# Paralelización de redes neuronales con MPI

Ignacio Rojo Pérez y Marcel Sarraseca Julian

Mayo 2021

## 1 Introducción

El objetivo de esta práctica es la paralelización de una red neuronal y de optimizar su rendimiento a través de MPI.

El algoritmo a trabajar se divide principalmente en dos funciones: TrainN() y runN().

La función TrainN() se encarga de entrenar la red neuronal con un conjunto de patrones, para que posteriormente la red pueda clasificar un conjunto de datos mediante la función runN().

## 2 Análisis

En este paradigma de paso por mensajes se tiene en cuenta la granularidad del trabajo que van a realizar los procesos, una granularidad muy fina conllevaría a una saturación de la red de comunicaciones y por lo tanto una ejecución ineficiente de la aplicación debido a la gran cantidad de mensajes. Se ha optado por una granularidad gruesa, se trabajará en batches o lotes. Cada proceso realizará un número determinado de batches en función de su identificador de proceso, conocido en el código como "my_rank". Tal como se puede ver en la siguiente figura, cada procesa realizará una cantidad de batches equitativa, esta ha sido la forma en que se ha balanceado la carga de trabajo:

```
for (int nb = my_rank*(NUMPAT/BSIZE)/nprocs; nb < (my_rank + 1)*(NUMPAT/BSIZE)/nprocs; nb++)
```

Figure 1: Repartición equitativa de la carga de trabajo en baches

Primero se ha definido desde qué batch empieza cada proceso

$$nb = my\_rank * (NUMPAT/BSIZE)/nprocs.$$

En este caso se ha dividido (NUMPAT/BSIZE) entre el número de procesos "nprocs", una vez se obtenga este valor, se hace el producto por el identificador de proceso, ej: P0 empieza por el batch [(NUMPAT/BSIZE)/nprocs] * 0, P1

empieza por el batch [(NUMPAT/BSIZE)/nprocs] * 1, P1 empieza por el bach [(NUMPAT/BSIZE)/nprocs] * 2 y así sucesivamente.

Segundo, se debe calcular hasta que batch ejecutará un proceso, el número de baches será equitativo para equilibrar la carga de trabajo entre los procesos. Se ha empleado esta sentencia nb ¡ (my_rank + 1)*(NUMPAT/BSIZE)/nprocs, junto con la sentencia anterior, se permite que el proceso 0 en caso de que (NUMPAT/BSIZE)/nprocs = 50, empieza por el batch 0 y acabe en el 49, el P1 inicia en el batch 50 y termina en el 99 y así sucesivamente.

Cabe remarcar que en este caso concreto no se ha hecho ningún tratamiento en cuanto a (NUMPAT/BSIZE)/nprocs no dé un valor divisible. En este caso el valor (NUMPAT/BSIZE)/nprocs = (1934/50)/4 = 38.68 / 4 = 9.67 baches por proceso.

El trabajo restante se ha considerado despreciable y solo se ha trabajado con la parte entera (9 baches por proceso). Esta simplificación no ha tenido repercusiones negativas en la ejecución y funcionamiento de la red neuronal.

# 3 Paralelización del algoritmo: Punto a Punto

Se divide en dos partes, la que ejecutará el root y la que ejecutaran los demás procesos.

En la parte de los procesos, cada uno de ellos hace un Send() para enviar su estructura de datos pertinente: Error, WeightHO, y WeightIH, y reciben la estructura "Error", es decir el "Error" promedio calculado por el proceso root. En la otra parte, el proceso root con identificador "0" recibirá "recv()" todas las estructuras de datos de los demás procesos una por una, acumulara sus valores en sus estructuras, realizará el promedio y enviará "Send()" la estructura "Error" al resto de procesos.

En cada epoch se repetirá la secuencia hasta que el error promedio sea inferior a 0.0004.

# 4 Paralelización del algoritmo: Colectivas

En este caso, la forma de paralelizar el código se resume en tres directivas "Allreduce()" ejecutadas por todos los procesos, las estructuras "Error, WeightHO, y WeightIH" de todos los procesos se acumulan unas con otras y todos los procesos tienen los mismos valores en dichas estructuras, luego cada proceso realiza el promedio de los valores en sus estructuras y se repetirá esta secuencia en cada epoch hasta que el error medio sea inferior a 0.0004.

