Practico 2: Paralelizar una aplicación con MPI

Tabla de contenido

Cambios realizados en el código base

Definición de macro

Para mejorar la legibilidad del código y facilitar el cambio del comportamiento de la aplicación, utilicé tres macros.

La primera macro `MASTER` es utilizado para saber si únicamente el nodo "MASTER" tiene que ejecutar un parte específico del código. De esta manera, es más fácil leer el código.

La segunda macro `ERROR_MULTIPLICATION` es utilizado para cambiar el error desde el cual decidimos que la red neuronal esta entrenada. Porqué cuando estamos desarrollando la aplicación no es necesario ir hasta un error de `0.0004` solamente queremos verificar que el programa funciona como debería.

La tercera macro nos permite activar o desactivar la visualización de los resultados cuando la función `runN()` es llamada.

Salida de pantalla

Para que la salida de pantalla queda limpia, cada llamada a la función `printf` se hace únicamente por el nodo MASTER.

Optimización

Una optimización fue hecha para mejorar la velocidad de ejecución del programa. Después del procesamiento de un batch para actualizar las matrices `WeightIH` y `WeightHO`, en lugar de utilizar dos bucles `for` utilizo únicamente un bucle donde actualizo las dos matrices. Por eso, actualizo únicamente la matriz `WeightHO` si los indexes son válidos. (`Update WeightHO[j][i] si j < 10 && i < 1024`). Esto permito acelerar el tiempo de ejecución.

Paralelización del código

Teoría

Antes implementar MPI en el programa, la primera etapa fue analizar el código y observar cual región sería mejor paralelizar para minimizar el overhead de la comunicación y maximizar los ganados de la paralelización.

Por explicar el razonamiento utilizaré dos diagramas. El primero diagrama explica como el programa función cuando se ejecuta en serie y el segundo cómo funciona cuando se ejecuta con MPI.

