Informe Trabajo Final Programacion Concurrente

Grupo 8

Integrantes:

- Elizabeth Gasañol, elizabeth.gasanol@gmail.com, legajo 42896.

Introducción

La intención de este trabajo práctico, consiste en el diseño de un modelo capaz de, consultando una planilla de datos de entrenamiento provista por MNIST, identificar correctamente el manuscrito de un número entre 0 y 9.

Para la implementación del modelo nos basaremos en el algoritmo *k-nearest neighbors*. Además de utilizar algunas clases implementadas para poder realizar la interpretación de los datos de entrenamiento de manera concurrente y algunas clases auxiliares necesarias para el manejo de datos.

Las clases implementadas son las detalladas a continuación:

- **Buffer** implementado como monitor, donde se van a almacenar las tareas de lectura de los datos de entrenamiento.
- **Task** como clase abstracta que implementa runnable extendida por **MNISTask**, tarea destinada a la interpretación de los datos de entrenamiento y el cálculo de su distancia con respecto a la muestra a interpretar y por **PoisonPill**, un tipo de tarea cuyo propósito es terminar al **Worker** que la consuma.
- **Worker** que extiende de **Thread**, cuyo propósito es el de tomar tareas del **Buffer** y ejecutarlas concurrentemente.
- **WorkerCounter** implementado como monitor, que evita que se ejecute la obtención de resultados de manera prematura antes de que los workers terminen de ejecutar las tareas.
- **ThreadPool** responsable de inicializar la cantidad de workers provista.
- **BufferedImageReader** y **CSVReader**, responsables de leer y parsear archivos .png y .csv respectivamente para su manejo en el código.
- **KNNHandler** que a partir del número de *k* deseado devuelve los k más cercanos a la muestra de una lista de **Resultado.**
- **Resultado** que actúa como un objeto de solo lectura que almacena el resultado de la comparación de la muestra con una línea del csv de entrenamiento, así como **AlmacenadorDeResultados** que guarda los *k* resultados de cada **Task** ejecutada para su posterior procesamiento para devolver el *k* mas cercano y **ResultadoComparator** que actúa como comparador de instancias de **Resultado** para saber cual es la más cercana.
- **Main** encargada de inicializar y coordinar a las clases nombradas para el procesamiento de los datos y la interpretación de resultados.

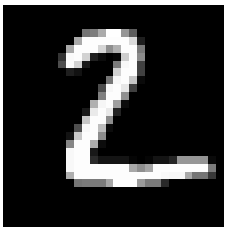
Evaluación

Para la evaluación, se realizaron pruebas de lectura de una imagen provista con 1 y 4 threads y se contempló la diferencia de resultados para k = 1, k = 5 y k = 10, el tamaño de Buffer utilizado fue de 20.

La computadora utilizada para las pruebas tiene las siguientes especificaciones:

- Procesador: AMD Ryzen 7 5700X 8-Core.
- Memoria: 32 Gb.
- Sistema operativo: Windows 11 64 Bits.

Imagen provista:



Resultados:

Para 1 thread: 1 segundo.

Para 4 threads: 0 segundos.