# 5 TAU

TAU es una herramienta para medir y perfilar el rendimiento de aplicaciones paralelas, en este caso ha sido usada para visualizar el comportamiento de cada uno de los procesos en la ejecución de la red neuronal.

Usando TAU para la versión punto a punto en una marca temporal aleatoria de la ejecución. Utilizando 4 procesos. En la figura 2, podemos ver el comportamiento de estos sobre las funciones anidadas Send(), Barrier() y Recv() dentro de la función TrainN().

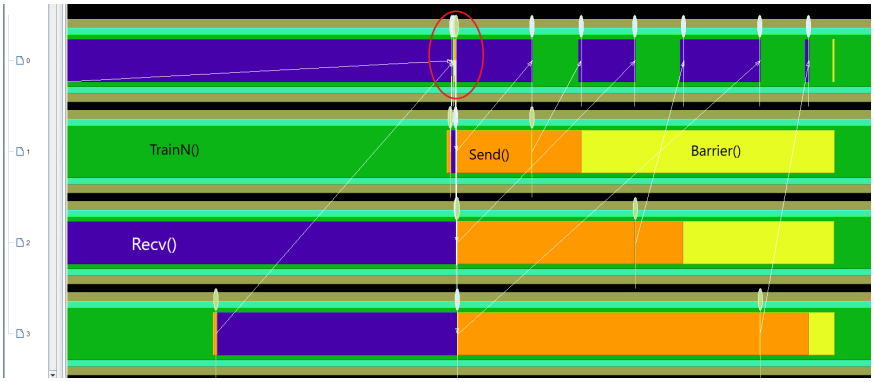


Figure 2: Punto a Punto: Send & Recv

A grandes rasgos se puede diferenciar que se van produciendo envíos y recepciones entre todos los procesos al PID 0, una vez tiene todas las estructuras de datos como se ha explicado anteriormente en la paralelización del algoritmo.

Cuando se concluye el paso anterior, cada proceso entra en una función barrier(), a la espera de que todos los demás se sincronizen.

En la figura 3, que es un zoom dentro de la región indicada en la anterior figura, se puede ver con más detalle, el momento en el que el proceso con el PID 0, ha recibido las estructuras, las promedia y las vuelve a enviar a todos los procesos esclavos.

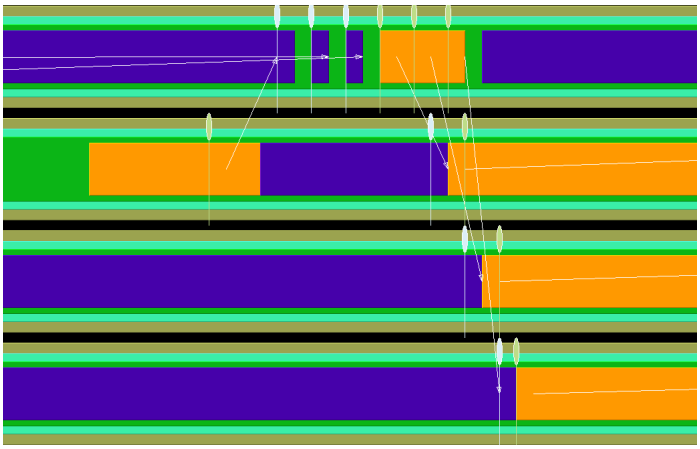

Figure 3: Punto a Punto

Para el caso de la versión colectiva, también se observa como se comporta cada proceso dentro de la función TrainN(). En la figura 4, los 4 procesos van realizando el cómputo de la función hasta que se llega a la llamada de función AllReduce(), que consiste en acumular las estructuras WeightHI, WeightHO, error y compartirlas con los demás procesos. Se seguirá computando la función TrainN(), se volverá a llegar a las funciones AllReduce() hasta que se termine el bucle principal de cómputo.

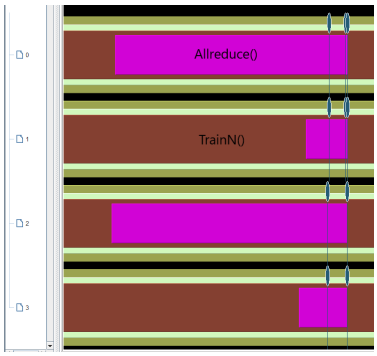


Figure 4: Colectivas

En el caso de la figura 5, se puede ver el final de la funcion TrainN(), dónde se observan los últimos allreduce() que se realizan, los procesos esclavos, llamaran a MPI_Finalize() y terminarán. Mientras que el proceso principal con valor my_rank =2, será el que ejecute la función RunN(), con todas sus funciones anidadas, y finalice.

