

Práctica 1

Metodología de la Programación Paralela
Curso 2024/25

Paralelización con OpenMP



Miembros del Grupo:

Miguel Saez Martinez miguel.saezm@um.es

Juan Hernández Acosta juan.hernandez@um.es

Índice

Cuestión 1.....	3
Creación de la población inicial.....	3
Evaluación del fitness de la población.....	3
Cálculo interno de la función fitness.....	3
Cruce.....	4
Mutación.....	4
Cuestión 2.....	5
Cambios principales en el código.....	5
Comparación de resultados.....	7
Cuestión 3.....	8
Critical.....	8
Atomic.....	8
Reduction.....	9
Cuestión 4.....	11
Bucle que genera la población inicial.....	11
Bucle de cruce.....	11
Bucle de mutación.....	12
Bucle de recálculo de fitness.....	12
Análisis de los resultados.....	13
Cuestión 5.....	14
Cuestión 6.....	16
Cuestión 7.....	17
Barrier.....	17
Single.....	17
Master.....	17
Ordered.....	17
Cuestión 9.....	18
Cambios principales en el código.....	18
Análisis de los resultados.....	18

Cuestión 1.

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

Cuando se aborda la paralelización de un algoritmo, el objetivo principal es encontrar las secciones de mayor coste computacional y aquellas que puedan beneficiarse del paralelismo sin añadir una sobrecarga excesiva. En el caso de nuestro código de algoritmo genético, las partes más costosas y más claramente independientes (crear individuos, calcular *fitness*, cruzar, mutar) se organizan en bucles que iteran sobre grandes estructuras, en este caso la población. Por ello, paralelizar a nivel de esos bucles reporta beneficios claros en tiempo de ejecución

Creación de la población inicial

La generación de cada individuo y el cálculo de su *fitness* pueden distribuirse en paralelo porque las iteraciones del bucle que recorre la población no dependen entre sí. Cada posición de la población se completa de manera independiente, sin necesitar información de los demás. Además, si el tamaño de la población es grande, esto puede suponer un incremento significativo en el rendimiento al aprovechar varios hilos. Sin embargo, si la población no es suficientemente numerosa, podría ocurrir que la sobrecarga de crear y gestionar los hilos resulte mayor que la mejora obtenida. También hay que tener precaución con la generación de números aleatorios, ya que llamar a la misma función `rand()` desde varios hilos puede crear condiciones de carrera y resultados no reproducibles.

Evaluación del *fitness* de la población

Después de crear o modificar los individuos, se recalcula su *fitness*, y este proceso también se presta bien al paralelismo. Cada individuo sólo necesita leer su propia información genética y acceder en modo lectura al array de distancias, sin necesidad de conocer nada sobre los demás individuos. Por eso, cada iteración es completamente independiente. El beneficio de paralelizar este bucle se hace aún más evidente cuando la población crece en tamaño o cuando el número de genes (*m*) es elevado, ya que el propio cálculo de la función de *fitness* puede resultar costoso. El coste de sobrecarga es, de nuevo, el de gestionar hilos y asegurar la correcta sincronización, pero a menudo vale la pena porque este recálculo se realiza en cada generación del algoritmo.

Cálculo interno de la función *fitness*

Dentro de la propia función *fitness*, se recorre cada par de genes para acumular la distancia total. Este doble bucle puede alcanzar un tamaño de orden m^2 , lo que supone un cómputo notable si el subconjunto de genes es grande. En este caso, podría ser útil incluir directivas de paralelización usando una reducción, de modo que cada hilo acumule sus sumas parciales sin pisar las de los demás. Si *m* es pequeño, la ganancia no siempre justifica el coste de arrancar varios hilos: en esas circunstancias, es habitual paralelizar en un nivel superior (por ejemplo, a escala de la población) y dejar la función *fitness* sin paralelizar internamente.

