Practico 1: Paralelizar una aplicación con OpenMP

Tabla de contenido

[Cambios realizados en el código base 2](#_Toc97832872)

[Compilacion con el flag -Ofast 2](#_Toc97832873)

[Cambios del código para la paralelización: 2](#_Toc97832874)

[Paralelización del código 3](#_Toc97832875)

[Análisis de los resultados 5](#_Toc97832876)

[Ejecutar el programa 6](#_Toc97832877)

Dylan CORNELIE

# Cambios realizados en el código base

## Compilacion con el flag -Ofast

Para compilar con el flag -Ofast, necesitamos recuperar el valor de retorno de fscanf. Si no, el compilador no me deja compilar el código.

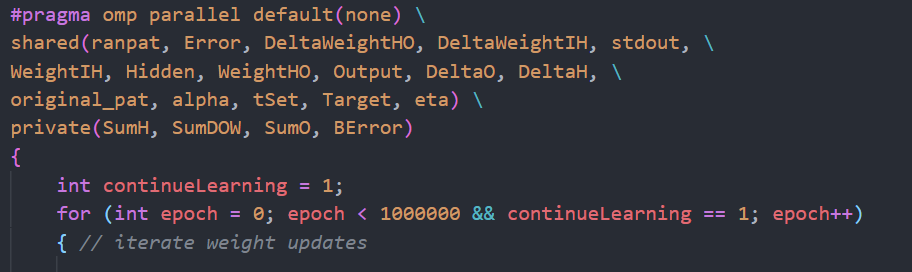
Por eso añadí un if que verifica el valor de retorno del fscanf: Si el valor de fscanf es diferente de 1, significa que hubo un problema durante la lectura del archivo. En este caso, las funciones que utilizan fscanf adentro devuelven con el valor NULL para significar que un problema ocurrió. Las funciones en cuestión son readImg y loadPatternSet.

## Cambios del código para la paralelización:

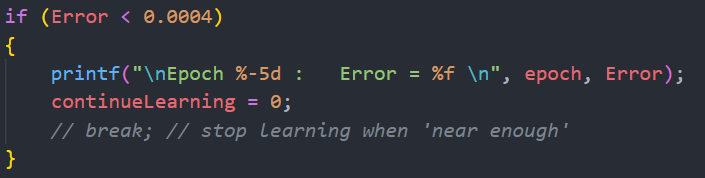
Cuando el Error es inferior a 0.0004 el bucle for debe pararse, en la implementación secuencial, utilizamos la instrucción break. Pero dentro una región paralela, no es posible utilizar la instrucción break. El compilador da un mensaje de error: “invalid branch to/from OpenMP structured block”.

Por eso utilizó un variable continueLearning que será igual a 0 cuando el Error es inferior a 0.0004. Cuando continueLearning es igual a 0, el bucle for principal se acaba y salimos de la región paralela.

Aquí inicializo la variable continueLearning a 1 y dentro del bucle for, dicemos que el bucle se pare cuando epoch mayor o igual que 1.000.000 o cuando continueLearning es diferente de 1.



Este variable continueLearning, como dicho antes, esta igual a cero cuando el Error es inferior a 0.0004.

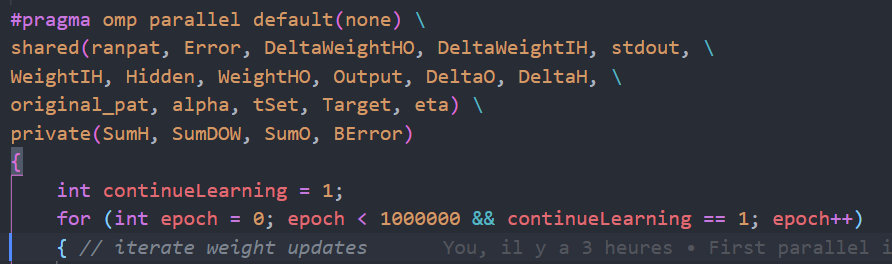


Así, los threads paran ejecutar el bucle for y pueden salir de la región paralela y el programa sigue su ejecución.

# Paralelización del código

La primera etapa fue determinar qué parte del código, sería interesante paralelizar. Después un análisis del código, me dio cuenta que la mayoría del tiempo de ejecución es deuda al entrenamiento de la red neuronal. El entrenamiento de la red neuronal está representado con la función trainN, por eso, la paralelización se hizo en esta función.

