Metodología de la Programación Paralela
Curso 2023/24

Práctica 1:

Paralelización con OpenMP

En esta práctica se va a llevar a cabo la paralelización del algoritmo genético proporcionado en la Práctica 0 mediante el uso de directivas ofrecidas por OpenMP, estándar *de facto* para la programación paralela en sistemas con memoria compartida. Para ello, se proponen una serie de cuestiones, que se han de realizar de forma progresiva analizando en cada momento la mejora que se obtiene en el rendimiento. Esta práctica representa el 10% de la calificación de la asignatura.

La fecha de entrega de la práctica será el **20 de octubre a las 23:55**. Entregas posteriores no se tendrán en cuenta. La entrega se realizará a través de la tarea habilitada en el Aula Virtual, adjuntando un fichero .zip o .tgz que contenga:

- El código fuente paralelizado y los ficheros *Makefile* y *run.sh* para su compilación y ejecución. Además, el código ha de incluir los comentarios necesarios para ayudar a comprender los cambios realizados.

- Un documento en formato *.pdf* que describa cómo se han resuelto las cuestiones propuestas, justificando las decisiones de paralelización tomadas en cada caso y analizando los resultados obtenidos. El código que se muestre debe coincidir con el que se entregue.

Asimismo, el documento ha de incluir una sección en la que se indique cómo ha sido la coordinación, el reparto del trabajo entre los miembros del grupo y el tiempo dedicado.

Es recomendable, aunque no obligatorio, el uso de la herramienta *GitLab* como repositorio software para ir almacenando los cambios realizados durante el desarrollo de la práctica. Si en alguna cuestión es necesario comparar varias versiones paralelas, se crearán diferentes ramas (una para cada versión). La rama máster contendrá la versión final del código con la que se obtienen las mejores prestaciones.

En aquellos casos en los que sea necesario mostrar la evolución de las prestaciones, se recomienda estructurar la información en tablas. El formato de cada tabla podría ser el siguiente:

N | *N_Gen* | *Tam_Pob* | M_Rate | *N_Threads* | *T_ejec* | *Speed-up* | *Eficiencia*

donde $Speed-up = \frac{t_{secuen.}}{t_{paral.}}$ y $Eficiencia = \frac{Speed-up}{p}$, con p = número de elementos de proceso (hilos en memoria compartida).

Se valorará la creación de gráficos en los que se muestre la evolución del tiempo de ejecución para cada configuración de valores de entrada (*n*, *n_gen*, *tam_pob*, *m_rate*) al variar el número de hilos. Describir el sistema computacional utilizado con la ayuda de la herramienta: `lstopo -f -p --no-legend --no-io sistema.svg`.

A continuación, se muestran las cuestiones a realizar indicando de forma orientativa su distribución en cada semana.

1ª Semana

Cuestión 1. Justificar qué funciones y/o zonas del código son paralelizables. No es necesario indicar las directivas OpenMP que sería necesario utilizar.

Cuestión 2. 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 ello, se recomienda crear en *ga.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) justo antes de generar la población inicial de *tours* aleatorios. Es en esta generación aleatoria donde el efecto sobre el paralelismo de *rand_r* en lugar de *rand* es claramente apreciable.

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, en términos de tiempo de ejecución, respecto a versión secuencial.

Cuestión 4. 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.

2ª Semana

Cuestión 5. 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.

Cuestión 6. 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.

Cuestión 7. 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; }

      mergeSort(poblacion, izq, med);
      mergeSort(poblacion, med, der);

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

Cuestión 8. Realizar el informe final incluyendo las conclusiones generales y valoración personal sobre el trabajo realizado.