Cruce

La operación de cruce, en la que se toman pares de padres para generar dos hijos, se puede ejecutar en paralelo siempre que cada par trabaje sobre sus propios índices y no se produzcan solapamientos en la escritura. Al leer los padres y escribir los hijos en posiciones bien diferenciadas de la población, no se crean dependencias entre iteraciones. Si el volumen de la población es amplio y se están formando muchos hijos, puede notarse una aceleración considerable.

Mutación

Cuando mutamos a un individuo concreto, no influimos en absoluto en los demás, ya que la mutación se limita a modificar ciertos genes dentro del mismo individuo. Eso hace que cada iteración que recorre la parte de la población a mutar sea independiente y, por ende, paralelizable. El ahorro de tiempo dependerá de la proporción de la población que se somete a mutación y de la complejidad del proceso de mutación. Aunque no suele ser tan caro como el cálculo de fitness, si el porcentaje de mutación es elevado y la población es grande, la paralelización también puede mejorar el rendimiento. En cualquier caso, sigue siendo necesaria la cautela con la generación de números aleatorios para evitar problemas de concurrencia.

Las otras funciones no se recomiendan paralelizar porque el coste de coordinarlas en paralelo supera con creces el beneficio que se obtendría, y/o porque requieren un rediseño complejo.

Cuestión 2.

Con el fin de obtener un código más eficiente en la parte de memoria compartida y poder optimizar 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`. Además, añadir el código necesario para implementar correctamente la generación aleatoria a lo largo del algoritmo.

Para ello, una posibilidad sería crear en `mh.c` una variable global `seed` que se hará `threadprivate` y que se pasará como argumento por referencia a `rand_r`. Así, además, cada hilo podrá inicializar `seed` en paralelo de forma independiente (por ejemplo, en función del tiempo del sistema, de su `pid` o una combinación de ambos) en una primera región paralela situada justo antes de generar la población inicial de subconjuntos B del problema MDP (bucle `for` que crea cada individuo, que identificamos como `FOR_INI`). Es en esta generación aleatoria de individuos donde el efecto sobre el paralelismo de `rand_r` en lugar de `rand` es claramente apreciable.

Lanzar ejecuciones y tomar tiempos del bucle `FOR_INI`, representando en una misma gráfica dos series de tiempos variando el número de hilos: serie (a) usando `rand`, y serie (b) usando `rand_r` y las modificaciones del código previamente indicadas. Analizar los resultados obtenidos.

Cambios principales en el código

Para mejorar la generación de números aleatorios y hacerla más adecuada al entorno paralelizado, se ha introducido una semilla local por hilo. Concretamente, se declara una variable global `seed` marcada como `threadprivate`, de forma que cada hilo conserva su propia copia.

```
unsigned int seed;

#pragma omp threadprivate(seed)
```

Dentro de la función que devuelve valores aleatorios, se sustituye la llamada a `rand()` por `rand_r(&seed)`, de modo que cada hilo maneja su propio estado del generador.

```
int aleatorio(int n)
{
    // Genera y devuelve un número aleatorio entre 0 y n-1.

    return (rand_r(&seed) % n);
}
```

Además, antes de entrar en los bucles paralelos (por ejemplo, el que crea la población inicial), cada hilo inicializa su semilla en función de la hora actual y del identificador de hilo (`omp_get_thread_num()`).

```
#pragma omp parallel private(i)

{

    // Inicializa la semilla para cada hilo

    seed = (unsigned int)time(NULL) + omp_get_thread_num();
```

Por último, se añaden directivas de OpenMP en diferentes zonas del algoritmo genético para distribuir las tareas entre varios hilos y así aprovechar los núcleos disponibles.

```
// Paraleliza la generación de la población inicial

#pragma omp parallel private(i)

{

    // Inicializa la semilla para cada hilo

    seed = (unsigned int)time(NULL) + omp_get_thread_num();


#pragma omp for schedule(static)

    for (i = 0; i < tam_pob; i++)

    {

        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));

        poblacion[i]->array_int = crear_individuo(n, m);


        // Calcula el fitness del individuo.

        fitness(d, poblacion[i], n, m);

    }

}

// Recalcula el fitness de todos los individuos después de cruce y mutación.

#pragma omp parallel for private(i) schedule(static)

    for (i = 0; i < tam_pob; i++)
```

```
{  
  
    fitness(d, poblacion[i], n, m);  
  
}
```