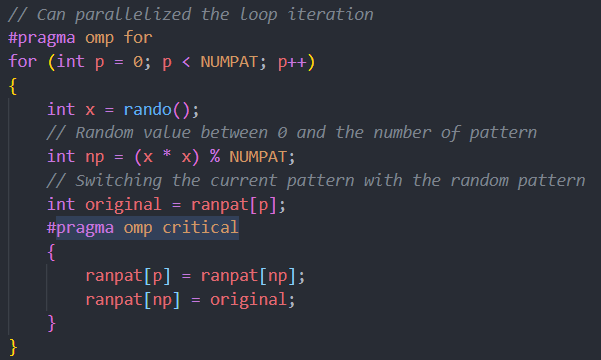
Para evitar el overhead de la creación de los threads en todas las iteraciones del bucle de entrenamiento, creamos una región paralela antes de entrar en el bucle for principal.



Podemos ver que tenemos únicamente cuatro prívate variable: SumH, SumDOW, SumO y BError.

No he paralelizado el primer bucle que inicializa el vector ranpat con sus valores por defecto porque es tan rápido hacerlo en un solo thread que hacerlo con múltiple threads. Podéis encontrar el código de test en la carpeta llamada test en el fichero long\_for\_loop.c.

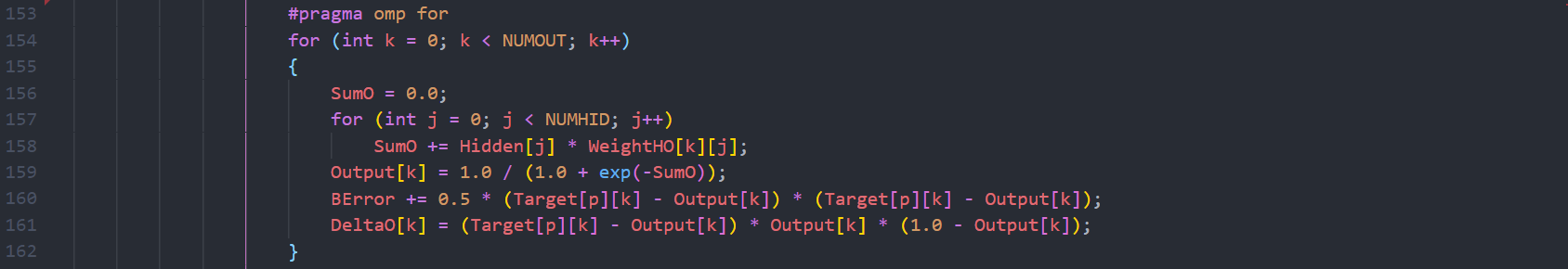
La primera paralelización fui el bucle que aleatoriza el orden de los pattern a analizar. Por eso utilizo un #pragma omp for para que los threads se compartan las iteraciones del bucle for. Una sutileza fui utilización de una región crítica para que cada threads pueda intercambiar la orden de los pattern de manera atómica.



Después, paralelizó el contenido del bucle que itera sobre cada elemento de un batch.

Cada bucle for es paralelizado con la instrucción #pragma omp for, paralelizo el bucle exterior. Intenté paralelizar el bucle interior de cada bucle for pero el rendimiento es peror que cuando paralelizo el bucle for exterior.

En un fichero test, intento comparar el rendimiento entre una paralelización del bucle exterior i interior, i resulta que el rendimiento es mejor con la paralelización del bucle exterior, que tenga un gran nombre de iteración o no. Pero eso es de verdad únicamente cuando utilizamos el flag -Ofast para compilar el código.

