1,276	0,159

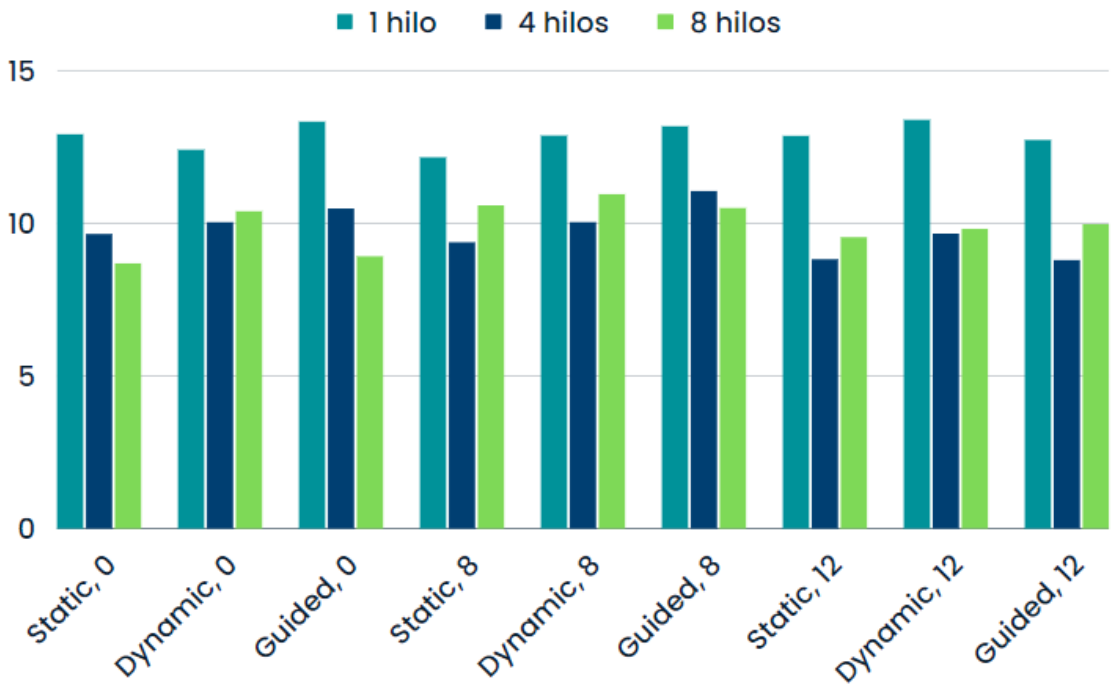

Figura 7: Cuarto bucle

4.5 Conclusiones.

En el primer bucle podemos observar que los tiempos son muy similares con la misma cantidad de hilos y que apenas varia añadiendo más tamaño. Esto puede sugerir que la iteración de cada bucle es similar y que los hilos se reparten de manera eficiente el trabajo haciendo que no haya diferencias entre cada tipo de schedule. Lo que si que podemos observar es que cuando añadimos más tamaño el tiempo mejora considerablemente en los tres schedule.

En el segundo bucle podemos ver que con el schedule de guided nunca mejora con ningún tamaño y ningún hilo, de hecho es simpre mejor la secuencial. Con dynamic y con static podemos ver que aunque con algunos tamaños si mejora, en general no se ve una mejora significativa.

En el tercer bucle parece que siempre mejora en todos los casos cuando los hilos son igual a 4 pero la diferencia es significativa y puede ser que cuantos más no mejore debido a que el costo de su inicialización sea mayor que el tiempo que tarda en si en ejecutarse el bucle.

En el cuarto bucle podemos ver que el mejor caso en con el schedule guided ya que siempre mejora el tiempo. Aunque esto también pasa con los otros dos, no siempre lo mejoran y se produce mas tiempo cuantos mas hilos hay. En estos dos ulitmos hilos lo más eficiente es coger la ejecución con 4 hilos.

5 Cuestión 5.

Enunciado: Indicar si es necesario utilizar de forma explícita en alguna zona del código los siguientes constructores: barrier, single, master, ordered. Justificar la decisión tomada en el caso de que así sea.

Ninguno de los constructores es necesario ya que nunca se indicará que haya paralelización en esa parte del código.