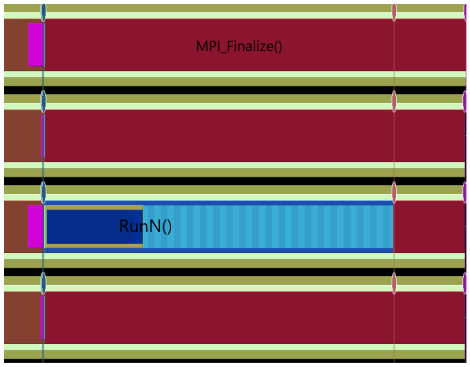


Figure 5: Colectivas

# 6 Resultados

En este paradigma se han realizado dos versiones a través de MPI, una versión de comunicación punto a punto y una versión de comunicación colectiva. Los resultados obtenidos se muestran en la siguiente tabla.

| Tipus | Processos | Temps (s) | Cost (s) | Eficiència | Speed up | Overhead | Error | Epoch |
|---|---|---|---|---|---|---|---|---|
| Seqüencial | 1 | 293 | 293 | - | - | - | - | 1148 |
| MPI P2P | 2 | 161.81 | 323.62 | 0.91 | 1.81 | 30.62 | 0.000395 | 1243 |
| MPI P2P | 4 | 68.85 | 275.1 | 1.06 | 4.26 | -17.9 | 0.000398 | 998 |
| MPI Collective | 4 | 72.35 | 289.1 | 1.01 | 4.05 | -3.9 | 0.000395 | 1001 |

Figure 6: Tabla de resultados

Se puede apreciar que la versión más eficiente es la versión punto a punto, pero no dista significativamente de la versión con comunicación colectiva. La mejora obtenida con 4 procesos en la versión punto a punto es de un Speedup = 4.26x respecto a la versión secuencial y la versión colectiva tiene un Speedup = 4.05 x respecto la versión secuencial.

En este caso, no se ha podido trabajar con más nodos de cómputo y por ende con más procesos, debido a la alta demanda de recursos en el clúster Wilma, se ha acotado a trabajar con 4 procesos como máximo.

De las dos versione presentadas, en cuanto a tiempo y eficiencia, la mejor versión resulta ser la versión punto a punto.

De hecho las dos versiones son muy eficientes, si se divide el tiempo secuencial entre el numero de procesos 293 / 4 = 73.25 s, el tiempo resultante de cada versión 68.85 (P2P) y 72.35 (colectiva) son un poco menor.

En cuanto a overhead, tal como se ha experimentado, al aumentar el número de recursos el overhead no ha aumentado como pasaba en memoria compartida, en este caso se ha reducido y esto es debido a la adecuada granularidad de trabajo que realiza cada proceso.

Hay que recordar que en la entrega anterior, con OpenMP a medida que se escalaban recursos, también crecía el overhead. En los casos tratados esto no ha sido así, aunque se deberían escalar a más recursos para poder sacar conclusiones más sólidas.

# 7 Conclusiones

En esta entrega se ha paralelizado la red neuronal con un paradigma de paso por mensajes llamado MPI. Con este paradigma se busca obtener una ejecución

eficiente y romper los límites de escalabilidad que tiene el paradigma de la media compartida.

Con MPI se han desarrollado dos versiones, una versión de comunicación punto a punto y una versión de comunicación colectiva.

En términos de tiempo y eficiencia, la versión punto a punto resulta ser más rápida y eficiente en un 5% que la versión de comunicación colectiva.

Hay que resaltar que el éxito de emplear MPI yace en la adecuada granularidad y el balanceo de carga de trabajo por proceso.

En este caso se ha optado por una granularidad gruesa.

Una granularidad fina o media sería un problema debido a la gran cantidad de mensajes transferidos y la red de interconexión se saturaría, convirtiéndose en el cuello de botella de la ejecución, es decir se tendría el mismo problema que en memoria compartida al escalar recursos.

