UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 2: Comparación de código genético

Curso 2024–2025

Índice

1. Introducción

En esta práctica trabajaremos con un programa que procesa un conjunto de supuestas muestras de códigos genéticos, obteniendo resultados respecto a la similitud entre dichas muestras.

El objetivo de la práctica es paralelizar el programa usando OpenMP, explorando distintas alternativas y comparando las prestaciones obtenidas.

2. Programa objeto de estudio

El programa proporcionado, `codes.c`, trabaja con un conjunto de muestras de códigos genéticos (array `samples`), las cuales se generan de forma aleatoria en la función `generate_data`. Cada muestra se representa como una cadena de texto formada por las letras `A`, `G`, `C` y `T`. Aunque la generación es aleatoria, la semilla es fija, por lo que los datos generados (para unos mismos argumentos del programa) serán siempre los mismos. Eso facilita la comparación de resultados de unas versiones del programa con otras, detectando si una modificación del programa produce resultados incorrectos.

El programa procesa el conjunto de muestras mediante la función `process`, la cual proporciona como resultado, para cada muestra:

- la diferencia mínima con cualquier otra muestra (array `mindiff`),
- la diferencia máxima con cualquier otra muestra (array `maxdiff`),

- el número de muestras que están próximas a ella, es decir, muestras cuya diferencia con ella sea menor que `delta` (array `nclose`).

En esta práctica, los esfuerzos de paralelización del programa se centrarán en la función `process`, debido a que consume la casi totalidad del tiempo de ejecución. Por ello, es necesario analizar la función, prestando especial atención a los dos bucles anidados que hay en ella. En cada iteración del bucle externo se considera la muestra `i` y se calcula la diferencia `d` entre dicha muestra y cada una de las siguientes (de la muestra `i`+1 en adelante, recorridas mediante el bucle `j`). Tras calcular cada diferencia `d`, se actualiza la mínima diferencia, máxima diferencia y número de muestras cercanas, tanto para la muestra `i` (variables `mind_i`, `maxd_i` y `nclose_i`) como para la muestra `j` (elementos `mindiff[j]`, `maxdiff[j]` y `nclose[j]`).

Observa que en una iteración `i` no se calculan las diferencias con las muestras anteriores (0 a `i`-1), puesto que esas diferencias ya han sido calculadas en las iteraciones anteriores, y se han actualizando con ellas los valores de `mindiff[i]`, `maxdiff[i]` y `nclose[i]`. Se evita de esta manera calcular la misma diferencia dos veces.

La función `process` invoca a la función `difference` para calcular la diferencia entre dos muestras, mediante lo que se conoce como distancia de Levenshtein[1]. La función tiene un coste cuadrático con respecto a la longitud de las cadenas que compara y en realidad es la parte computacionalmente más costosa del programa.

Uso del programa

El programa acepta dos argumentos numéricos opcionales, el primero correspondiente al número de muestras y el segundo a la longitud (número de letras) de las muestras. Se trata de una longitud base, pudiendo cada muestra ser ligeramente más corta o larga. Un ejemplo de ejecución es:

```
$ ./codes 100 50
```

donde se utilizan 100 muestras de longitud 50. Si no se especifican los argumentos, se utilizará por defecto 200 muestras de tamaño 900 (en ese caso el tiempo de ejecución puede ser considerable).

Los resultados del programa se guardan en un fichero con el nombre `results.txt`. Esto servirá para poder comparar los resultados de las versiones paralelas con los de la versión secuencial.

3. Desarrollo de versiones paralelas con OpenMP

En los ejercicios del apartado 4 necesitaremos saber el tiempo que tarda tanto el programa secuencial como el paralelo. Así pues, antes de considerar la paralelización, conviene modificar el código original para que muestre por pantalla el tiempo que se tarda en procesar el conjunto de muestras (solo la parte de procesar las muestras, sin incluir su generación o la escritura de resultados). Aunque sea un código secuencial, utiliza la función de OpenMP adecuada para medir tiempos. En el código entregado, esta versión debe llamarse `codes0.c`.