Comparación de resultados

Al utilizar varios hilos, se observan reducciones notables en los tiempos de ejecución en aquellas partes del código que hacen un uso intensivo de números aleatorios, como la generación de la población inicial y la mutación. En problemas grandes, se han registrado aceleraciones (speed-up) que pueden superar el factor de 2 o 3, dependiendo del número de hilos y del tamaño de la instancia. Por ejemplo, para $n=1600$, $m=600$ y $tam_pob=700$, la ejecución con 2 hilos reduce el tiempo desde más de 100 segundos en secuencial a aproximadamente 36.19 segundos, logrando un speed-up cercano a 2.94. Con un mayor número de hilos, el speed-up continúa aumentando, aunque la eficiencia puede descender debido a la sobrecarga de coordinación entre hilos.

La fiabilidad también mejora al utilizar `rand_r()`, ya que cada hilo genera valores independientes y se evitan conflictos en el acceso a un estado global compartido. En conjunto, los resultados muestran que, cuando el problema crece en tamaño y se dispone de suficientes hilos, la versión con `rand_r()` y las directivas de OpenMP reduce de forma considerable el tiempo de ejecución comparado con la versión secuencial original.

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej2_randr** o puede acceder directamente usando este enlace:

[x Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#)

Cuestión 3.

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 respecto a versión secuencial utilizando las métricas de rendimiento.

Justificar de forma teórica las diferencias y optimizar, en lo posible, el código de critical y atomic teniendo en cuenta cómo está implementada internamente reduction.

Critical

```
#pragma omp parallel private(i, j) shared(d, individuo, n, m, elements)
{
    double thread_sum = 0.0; // Acumulador privado del hilo

#pragma omp for schedule(static)
    for (i = 0; i < m - 1; i++){
        for (j = i + 1; j < m; j++){
            thread_sum += distancia_ij(d, elements[i], elements[j], n);
        }
    }

    // Una única operación crítica por hilo al final
#pragma omp critical
    {
        sum += thread_sum;
    }

    individuo->fitness = sum;
}
```

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej3_critical** o puede acceder directamente usando este enlace:

[x Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#) .

Atomic

```
void fitness(const double *d, Individuo *individuo, int n, int m)
{
    int *elements = individuo->array_int;
    double sum = 0.0;
    int i, j;

#pragma omp parallel private(i, j) shared(d, individuo, n, m, elements)
    {
        double thread_sum = 0.0;

#pragma omp for schedule(static)
        for (i = 0; i < m - 1; i++)
        {
            for (j = i + 1; j < m; j++)
            {
                thread_sum += distancia_ij(d, elements[i], elements[j], n);
            }
        }

        // Actualizamos sum una sola vez por hilo, de forma atómica
#pragma omp atomic
        sum += thread_sum;
    }
}
```



```
    individuo->fitness = sum;
}
```

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej3_atomic** o puede acceder directamente usando este enlace:

[X Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#) .

Reduction

```
void fitness(const double *d, Individuo *individuo, int n, int m)
{
    int *elements = individuo->array_int;

    double sum = 0.0;

    int i, j;

#pragma omp parallel for private(j) shared(d, individuo, n, m, elements) reduction(+ : sum)
    schedule(static)

    for (i = 0; i < m - 1; i++){

        for (j = i + 1; j < m; j++){

            sum += distancia_ij(d, elements[i], elements[j], n);

        }

    }

    individuo->fitness = sum;
}
```

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej3_reduction** o puede acceder directamente usando este enlace:

[X Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#) .

En las distintas hojas de resultados podemos centrarnos en los valores de exe_time, speed-up y eficiencia para comparar los desempeños de las distintas formas de paralelizar.