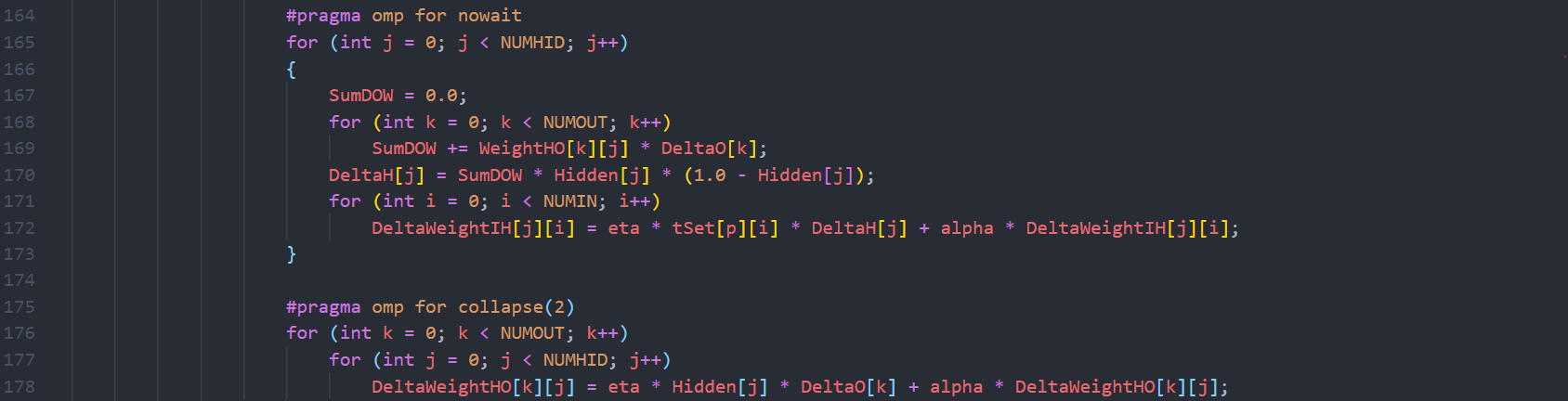


Aquí cada thread tiene una copia de la variable SumO, por eso no es necesario añadir una región critica porque cada thread va iterar el segundo bucle enteramente et no tendremos conflictos. Igual para la variable BError, cada thread tiene una copia de esta variable para que no tengamos un conflicto cuando un thread aumenta el valor de BError.

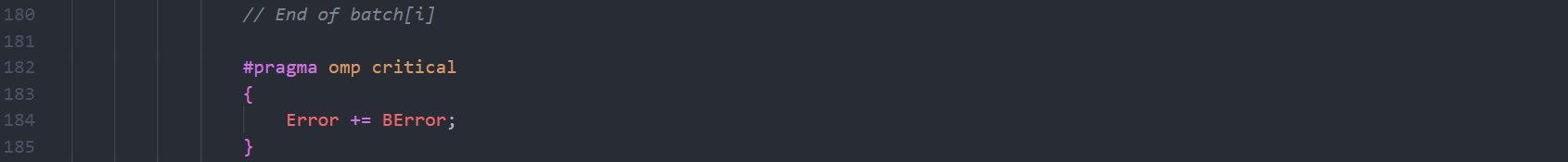
Hacemos lo mismo para los tres bucles que pone al día los matrices de la red neuronal.

Para el antepenúltimo bucle utilizo un #pragma omp for nowait porque el siguiente bucle no puede ser ejecutado sin que este bucle sea terminado.

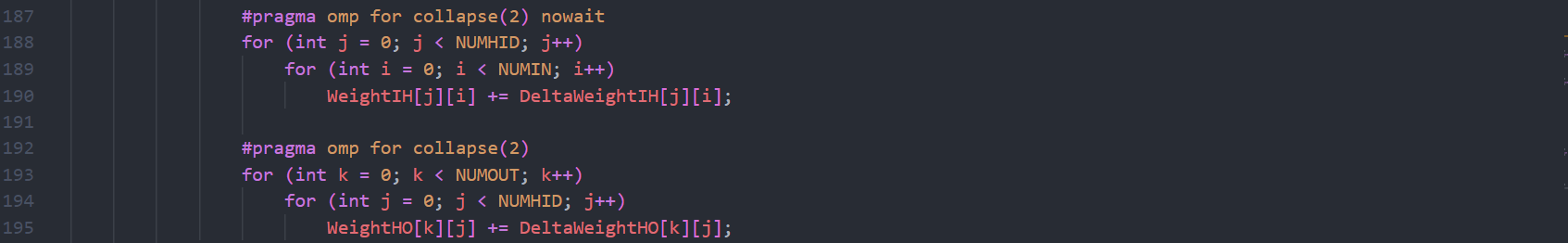
Para el ultimo bucle, utilizo un #pragma omp for collapse(2) porque tenemos dos bucles for uno después del otro. Esta instrucción permite transformar un doble bucle for en un bucle for único cuyo iteraciones van ser compartido entre los threads.