En los ejercicios de este apartado se pide que desarrolles dos versiones paralelas del programa. En ambos la paralelización se centra en la función `process`, que como se ha comentado consume la mayor parte del tiempo de ejecución. Ambas versiones deberán mostrar, además del tiempo de ejecución, el **número de hilos con que se ejecutan**. Se debe **justificar en la memoria** las decisiones tomadas sobre el ámbito de las variables (compartidas, privadas, *reduction*), así como el posible uso de directivas de sincronización.

La complicación principal para implementar las dos versiones paralelas que se piden es el acceso a los arrays `mindiff`, `maxdiff` y `nclose`. Debes analizar el código para determinar si **un mismo elemento del array se modifica por más de una iteración** del bucle que se está paralelizando. Si es así, será necesaria sincronización para evitar condiciones de carrera.

[1] `https://es.wikipedia.org/wiki/Distancia_de_Levenshtein`

Ejercicio 1: Dados los dos bucles anidados de la función `process`, haz una versión en la que se paralelice el bucle interno usando OpenMP. En la entrega, esta versión debe llamarse `codesp1.c`.

Haz distintas ejecuciones y comprueba que el fichero obtenido en cada caso es igual al producido por el programa secuencial (utiliza los comandos `cmp` o `diff` para comparar ficheros, como se indicó en la práctica 1).

Ejercicio 2: Dados los dos bucles anidados de la función `process`, haz una versión en la que se paralelice el bucle externo usando OpenMP. En la entrega, esta versión debe llamarse `codesp2.c`.

Comprueba de nuevo, con distintas ejecuciones, que el fichero obtenido coincide con el del programa secuencial.

Sobre la comprobación de la corrección

En algunos casos, como en este programa, resulta muy difícil detectar incorrecciones en la versión paralela, porque aunque el código sea incorrecto y haya condiciones de carrera, el resultado puede ser correcto en un porcentaje muy alto de las ejecuciones. Ten en cuenta que **un resultado correcto no garantiza que el programa sea correcto**.

4. Estudio de prestaciones

En los siguientes ejercicios se pide analizar las prestaciones midiendo el tiempo de ejecución en el sistema de colas de `kahan`. Se ejecutará el programa con los valores por defecto (200 muestras de tamaño 900), para que el tiempo de ejecución sea considerable.

Ejercicio 3: En este ejercicio se pide evaluar la influencia de la planificación elegida en las prestaciones de las versiones paralelas.

Utilizando el sistema de colas del cluster `kahan`, saca tiempos de ejecución de las dos versiones paralelas realizadas, usando 32 hilos y distintas planificaciones. En concreto, usa las planificaciones siguientes:

- Planificación en la que a cada hilo se le asigna un solo bloque de iteraciones consecutivas, siendo los bloques de los distintos hilos aproximadamente del mismo tamaño.
- Planificación en la que las iteraciones se asignan a hilos de forma cíclica o round-robin (con bloq de una sola iteración).

- Planificación dinámica, con bloques de 1 iteración.

Indica cómo has establecido la planificación en cada caso, cómo has fijado el número de hilos y cómo has lanzado las ejecuciones en el cluster, adjuntando los ficheros de trabajo relevantes utilizados.

Analiza cuál es la mejor/peor planificación en cada versión paralela, justificando a qué puede deberse. Relaciónalo con el concepto de **equilibrio de carga**.

Ejercicio 4: Utilizando el sistema de colas del cluster `kahan`, saca tiempos de ejecución de las dos versiones paralelas realizadas, variando el número de hilos y eligiendo en cada versión la planificación con la que se hayan obtenido mejores resultados en el ejercicio anterior. Para limitar el número de ejecuciones, se recomienda usar potencias de 2 para los valores del número de hilos (2, 4, 8...). Justifica el número máximo de hilos utilizado, teniendo en cuenta no solo los resultados experimentales, sino el también *hardware* empleado.