Barrier:

En el bucle principal de la función aplicar_ga, puedes utilizar una barrera para asegurarte de que todos los hilos finalicen una iteración antes de continuar con la siguiente iteración del bucle.

Single y Master:

No parece haber una sección crítica que requiera que solo un hilo ejecute un bloque de código específico. Por lo tanto, no parece necesario utilizar los constructores single o master en este código.

Ordered:

No hay bucles que necesiten un orden específico en el código proporcionado, por lo que el uso del constructor ordered no parece necesario.

6 Cuestión 6.

Enunciado: Paralelizar la función cruzar utilizando secciones, explicando por qué el código secuencial implementado puede paralelizarse con OpenMP sin que se produzcan condiciones de carrera en los accesos concurrentes a memoria por parte de los hilos. ¿Se podría usar la cláusula nowait? Ejecutar el código variando el número de hilos y justificar el tiempo de ejecución obtenido.

Podría causar condiciones de carrera si se usa la cláusula nowait, ya que se utiliza para indicar que un hilo no espera a los otros hilos antes de continuar ejecutándose. El bucle de la segunda mitad depende de los valores que se copian en la primera mitad, y si los hilos no esperan, los resultados serán impredecibles y probablemente incorrectos.

```
void cruzar(Individuo *padre1, Individuo *padre2, Individuo *hijo1, Individuo *hijo2, int n)
{
  // Elegir un punto (o puntos) de corte aleato   rio a partir del que se realiza el
    intercambio de los genes.
  int puntoCorteAleatorio = aleatorio(n-1);

  // Entonces, por ejemplo, los primeros genes del padre1 van al hijo1, y los primeros del
    padre2 al hijo2.
        // Se debe evitar en cada paso la introduccion de duplicados en los hijos
  // Y los restantes genes de cada hijo son del otro padre, respectivamente.
  #pragma omp parallel sections shared(padre1, padre2, hijo1, hijo2, puntoCorteAleatorio, n)
  {
    #pragma omp section
    {
      for(int i = 0 ; i<puntoCorteAleatorio ; i++){
        hijo1->array_int[i] = padre1->array_int[i];
        hijo2->array_int[i] = padre2->array_int[i];
      }
    }

    #pragma omp section
    {
      for(int i = puntoCorteAleatorio ; i<n ; i++){
        hijo1->array_int[i] = padre2->array_int[i];
        hijo2->array_int[i] = padre1->array_int[i];
      }
    }

  }

}
```

Listing 12: Función cruzar.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	6,652 seg	1	1
2500	50	800	0.15	2	6,359 seg	1,046	0,52
2500	50	800	0.15	4	7,234 seg	0,919	0,22
2500	50	800	0.15	8	7,158 seg	0.929	0,116
2500	50	800	0.15	12	8,562 seg	0,776	0,064
2500	50	800	0.15	16	9,505 seg	0,699	0,043

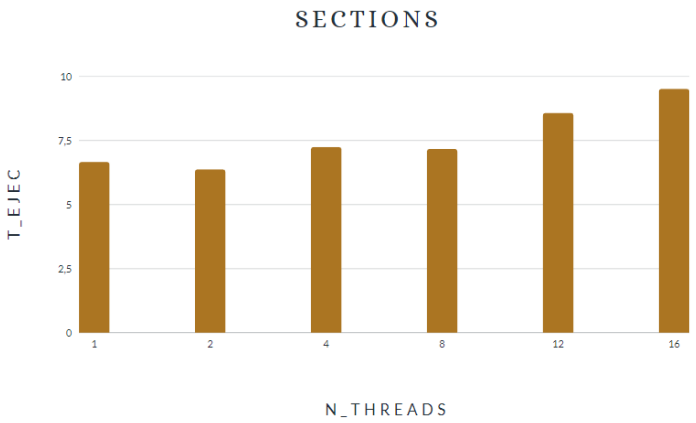

Figura 8: Sections

Dado que la tarea de cruzar genes y asignarlos a la descendencia es relativamente pequeña, la ejecución secuencial puede ser más eficiente en términos de tiempo total debido a la ausencia de gastos generales de paralelización. Los subprocesos compiten por recursos compartidos como la memoria y el caché. Además, la creación y sincronización de subprocesos puede generar una sobrecarga adicional. En este caso, el tamaño de la tarea es pequeño, por lo que los beneficios de la paralelización pueden no compensar los costos adicionales asociados con la administración de múltiples subprocesos, lo que resulta en un mayor tiempo de ejecución. Usar solo dos subprocesos es menos costoso de administrar que una mayor cantidad de subprocesos, y dividir el trabajo entre los dos subprocesos puede mejorar ligeramente el rendimiento. Sin embargo, el beneficio es mínimo debido a la pequeña cantidad de trabajo realizado en cada hilo.