En cuanto al tiempo de ejecución como es de esperar, 1 solo hilo es similar entre todos los métodos y conforme se aumentan los hilos reduction muestra tiempos de ejecución más consistentes.

En el speed-up reduction destaca a partir de los 4 hilos y critical muestra el decrecimiento más pronunciado, probablemente debido al coste de los bloqueos. Atomic se comporta mejor que critical pero no iguala a reduction.

Finalmente respecto a la eficiencia `reduction` sigue siendo el que muestra resultados más altos.

Con respecto a las variables, utilizando **critical**, las variables privadas son `i`, `j` y `thread_sum`, ya que cada hilo tiene su propio acumulador y sus índices para los bucles. Las variables compartidas son `d`, `individuo`, `n`, `m` y `elements`, ya que contienen datos comunes que todos los hilos deben acceder para realizar los cálculos. En el código con **atomic**, las variables privadas son también `i`, `j` y `thread_sum`, porque cada hilo tiene un acumulador local que se suma de manera atómica a la variable compartida `sum`. Las variables compartidas son `d`, `individuo`, `n`, `m` y `elements`, porque todos los hilos necesitan acceder a estos datos para calcular las distancias. Por último, en el código con **reduction**, las variables privadas son `i` y `j`, ya que cada hilo utiliza índices locales en los bucles, mientras que las variables compartidas son `d`, `individuo`, `n`, `m` y `elements`. La variable `sum` se utiliza como una reducción, lo que significa que cada hilo tiene su copia local que se combina al final de forma eficiente.

Estos resultados se alinean con la teoría que nos indica que `critical` es el más simple de implementar pero menos eficiente al escalar. `Atomic` es más eficiente que `critical` pero tampoco presenta una buena escalabilidad debido a las actualizaciones concurrentes. `Reduction` reduce el número de conflictos en memoria compartida y es por tanto el que tiene la mejor escalabilidad así mostrándolo conforme aumentamos el número de hilos.

Cuestión 4.

Probar diferentes formas de reparto de trabajo, 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 y rendimiento obtenidos con cada política.

Los bucles que se han paralelizado son los que, como se ha explicado anteriormente, esperan reportar más beneficio asociado a la paralelización por trabajar con grandes fragmentos de información. En este caso, los siguientes bucles internos del algoritmo genético:

Bucle que genera la población inicial

```
#pragma omp for schedule(política, chunk_size)

    for (i = 0; i < tam_pob; i++)

    {

        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));

        poblacion[i]->array_int = crear_individuo(n, m);

        fitness(d, poblacion[i], n, m);

    }

}
```

Bucle de cruce

```
#pragma omp parallel private(i)

{

#pragma omp for schedule(política, chunk_size)

    for (i = 0; i < (tam_pob / 2) - 1; i += 2)

    {

        if (poblacion[tam_pob / 2 + i]->array_int == NULL)

            poblacion[tam_pob / 2 + i]->array_int = (int *)malloc(m * sizeof(int));

        if (poblacion[tam_pob / 2 + i + 1]->array_int == NULL)

            poblacion[tam_pob / 2 + i + 1]->array_int = (int *)malloc(m *

sizeof(int));
```

```

        cruzar(poblacion[i], poblacion[i + 1],

               poblacion[tam_pob / 2 + i], poblacion[tam_pob / 2 + i + 1],

               n, m);
    }
}

```

Bucle de mutación

```

mutation_start = tam_pob / 4;

#pragma omp parallel for private(i) schedule(política, chunk_size)

    for (i = mutation_start; i < tam_pob; i++)
    {
        seed = (unsigned int)time(NULL) + omp_get_thread_num() + i;

        mutar(poblacion[i], n, m);
    }

```

Bucle de recálculo de *fitness*

```

#pragma omp parallel for private(i) schedule(política, chunk_size)

    for (i = 0; i < tam_pob; i++)
    {
        fitness(d, poblacion[i], n, m);
    }