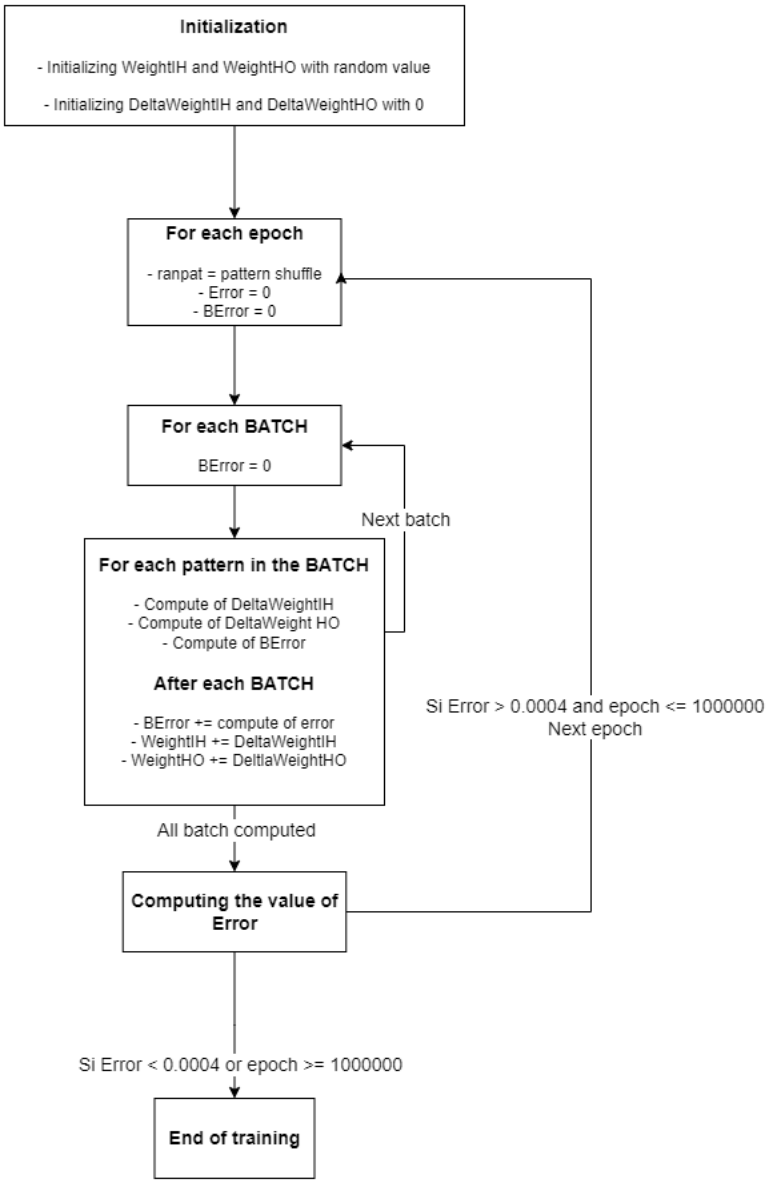

Diagrama de funcionamiento del programa en serie

Para paralelizar este programa dos soluciones están disponibles.

La primera es paralelizar el bucle que calcula los matrices `DeltaWeightIH`, `DeltaWeightHO`. Es decir, repartir todos los patrones dentro un BATCH entre los nodos. Sin embargo, implementar la paralelización de esta manera no es buena idea porque al fin de cada BATCH, los nodos deberían sincronizarse y por tanto enviarse mensaje, resulta que el overhead de comunicación será muy elevado.

La segunda soluciona es paralelizar el procesamiento de los BATCH, es decir repartir todos los BATCH de un epoch entre los nodos. De esta manera los nodos deben sincronizarse únicamente una vez sus BATCH asignados calculados. Por eso los nodos deben tener en común los matrices `WeightIH`, `WeightOH`, `DeltaWeightIH`, `DeltaWeightHO` y sus Errores al fin de cada epoch. Para los matrices, calculamos el promedio de cada matriz para tener en cuenta los resultados de cada nodo. Para el Error, sumamos cada Error de cada nodo. Una vez echo, todos los nodos tienen los mismos dades. Al principio del entrenamiento de la red neuronal, es importante que todos los nodos tengan los mismos dades para que los resultados calculados sean coherentes. Por eso únicamente un nodo inicializa el array de ranpat entre cada época y al entrar del bucle de las épocas únicamente un nodo inicializara las matrices `WeightIH` y `WeightHO` para después comunicarlos a los otros nodos.