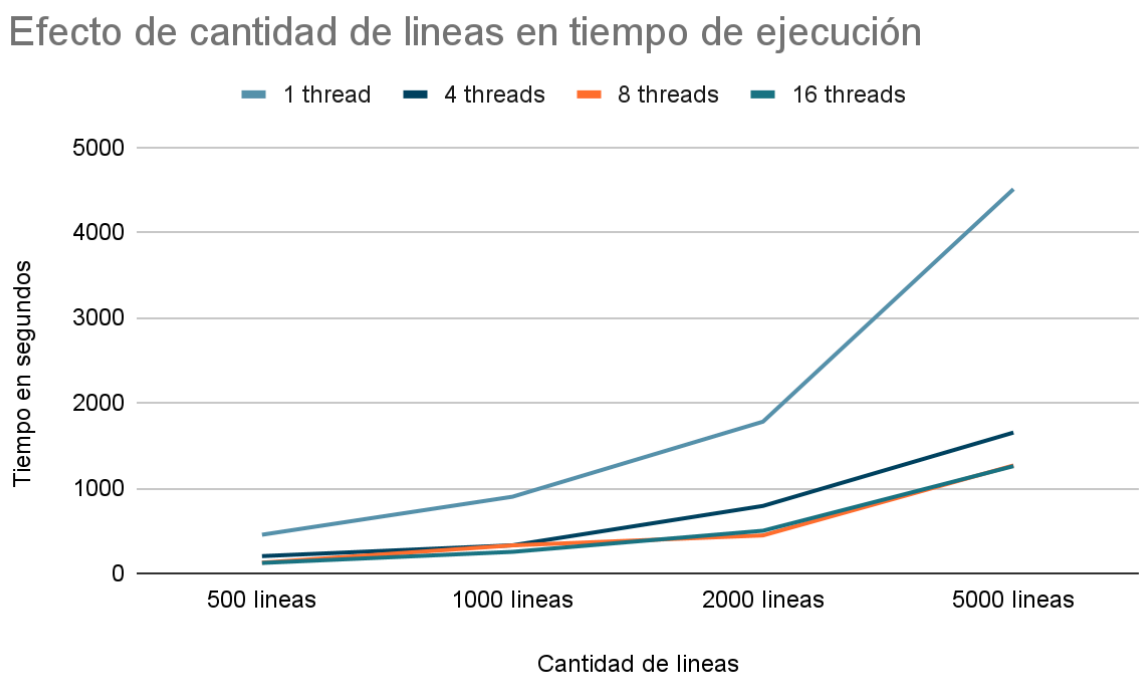
El resultado obtenido fue 2 para todos los valores de k dados.

Adicionalmente, se realizaron pruebas leyendo el .csv con distintas combinaciones de cantidad de threads y cantidad de líneas procesadas, las pruebas siempre se realizaron con un buffer de 20 y un k de 10. Resultados detallados en la tabla a continuación:

Threads	Líneas	Tiempo (segundos)	Precisión
1	500	453	95.6%
1	1000	899	95.4%
1	2000	1780	95.1%
1	5000	4509	95.14%
4	500	202	95.6%
4	1000	329	95.4%
4	2000	791	95.1%

Threads	Líneas	Tiempo (segundos)	Precisión
4	5000	1652	95.14%
8	500	125	95.6%
8	1000	328	95.4%
8	2000	445	95.1%
8	5000	1266	95.14%
16	500	122	95.6%
16	1000	252	95.4%
16	2000	500	95.1%
16	5000	1258	95.12%

Gráfico comparativo



Análisis

A partir de los datos obtenidos, podemos hacer ciertas observaciones con respecto a la variabilidad de tiempo de procesamiento y porcentaje de aciertos.

En primer lugar, la cantidad de k elegidos no parece tener un efecto significativo en la tasa de acierto del algoritmo.

En segundo lugar, podemos afirmar que el tiempo de procesamiento en función de la cantidad de líneas tiene un crecimiento exponencial. Al agregar más threads para realizar los cálculos, se aplana la curva exponencial, por esto mismo, la distancia relativa entre las curvas (Donde cada curva corresponde a la cantidad de threads utilizados) disminuye de forma exponencial, al punto que 16 threads performa casi de manera equivalente a 8 threads.

Esto también podría deberse a la cantidad de tasks a ejecutar en el buffer. Ya que en nuestra implementación decidimos particionar los datos de entrenamiento en particiones de 10000 líneas, es decir, que trabajamos con 6 particiones y por lo tanto con 6 tasks, por lo que a lo sumo 6 workers estarían procesando líneas de manera concurrente, podría hacerse evaluaciones adicionales aumentando la cantidad de particiones y tasks para comprobar su efecto en el tiempo de procesado.

De ser verdad esto último, las pequeñas variaciones de tiempo entre 16 y 8 threads, pueden deberse a la gestión de recursos del sistema operativo en el momento de correr las pruebas.