Paralelización con MPI

Joan Rosell, Sergi Molina

1 Introducción

En esta práctica se nos pide continuar paralelizando el perceptrón multicapa empleando memoria distribuida. Para esto utilizaremos MPI, un estándar abierto para el paso de mensajes en aplicaciones de alto rendimiento. Para medir el rendimiento de la aplicación usaremos TAU, generando unos archivos en formato slog2 con las trazas de la ejecución. Algunos de los nuevos retos que se plantean en este proyecto son el balanceo de carga y la gestión de las comunicaciones.

1.1 Herramientas

Para la realización de la práctica, ha sido necesario el uso de varias herramientas tales como, *Valgrind*, con la que se pueden observar problemas de memoria y rendimiento del programa, *perf*, con la que se puede realizar un análisis completo de rendimiento de los programas sobre la plataforma Linux, *CMake*, la cual permite automatizar la generación de código y *SLURM*, acrónimo de Simple Linux Utility for Resource Management, con la que se ha lanzado el trabajo a los nodos del clúster. Las versiones utilizadas de cada uno de los programas ha sido la siguiente:

Herramienta	Versión
Valgrind	3.11.0
perf	3.10.0
CMake	3.13.4
SLURM	19.05.4

Como sistema de control de versiones se ha utilizado Git, sobre la plataforma GitHub.

1.2 Plataforma de cómputo

Para la realización de esta práctica se ha trabajado sobre el clúster HPC ubicado en el departamento de Ciencias de Computación. El clúster está formado por 12 nodos *Dell PowerEdge R415* en rack, los cuales montan 2 procesadores *AMD Opteron 4180*, interconectados mediante el bus HyperTransport3, 8GB de RAM y un HDD de 250GB; todos ellos funcionando con CentOS Linux.
Las especificaciones más relevantes de los procesadores integrados se muestran a continuación:

- **Frecuencia de reloj:** 2.6GHz.
- **Caché:** 3 niveles. 384KB (L1), 3MB (L2) y 6MB(L3).
- **# cores:** 6 cores, 6 hilos (no multi-threading).
- **Ancho bus:** 64 bits.
- **GPU:** Sin GPU interna.
- **TDP:** 95W.

2 Paralelización con MPI

En esta práctica vamos a trabajar con MPI sobre el código anterior paralelizando la red neuronal en un paradigma multicomputador. Varios de los retos importantes de esta práctica han venido por la importancia que juegan las comunicaciones a la hora de programar sistemas multicomputador. La versión final del programa reparte los distintos batches entre los computadores de forma que se asegura el balanceo de carga usando una política de asignación cíclica.

De nuevo, se utiliza la metodología de Foster para paralelizar el programa, detallada a continuación.

2.1 Análisis inicial de rendimiento

Cómo ya conocíamos donde estaban los cuellos de botella de la versión secuencial no vamos a detenernos volviendo a comentar lo mismo que en la práctica anterior. Si que es interesante destacar que ahora no podemos atacar el problema con la misma granularidad que antes debido a los elevados costes de comunicación que supone la sincronización de la memoria distribuida.

2.2 Propuesta de paralelización con MPI

Para paralelizar la red neuronal lo que hemos hecho ha sido dividir el trabajo **a nivel de batch**. Hemos probado a dividir el trabajo a nivel de patrón, pero los costes de comunicación crecen demasiado ya que la complejidad de la comunicación crece de forma cuadrática en vez de lineal (es decir, en vez de comunicar una vez por batch de forma lineal comunicamos N veces por batch, donde N es el número de patrones, de forma cuadrática).