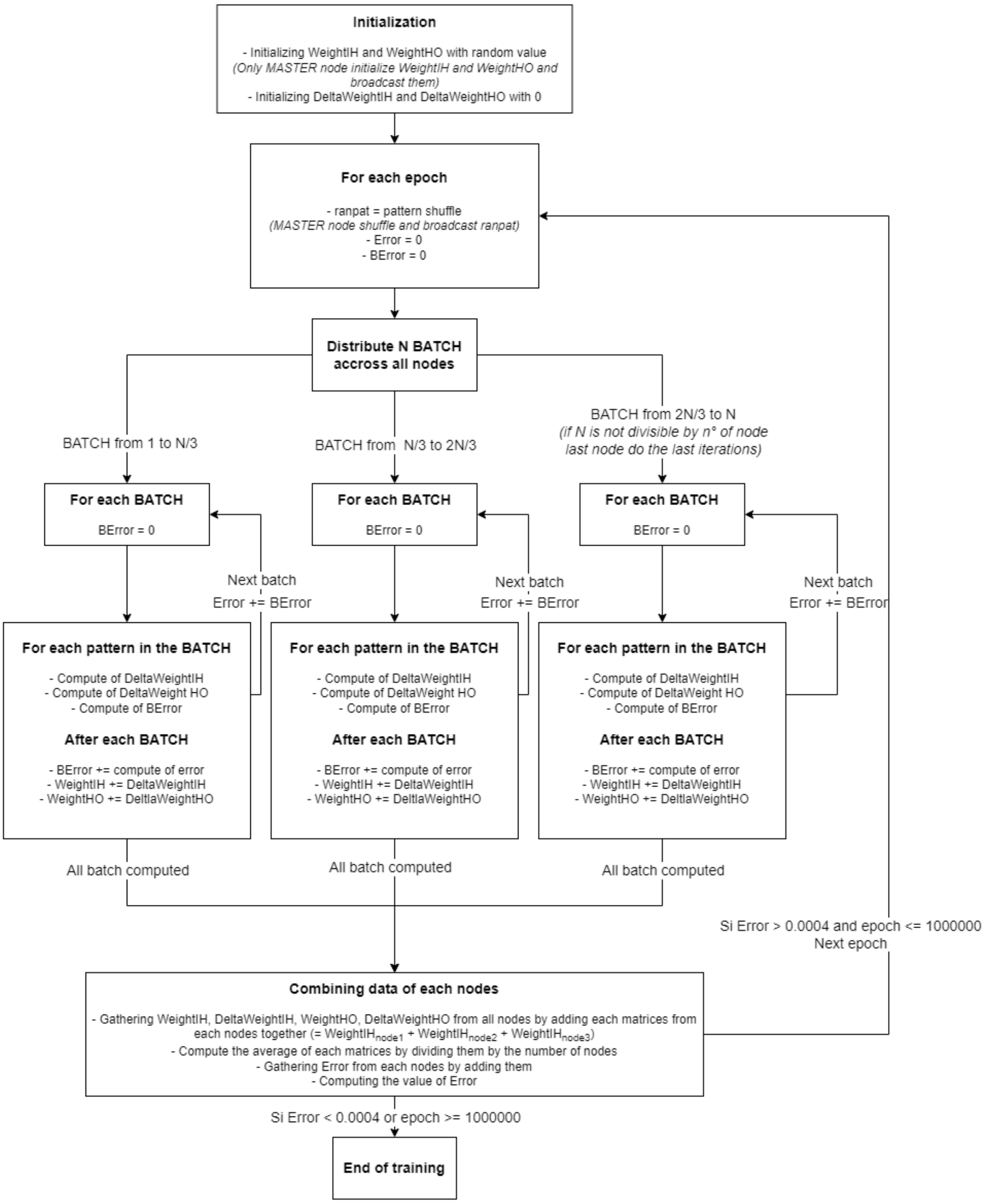

Diagrama de funcionamiento del programa en paralelo

No era posible paralelizar el bucle de las épocas simplemente por qué no podemos saber de antemano si es necesario procesar un epoch sin tener el resulto del epoch “epoch - 1”.

Implementación con MPI

Para utilizar MPI, tenemos que cridar dos funciones qué son `MPI_Init(…)` que permite inicializar MPI y `MPI_Finalize(…)` para finalizar MPI.

Para implementar esta implementación con MPI, utilicé dos nuevas variables global: `numberOfProcess` que contiene el número de nodos ejecutando el programa y `rank` que contiene el número del proceso ejecutando el código, cada proceso se ve asignado un numero para referirse a él. Para inicializar estas variables, utilizamos dos función de MPI, respectivamente `MPI_Comm_size(…)` y `MPI_Comm_rank(…)`.

Para que todos los nodos tengan las mismas matrices `WeightIH` y `WeightHO` y `ranpat` únicamente el proceso `MASTER` inicializara estas matrices y después les enviara utilizando la función `MPI_Bcast(…)`. Esté función permite broadcast datos a todos los procesos. Para que sea únicamente el proceso `MASTER` que inicializa las dos matrices, utilizo la variable `rank`.

Después para repartir las iteraciones de los BATCH, decidí repartir las iteraciones divisando la variable `work` que representa el numero total de iteraciones, entre el numero de nodos representado con la variable `numberOfProcess`. Si las ultimas iteraciones no han sido repartido, el último proceso las hará. Intenté implementar otra manera para repartir las iteraciones. Si el número de iteraciones no es divisible por el número de proceso, las iteraciones se reparten desde el proceso `0` hasta el proceso `n-1` y el proceso `n` coge las ultimas iteraciones no repartidas. Pero implementar esto no me dio un tiempo de ejecución significativamente más rápido qué con mi primera implementación.