7 Cuestión 7.

Enunciado: Reemplazar las llamadas a la función qsort por la función mergeSort que se muestra. A continuación, paralelizar dicha rutina mediante el uso de tareas (OpenMP tasks) y ejecutar el código variando el número de hilos. ¿Se reduce el tiempo de ejecución? Si no es así, explica las posibles causas. La función mezclar a la que invoca se proporciona en el fichero mezclar.c

```
void mergeSort(Individuo **poblacion, int izq, int der)
{
    int med = (izq + der) / 2;
    if ((der - izq) < 2) {
        return;
    }

    #pragma omp task
    mergeSort(poblacion, izq, med);

    #pragma omp task
    mergeSort(poblacion, med, der);

    #pragma omp taskwait
    mezclar(poblacion, izq, med, der);
}
```

Listing 13: Función mergeSort.

```
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) {
      // copiar poblacion[i++] en pob[k++]
      memmove(pob[k++], poblacion[i++], sizeof(Individuo));
    }
    else {
      // copiar poblacion[j++] en pob[k++]
      memmove(pob[k++], poblacion[j++], sizeof(Individuo));
    }
  }

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

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

  i = 0;
  while(i < (der - izq)) {
    // copiar pob[i] en poblacion[i + izq]
    memmove(poblacion[i+izq], pob[i], sizeof(Individuo));
    // liberar individuo 'i'
    free(pob[i]);
    i++;
  }
  free(pob);
```

```
45 }
```

Listing 14: Función mezclar.

N	N_GEN	TAM_POB	M_RATE	N_THREADS	T_EJEC	SPEEDUP	EFICIENCIA
2500	50	800	0.15	1	9,39 seg	1	1
2500	50	800	0.15	2	9,30 seg	1,0092	0,504
2500	50	800	0.15	4	9,46 seg	0,992	0,248
2500	50	800	0.15	8	9,43 seg	0.995	0,124
2500	50	800	0.15	12	9,43 seg	0,995	0,082
2500	50	800	0.15	16	9,54 seg	0,984	0,061

Como se puede observar, el aumento en el número de hilos no mejora el rendimiento, al contrario, empeora mínimamente. Esta situación probablemente se debe a que la carga de trabajo es demasiado pequeña para aprovechar la paralelización proporcionada por las tareas. Este fenómeno se presenta a pesar de que la función mergeSort es recursiva y utiliza bucles while en la función mezclar. La falta de mejoras en el rendimiento es evidente a pesar de la naturaleza recursiva de mergeSort y de los bucles while en mezclar.

8 Cuestión 8.

Enunciado: Realizar el informe final incluyendo las conclusiones generales y valoración personal sobre el trabajo realizado

Este proyecto nos ha dado la oportunidad de aprender cómo aplicar la paralelización con OpenMP en un contexto que nosotros mismos pomos hacer. Nos ha permitido entender las virtudes de OpenMP, pero también nos ha enseñado que la decisión de cuándo y cómo paralelizar no siempre es evidente, ya que puede ser complicado elegir el enfoque adecuado.

Una de las mayores ventajas de esta práctica ha sido consolidar los conocimientos teóricos sobre la paralelización y tomar decisiones sobre cuándo y cómo implementar la paralelización de manera efectiva.

En cambio, es frustrante hacer la práctica 1 utilizando el mismo código de la práctica 0 que aún no ha sido corregida. Esto nos genera preocupación acerca de nuestras soluciones, ya que los errores no corregidos del código anterior podrían afectar negativamente a esta práctica. Estaría bien contar con un código base corregido para la práctica 1.