Muestra **tablas y gráficas de tiempos, speed-ups y eficiencias** y utilízalas para comparar las prestaciones de las dos versiones paralelas. En vista de los resultados, extrae conclusiones indicando **cuál es la mejor versión paralela** o si tienen un comportamiento similar, y trata de **justificar** los resultados que obtengas.

5. Ejercicio adicional

Ejercicio 5: Sobre la versión `codesp2.c`, añade las instrucciones necesarias para que:

- Cada hilo imprima cuántas diferencias entre muestras ha calculado (cuántas veces ha llamado el hilo a la función `difference`).
- El programa imprima el número de diferencias calculadas por el hilo que más diferencias ha calculado.
- El programa imprima el número de diferencias calculadas por el hilo que menos diferencias ha calculado.

En la entrega, este fichero debe llamarse `codesp3.c`. Por ejemplo, si se ejecuta el programa con 8 hilos y 400 muestras de tamaño 200, podríamos obtener algo parecido a lo siguiente:

```
Thread 0:  18725 differences between samples
Thread 3:  11225 differences between samples
Thread 4:   8725 differences between samples
Thread 7:   1225 differences between samples
Thread 6:   3725 differences between samples
Thread 5:   6225 differences between samples
Thread 1:  16225 differences between samples
Thread 2:  13725 differences between samples
Maximum number of differences computed by a single thread: 18725
Minumum number of differences computed by a single thread: 1225
```

Explica en la memoria las modificaciones introducidas para realizar esta versión.

El ejemplo de ejecución mostrado corresponde a la primera de las planificaciones consideradas en el ejercicio 3. Si ordenas las líneas de los distintos hilos según el índice de hilo, verás que las diferencias calculadas por los distintos hilos también están ordenadas. Esto puede ayudar a entender los resultados que habrás obtenido en el ejercicio 3.

6. Sobre el trabajo

La memoria y el código fuente a entregar en esta práctica deben ser el **trabajo personal y original** del correspondiente grupo (de un **máximo de dos estudiantes**, ambos pertenecientes al **mismo grupo de prácticas**). Por tanto, la copia de cualquier parte de la memoria o del código supondrá una nota de 0 en el trabajo completo.

Hay **dos tareas** en PoliformaT para la entrega de esta práctica:

- En una de las tareas debes subir un fichero **en formato PDF** con la memoria de la práctica. No se admitirán otros formatos.
- En la otra tarea debes subir un único archivo comprimido con los ficheros de código fuente de las distintas versiones que hayas desarrollado, junto con los ficheros de trabajo relevantes utilizados para lanzar las ejecuciones. **No incluyas los ejecutables resultantes de la compilación ni tampoco los ficheros de resultados**. El archivo debe estar comprimido en formato `.tgz` o `.zip`.

  Comprueba que todos los ficheros compilan correctamente y tienen el nombre que se especifica en este boletín.

A la hora de realizar las tareas anteriores, hay que tener en cuenta las siguientes recomendaciones:

- Hay que entregar una memoria descriptiva de los códigos empleados y los resultados obtenidos. Procura que la memoria tenga un tamaño razonable (ni un par de páginas, ni varias decenas).

- No incluyas el código fuente completo de los programas en la memoria. Sí puedes incluir, si así lo deseas, las porciones de código que hayas modificado.
- Pon especial cuidado en preparar una buena memoria del trabajo. No se trata de una mera exposición de resultados, sino que la memoria debe tener una estructura adecuada y contener los análisis, explicaciones y conclusiones necesarias. Lee atentamente lo que se pide en cada ejercicio.

Evaluación del trabajo

La puntuación del trabajo se dividirá de la siguiente forma:

0,1 puntos. Estructura de la memoria, vocabulario, redacción.

0,8 puntos. Versiones paralelas de los ejercicios 1, 2 y 5, de los cuales el ejercicio 5 tiene una puntuación de 0,3 puntos. Se valorará también la explicación de los códigos en la memoria.

0,6 puntos. Ejercicios 3 y 4. Se valorará la presentación de resultados, la adecuación de las pruebas y la corrección y coherencia de los análisis y conclusiones.