En cuanto al balanceo de carga se ha dividido el trabajo a nivel de bach y cada proceso ha ejecutado el mismo número de baches.

Aunque cabe decir que al ejecutar trainN(), en la versión punto a punto el proceso root realiza más trabajo que los demás: hace las acumulaciones, promedios y comunicaciones pertinentes (sends y recvs) todos los procesos mandan o reciben del proceso root.

Sin embargo, aunque en la versión colectiva todos los procesos realizan la misma cantidad de trabajo, con acumulaciones y promedios incluidos, esta realiza más comunicaciones entre procesos que la versión punto a punto.

Dados los resultados, se concluye que la granularidad gruesa es la más adecuada para este caso.

En este caso, las dos versiones presentadas con MPI, no tienen los problemas de escalabilidad que había en memoria compartida, el overhead no aumenta progresivamente, sino que además disminuye en el caso de emplear 4 procesos.

Hay que tener en cuenta que todas las conclusiones y afirmaciones son basadas en los experimentos realizados, en este caso las experimentaciones no ha sido tan ricas como se esperaba en un inicio debido a la saturación de Wilma, eso no quita que al escalar a "n" recursos haya problemas como podría ser la saturación en la red de comunicaciones y de como resultado una ejecución ineficiente.

# 8 Anexos

## 8.1 Compilación y Ejecución del programa

Antes que nada, para realizar correctamente la compilación del programa se requieren de los siguientes archivos: nn-main.c common.h, common.c, optdigits.cv y optdigits.tra.

Primero se actualiza a la versión más reciente del compilador gcc, en nuestro caso se empleará la versión gcc/10.2.0 con la sentencia "module add gcc/10.2.0".

Segundo, se carga el móodulo que permite emplear MPI con la sentencia "module load openmpi/3.0.0".

En cuanto a la compilación del código, quedaría de la siguiente forma: mpicc -std=c99 -Ofast -lm -lmpi common.c nn-main.c -o main

Se hace a través del compilador mpicc, este funciona con gcc y permite compilar un programa paralelizado con MPI.

La opción "-lmpi" permite habilitar las directivas de MPI en el código. Con "std=c99" se le indica al compilador que se va a emplear la versión estándar C99. La opción "-Ofast" es el máximo grado de optimización que realiza el compilador al crear el ejecutable del programa. Y la opción "-lm" permite incluir la librería math.

Para enviar el ejecutable a Wilma se ha empleado la siguiente sentencia: sbatch -o resultado.txt ./script.sh nnmain.exe nprocs

En resultado.txt se almacenarán los resultados de la ejecución, a través del script.sh se envía el ejecutable "nnmain.exe" a Wilma a través de SLURM.

El argumento "nprocs" permite especificar el numero de procesos que se van a usar para la ejecución de la aplicación, en este caso se ha trabajado con 4 procesos.

## 8.2 Script SLURM & Script TAU

```
#!/bin/bash

#SBATCH –job-name=mpi_exec
#SBATCH –output=mpi_%j.txt
#SBATCH -N 1 # Num nodes
#SBATCH -n 4 # Num CORES
#SBATCH –partition=nodo.q
#SBATCH –exclusive

hostname

mpirun -np 2./1
```

```
#!/bin/bash

#SBATCH –job-name=mpi_tau
#SBATCH –output=mpi_tau.txt
#SBATCH -N 1 # number of nodes
#SBATCH -n 4 # number of processes
#SBATCH –distribution=cyclic
#SBATCH –partition=nodo.q
#SBATCH –exclusive

module load gcc/10.2.0
module load openmpi/3.0.0
module load tau/2.29
```

```
export TAU_MAKEFILE=/soft/tau-2.29/x86_64/lib/Makefile.tau-mpi
export TAU_OPTIONS=-optCompInst
export TAU_TRACE=1

tau_cc.sh -Ofast -o main nn-main.c common.c -lm
mpirun -n 4 ./main

tau_treemerge.pl
tau2slog2 tau.trc tau.edf -o tau.slog2
```

# References

[1] MPI
Technical Information [Online].Available:
https://www.open-mpi.org

[2] Bullinaria, J., Implementing a neural network in C. Available: https://www.cs.bham.ac.uk/ jxb/NN/nn.html.