Una vez todos los BATCH calculados, tenemos que coger los resultados de cada proceso y unificarlos, por eso utilizo la función `MPI_Allreduce(…)` sobre las matrices `WeightIH`, `WeightOH`, `DeltaWeightIH`, `DeltaWeightOH`, la operación de reducción es una suma. Dentro los parámetros de crida de la función `MPI_Allreduce`, por el `sendBuffer` parámetro utilicé una macro especial llamada `MPI_IN_PLACE` que permite que el `sendBuffer` parámetro sea lo mismo que el `receiveBuffer`. En nuestro caso, nos permite evitar crear otras matrices `WeightXY` y `DeltaWeightXY`, las matrices serán directamente reducido en lugar de las matrices originales. Es importa notar que utilizamos un `Allreduce` y no un Reduce, por esta manera todos los nodos tienen los resultados de los reducciónes. Una vez las matrices reducidas, necesitamos dividirlas por el número de procesos para obtener la media.

Para las errores hacemos también un `MPI_Allreduce(…)` y utilizamos también un `MPI_IN_PLACE` para el `receiveBuffer` de esta manera el Error es remplazada por la suma de los errores de cada nodo, después el funcionamiento es el mismo que la versión en serie.

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 mpicc.

	1	2	4	6	8	12	16
Tiempo de ejecución 1	339,19	326,91	278,05	213,05	364,79	332,39	726,11
Tiempo de ejecución 2	339,13	326,08	276,34	212,83	363,93	330,92	724,28
Tiempo de ejecución 3	338,92	326,2	276,48	215,55	365,3	332,99	727,02
Tiempo de ejecución 4	339,97	325,88	276,79	214,44	364,04	334,84	724,39
Tiempo de ejecución 5	339,73	326,04	276,12	213,34	365,67	330,65	725,94
Mean ejecución	339,35	326,11	276,54	213,61	364,71	332,10	725,48
Numero de iteración 1	1148	2186	3139	3139	4443	7604	10758
Numero de iteración 2	1148	2186	3139	3139	4443	7604	10758
Numero de iteración 3	1148	2186	3139	3139	4443	7604	10758
Numero de iteración 4	1148	2186	3139	3139	4443	7604	10758
Numero de iteración 5	1148	2186	3139	3139	4443	7604	10758
Mean iteración	1148	2186	3139	3139	4443	7604	10758
iteración / segundo	3,38	6,70	11,35	14,70	12,18	22,90	14,83
Total encerts 1	924	924	925	925	926	926	926
Total encerts 2	924	924	925	925	926	926	926
Total encerts 3	924	924	925	925	926	926	926
Total encerts 4	924	924	925	925	926	926	926
Total encerts 5	924	924	925	925	926	926	926
Mean total encerts	924	924	925	925	926	926	926
Speed UP	1,00	1,98	3,36	4,34	3,60	6,77	4,38
Eficiencia	1,00	0,99	0,84	0,72	0,45	0,56	0,27

Podemos ver que el número de iteración por segundo lo mas alto es obtenido con 12 procesos, después el rendimiento disminuye. Esto se puede ver mirando el speedup. Esta disminución es deuda a el overhead de comunicación que aumento cuando el número de proceso aumento. Si únicamente nos interesa la eficiencia lo más interesante es utilizar 2 procesos. De esta manera casi obtenemos un speed up de 2. Eso significa que el overhead de paralelización es casi imperceptible.

Una cosa interesante seria utilizar OpenMP y MPI juntos para minimizar el tiempo de ejecución.

Para concluir con este programa lo mas interesante es ejecutarlo sobre dos maquines del cluster wilma, con más maquines no ganamos en eficiencia sino perdemos.

Ejecutar el programa

Para ejecutar el programa sobre el wilma cluster. Solo ejecuta la command: sbatch ./job_mpi.sh

En el script, se puede cambiar el número de nodos que ejecutan el programa cambiando los valores de estas líneas:

#SBATCH -N NUMBER_OF_NODE_TO_RUN_ON # number of nodes

#SBATCH -n NUMBER_OF_CORE_PER_NODE # number of cores/processes

Para ejecutar el programa localmente, simplemente hay que ejecutar el siguiente comando en la carpeta parallel (hay que instalar mpich antes):

```
mpicc *.c -o test -lm -Ofast

mpirun -np NUMBER_OF_NODE ./test
```

Se crea un ejecutable llamado test, simplemente tenéis que ejecutarlo con el comando mpirun.