```

Se han elegido tamaños de chunk de 4, 8, 16 y 32 para evaluar cómo influyen en el rendimiento del programa. Este rango cubre distintos equilibrios entre, por un lado, la sobrecarga de coordinar bloques de iteraciones muy pequeños (lo que aumenta el coste de *scheduling*), y por otro, el posible desbalanceo al asignar bloques demasiado grandes. Además, la elección de estos valores permite estudiar el impacto en el tiempo de ejecución para diferentes tamaños de entrada y niveles de complejidad, determinando así el *trade-off* entre rendimiento y sobrecarga en la paralelización estática.

Análisis de los resultados

Los resultados obtenidos para las distintas políticas se encuentran en los siguientes enlaces:

-Static: [x Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#)

-Dynamic: [x Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#)

-Guided: [x Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx](#)

Como se aprecia en los resultados, cualquiera de las tres políticas de *scheduling* (static, dynamic o guided) aporta beneficios notables en la paralelización de los bucles. Sin embargo, aunque en la mayoría de los casos el *speed-up* aumenta a medida que incrementamos el número de hilos, la eficiencia sólo mejora de manera significativa en los experimentos con mayores tamaños de entrada. Una posible explicación es que, para problemas pequeños, la sobrecarga de coordinar y comunicar entre varios hilos puede exceder la ganancia que se obtiene al paralelizar, mientras que en problemas más grandes la carga de trabajo es suficiente para justificar esa sobrecarga y aprovechar al máximo los recursos del sistema.

Entre las tres políticas, la que ofrece mejores resultados parece ser la variante *guided*, donde se observa que la eficiencia puede incluso llegar a superar el valor de 1 en muchos de los casos de prueba. Esto sugiere que, al ir asignando bloques de iteraciones cada vez más pequeños según avanza el cálculo, la carga de trabajo se reparte de forma muy equilibrada. Además, puede suceder que algunos efectos de optimización (como un mejor uso de la caché o un acceso más regular a la memoria) contribuyan a que el tiempo de ejecución paralelo llegue a ser menor que el de la versión secuencial, dando lugar a esas eficiencias superiores a 1.

Cuestión 5.

Paralelizar la función cruzar utilizando secciones. ¿Se podría usar la cláusula `nowait`? (argumentar la respuesta). Ejecutar el código variando el número de hilos y justificar el rendimiento obtenido.

```
void cruzar(Individuo *padre1, Individuo *padre2, Individuo *hijo1, Individuo *hijo2, int
n, int m){

    // Selecciona un punto de cruce aleatorio entre 1 y m - 1.

    int punto = aleatorio(m - 1) + 1;

    // Asegura que los arrays de los hijos están asignados y inicializados.

    if (hijo1->array_int == NULL)

        hijo1->array_int = (int *)malloc(m * sizeof(int));

    if (hijo2->array_int == NULL)

        hijo2->array_int = (int *)malloc(m * sizeof(int));

    memset(hijo1->array_int, -1, m * sizeof(int));

    memset(hijo2->array_int, -1, m * sizeof(int));

#pragma omp parallel sections{
#pragma omp section{

    // Construcción del hijo1

    for (int i = 0; i < punto; i++)

        hijo1->array_int[i] = padre1->array_int[i];

    int index = punto;

    for (int i = 0; i < m && index < m; i++){

        int gene = padre2->array_int[i];

        if (!find_element(hijo1->array_int, index, gene)){

            hijo1->array_int[index++] = gene;

        }

    }

    while (index < m){

        int gene = aleatorio(n);

        if (!find_element(hijo1->array_int, index, gene))
```

```

        hijo1->array_int[index++] = gene;
    }
}

#pragma omp section
{
    // Construcción del hijo2

    for (int i = 0; i < punto; i++)

        hijo2->array_int[i] = padre2->array_int[i];

    int index = punto;

    for (int i = 0; i < m && index < m; i++){

        int gene = padre1->array_int[i];

        if (!find_element(hijo2->array_int, index, gene)){

            hijo2->array_int[index++] = gene;

        }

    }

    while (index < m){

        int gene = aleatorio(n);

        if (!find_element(hijo2->array_int, index, gene))

            hijo2->array_int[index++] = gene;

    }

}

} // Fin de parallel sections
}