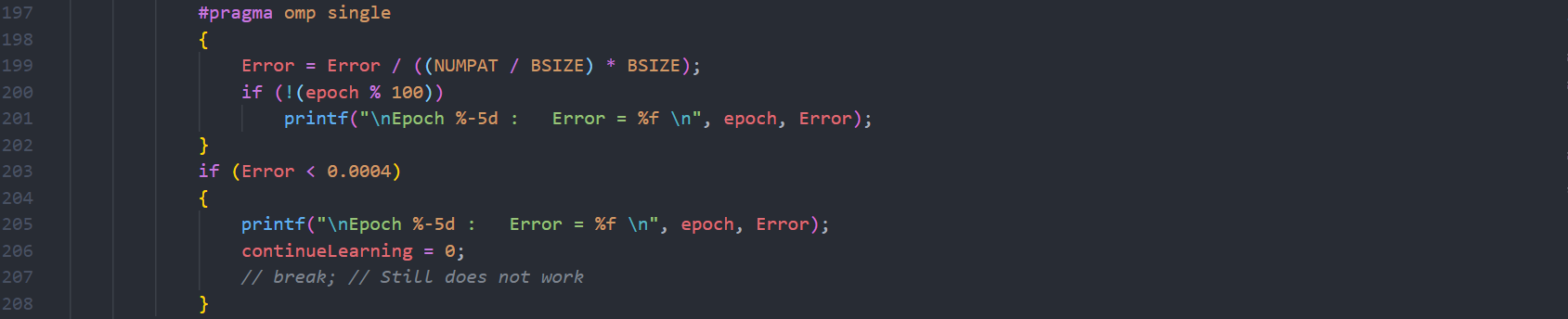
Cuando la iteración batch se acaba, hay que cada thread añada su valor de BError a la variable compartida Error. Para que no haya conflictos, utilizo un #pragma omp critical para que cada thread accedan a la variable Error uno a la vez.



Después ponemos al día las valores de lost WeightIH y WeightHO, por eso utilizo un #pragma for collapse(2) nowait. La cláusula nowait permite a los threads de no sincronizarse después el primer bucle for. La meta es de empezar a tratar el segundo bucle lo más rápido posible.



Después que el programa hubiera iterado sobre todos los elementos del batch, calculamos el Error medio y mostramos en la pantalla avanzado de la ejecución. Este parte es únicamente ejecutado por un solo thread, de ahí el uso de la instrucción #pragma omp single.



Si el Error es inferior a 0.0004 salimos del bucle for de 1.000.000 de iteraciones y de la región paralela.

# Análisis de los resultados

Para mesurar el tiempo de ejecución, cada versión del programa fue ejecutado 5 veces, y las valores más alta y más baja han sido eliminadas para al final hacer una media sobre los 3 valores restantes. El programa fue ejecutado sobre wilma cluster con la optimización -Ofast del compilador GCC.

|  |  |  |  |
| --- | --- | --- | --- |
| **Nombre de thread** | **Tiempo ejecución (segundos)** | **Speed up** | **Eficiencia** |
| **1** | 159,6375597 | 1 | 1 |
| **2** | 323,3813661 | 0,49 | 0,25 |
| **4** | 139,556464 | 1,14 | 0,29 |
| **6** | 90,02437605 | 1,77 | 0,30 |
| **8** | 57,82015227 | 2,76 | 0,35 |
| **12** | 42,3435765 | 3,77 | 0,31 |

Podemos ver que el tiempo de ejecución lo más pequeño se obtiene con 12 threads: 42,3 segundos con un speed up de 3,77. Pero cuando miramos la eficiencia, ejecutar el programa con 12 threads no es lo más interesante desde un punto de visto de la eficiencia.

En efecto la eficiencia la más alta se obtiene cuando ejecutamos el programa con 8 threads, aunque el tiempo de ejecución es mayor, la eficiencia es mejor.

# Ejecutar el programa

Para ejecutar el programa sobre el wilma cluster. Solo ejecuta la command: sbash ./submit\_job.sh en la carpeta secuencial para la versión secuencial o la carpeta parallel para la versión paralela.

Cuando ejecuta la versión paralela, en el script, se puede cambiar el número de thread que ejecutan el programa cambiando el valor de estas líneas:

**#SBATCH -n** *NUMBER\_OF\_THREADS\_TO\_USE*

**export OMP\_NUM\_THREADS=** *NUMBER\_OF\_THREADS\_TO\_USE*

Para ejecutar el programa localmente, simplemente hay que ejecutar el siguiente comando en la carpeta parallel:

gcc \*.c -lm -fopenmp -Ofast -o test

Se crea un ejecutable llamado test, simplemente tenéis que ejecutarlo.

Para ejecutar la versión parallel podéis ejecutar el mismo comando en la carpeta sequencial.