```

En el código usamos `#pragma omp parallel sections` para dividir el trabajo en distintas secciones paralelas ya que las operaciones son independientes y no comparten datos modificables durante su ejecución. Podría añadirse la cláusula `nowait` ya que no es necesario sincronizar los hilos al final de las secciones mejorando ligeramente el rendimiento.

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej5_cruzar** o puede acceder directamente usando este enlace:


 Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

Cuestión 6.

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.

Para reemplazar la función estándar `qsort` por una rutina de ordenación personalizada, lo primero que se ha hecho es implementar el método mergesort en dos funciones: una para dividir recursivamente la población en mitades y otra para mezclar los resultados de cada partición. En vez de llamar a `qsort`, ahora se invoca a `mergeSort(poblacion, 0, tam_pob)`. Para ello, se recurre a OpenMP tasks, de forma que cada llamada recursiva de `mergeSort` puede generar nuevas tareas que se ejecutan en paralelo si el tamaño de la porción a ordenar supera cierto umbral (`threshold`).

Como se puede observar en los resultados

 Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx, el reemplazo de las llamadas a `qsort` por un `mergesort` propio, combinado con la ejecución en paralelo mediante *OpenMP tasks*, ha reducido significativamente el tiempo de ejecución en los casos de mayor tamaño. En lugar de delegar la ordenación a una función secuencial, se implementa un **divide y vencerás** capaz de dividir de forma recursiva la población en sublistas, las cuales se procesan como tareas paralelas siempre que superen un cierto umbral de tamaño. Esta modificación no solo reparte mejor la carga de trabajo entre varios hilos, sino que también puede aprovechar mejor los accesos a memoria, lo que repercute en un *speed-up* notable al aumentar la cantidad de hilos.

Los resultados muestran que, para instancias grandes, el algoritmo ordena la población de manera mucho más ágil al crear múltiples tareas para cada segmento del *mergesort*, mejorando la escalabilidad global. Además, se observan casos en los que, incluso con un único hilo, el nuevo método de ordenación puede comportarse más rápido que la versión secuencial basada en `qsort`, debido tanto al diseño recursivo como a los ajustes de la rutina de mezcla y de gestión de memoria. Por otro lado, en problemas muy pequeños, la ganancia no siempre llega a apreciarse, pues la sobrecarga de crear y coordinar tareas puede superar la ventaja potencial de este mergesort paralelo. Con todo, se concluye que, especialmente en escenarios de mayor envergadura, la combinación de mergesort y OpenMP tasks ofrece mejoras significativas de rendimiento respecto a la estrategia original con `qsort`.

Cuestión 7.

Explicar para qué se usan en general, y si es necesario utilizar de forma explícita en alguna zona del código los siguientes constructores: barrier, single, master, ordered. Considerar también su posible uso en un código paralelo optimizado (como el requerido en la Cuestión 9).

Barrier

Fuerza a los hilos a detenerse hasta que todos los hilos hayan llegado a ese mismo punto. Normalmente se usa para asegurar que todos los hilos hayan completado una sección de código antes de continuar con la siguiente.

Single

Especifica que un único hilo ejecutará el bloque de código, mientras los demás hilos esperan implícitamente hasta que ese hilo termine. Generalmente es usado para realizar tareas que solo deben ejecutarse una vez, como inicialización de variables o escritura en un archivo y reducir la sobrecarga de tareas redundantes ejecutadas por múltiples hilos.

En nuestro código se utiliza a la hora de llamar a la función mergeSort.

Master

Similar a single, pero solo el hilo maestro ejecuta el bloque de código, sin que haya una sincronización implícita al final. Sus usos más comunes son ejecutar tareas específicas del hilo maestro, como administrar I/O o inicializar estructuras compartidas y garantizar que una sección de código no se ejecute en paralelo por otros hilos, pero sin sincronización adicional.

Ordered

Debe estar dentro de un bucle paralelo con la cláusula ordered donde define una región ordenada dentro de un bucle paralelo. Los hilos que ejecutan esta región lo hacen en el mismo orden que lo harían en una ejecución secuencial. Por tanto se usa para asegurar un orden para las operaciones que no son paralelizables, como escribir un archivo en un orden concreto o procesar datos secuencialmente.

Cuestión 9.

Paralelizar la totalidad del algoritmo utilizando las directivas OpenMP que se considere mejor para cada función o bucle (y que pueden ser distintas a las usadas para resolver las cuestiones). Justificar las decisiones tomadas. De este modo, se pretende tener un código paralelo óptimo en memoria compartida que se podrá usar posteriormente para comparar con el código paralelo de paso de mensajes e híbrido implementado en las siguientes prácticas.

Cambios principales en el código

Para paralelizar completamente el algoritmo, se han tomado las mejoras que mejores beneficios reportaban de entre todas las propuestas en la práctica y se han implementado en el mismo código. Las mejoras implementadas han sido:

1. Nuevas directivas OpenMP y uso de `rand_r(&seed)`

En la versión paralela completa, cada hilo dispone de su propia semilla para generar números aleatorios, evitando así las colisiones al utilizar `rand()`. En los bucles principales del algoritmo genético (generación de población, cruce, mutación y recálculo de fitness) se han añadido directivas `#pragma omp parallel for` con distintas políticas de *scheduling* (por ejemplo, `schedule(dynamic, chunk_size)` o `schedule(static, chunk_size)`). El objetivo es equilibrar la carga de trabajo dependiendo de la variabilidad del coste en cada bucle: para mutación, que puede ser más impredecible, se elige *dynamic*, mientras que en el cruce, de coste más uniforme, se utiliza *static*.


2. Sustitución de `qsort` por `mergeSort` con tareas

En lugar de invocar la función estándar `qsort`, que no permite una paralelización directa con OpenMP, se ha implementado un mergesort propio.

Finalmente, se mantiene una última ordenación con `qsort` solo para el mejor individuo, dado que se trata de un array pequeño (el número de genes de ese individuo), y aquí la ganancia de un enfoque paralelo no sería apreciable.

Análisis de los resultados

Como se puede observar en los resultados

 Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx, la versión completamente paralelizada, consigue tiempos de ejecución muy inferiores a la versión secuencial en las instancias de mayor tamaño. Por ejemplo, en los casos con $n=1600$, $m=600$ y $\text{tam_pob}=180$, el tiempo, que en la implementación secuencial podía llegar a superar los 100 segundos, cae a valores entre 15 y 30 segundos en la versión paralela. Esto implica un *speed-up* significativo, a menudo de 3, 4 o más, cuando se utilizan varios hilos.

En términos de eficiencia, se aprecia que, con unos pocos hilos (2 o 4), la relación entre el *speed-up* y el número de hilos se acerca a 1, lo que indica un uso muy productivo de los recursos. Sin embargo, conforme se incrementa la cantidad de hilos (a 8 o 16), la eficiencia se reduce en la mayoría de los casos debido a la sobrecarga de coordinación y a la

saturación de la caché, aunque el tiempo total aún mejora con respecto a la versión secuencial.

Comparando con soluciones intermedias en las que solo se habían paralelizado partes puntuales o seguía empleándose qsort, la nueva implementación extrae más provecho de la arquitectura multicore, especialmente en la etapa de ordenación, que tradicionalmente supone un cuello de botella. No obstante, en problemas muy pequeños, el trabajo total es tan reducido que la diferencia entre secuencial y paralelo casi desaparece, o incluso la gestión de hilos puede penalizar ligeramente.