2.2.1 Particionamiento

Para conseguir esto se ha repartido el bucle que reparte los batches utilizando la estrategia de cálculo de índices enseñada en clase. Al estar usando memoria distribuida tenemos el problema de que no podemos compartir de forma automática los resultados de aprendizaje de cada batch. Esto implica una sincronización entre las máquinas una vez cada proceso ha completado su conjunto de batches (sub-batches?) haciendo una media de los distintos pesos que gobiernan la ejecución de la red neuronal. Además,

para asegurarnos que el bucle de convergencia se mantiene actualizado entre los procesos también debemos sincronizar la variable de error. De nuevo hemos elegido hacer una media entre los errores locales.

Además, para evitar problemas de balanceo de carga hemos implementado una política de asignación de batches cíclica, por lo que en los peores casos (para ocho y veinticuatro procesos) el tiempo de ejecución se reduce drásticamente al balancear correctamente la carga.

2.2.2 Comunicación

Las sincronizaciones se han hecho mediante comunicación colectiva usando llamadas de la API de MPI. Como el patrón de cómputo del programa es, en el fondo, de tipo MapReduce las comunicaciones colectivas van bien ya que todos los nodos hacen el mismo trabajo. En el caso de implementar una solución donde hubiese nodos designados a tareas específicas entonces si nos sería útil emplear comunicación punto a punto.

Al repartir a nivel de batch podemos obviar las comunicacines intra-proceso, que en el caso de la práctica de OpenMP ya comentamos que eran elevadas. De esta forma, estamos explotando la localidad espacial y temporal del procesado interno de los batches al asignarlos en su totalidad al mismo proceso, evitando esa gran cantidad de comunicación necesaria para procesar cada patrón del batch.

2.2.3 Agrupación y Asignación

En lo que respecta a la agrupación y asignación ya hemos comentado que hemos decidido asignar, en la medida de lo posible, un batch por proceso. Esto se debe a que el clúster está configurado para mapear un proceso a un core y, por lo tanto, para aprovechar al máximo los recursos de cómputo hemos intentado asignar un batch por core.

2.3 Implementación del algoritmo paralelo

Para implementar nuestra propuesta hemos extendido ligeramente el bucle de convergencia y, una vez cada proceso ha terminado sus batches, se sincronizan los resultados utilizando las funciones *MPI_Allreduce* y posteriormente se calculan las medias de estos resultados, ya que en el fondo lo que hemos hecho es sumar todos los pesos y errores de cada proceso.

En cuanto a la política de asignación, simplemente se asigna el número mínimo de batches a todos los procesos (usando una división entera) y luego se gestionan los batches sobrantes de forma cíclica, asignando un batch extra empezando desde el principio hasta que nos quedemos sin batches. Como el número de batches sobrantes no puede sobrepasar al número de procesos en el peor de los casos P - 1 procesos realizaran el cálculo de un batch de más.

2.4 Resultados de rendimiento

Para evitar extendernos demasiado con esta sección comentaremos los resultados de rendimiento de nuestra versión en el mejor caso (con 38 procesos) y compararemos los resultados de la versión base que está en el Campus Virtual con la nuestra para ver cómo el balanceo de carga juega un rol vital en el rendimiento de este tipo de sistemas. Para esto último hemos seleccionado los dos peores casos de la versión base.

Como comparar los tiempos de ejecución puede no ser justo ya que el número de epochs cambia ligeramente con esta estrategia de paralelización vamos a utilizar el *throughput* para medir el rendimiento del programa. Dicho esto, estos son los resultados obtenidos:

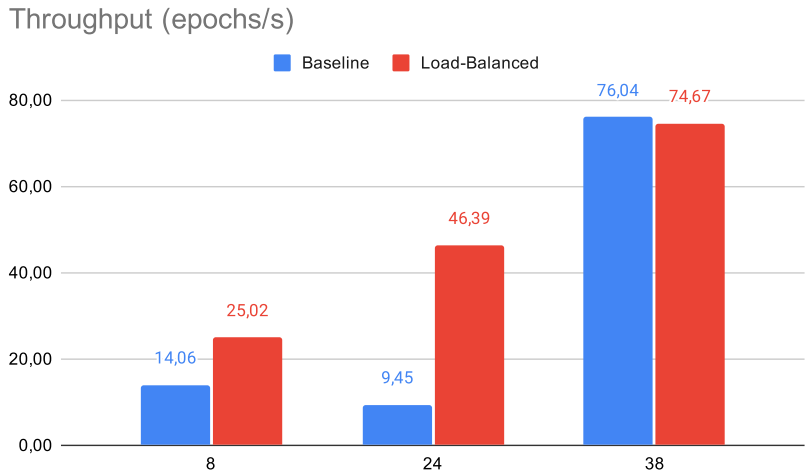

FIGURE 1. *Throughput*

En este gráfico presentamos las dos versiones en los tres casos comentados previamente. En rojo está el throughput de la versión con balanceo de carga y en azul el de la versión sin balanceo de carga. Para el mejor caso no vemos ningún resultado sorprendente, nuestra versión obtiene un *throughput* equivalente al de la versión base.

Para los casos de ocho y veinticuatro si que podemos apreciar una mejora sustancial en el *throughput*, ya que la versión con balanceo de carga logra distribuir los batches de forma equitativa entre los procesos. En el peor de los casos se distribuían 38 batches entre 24 procesos, por lo que se repartían $\frac{38}{24} = 1$ batch por proceso. Sin embargo, esta distribución deja $38 - 24 = 14$ batches sin asignar, por lo que un proceso terminará haciendo 15 batches mientras el resto solo hace 1 batch. Nuestra versión soluciona esto ya que 14 de los 24 procesos calculan 2 batches y el resto sólo 1 batch debido a la política de distribución cíclica.

Esta mejora en el balanceo de carga se traduce en un incremento del *throughput*

del $\frac{46,39}{9,45} = 4,90$, multiplicando casi por cinco el *throughput*. Para el caso de ocho procesos pasa lo mismo pero a una escala menor ya que se logran repartir de forma más o menos distribuida los batches, ya que sólo sobran 6 batches.

Veamos ahora cómo estas mejoras en el *throughput* se correlacionan con los speedups obtenidos:

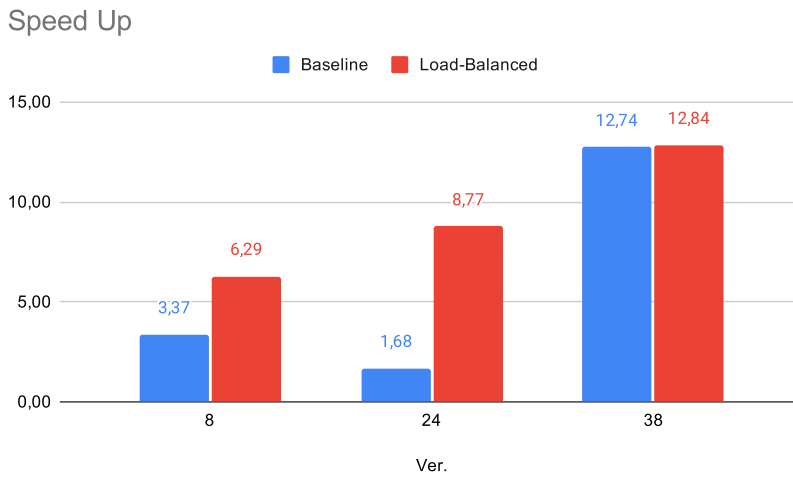

FIGURE 2. *Speed Up para 8, 24 y 38 procesos*

De nuevo, podemos ver como para el caso de 24 procesos la versión con balanceo de carga mejora significativamente los resultados de rendimiento. En concreto, el Speed Up alcanzado por la versión con balanceo de carga respecto a la versión sin balanceo de carga es de $\frac{8,77}{1,68} = 5,22$, aproximadamente la misma mejora obtenida comparando los throughputs. También podemos ver que, para el mejor caso de 38 procesos, ambas versiones son idénticas, por lo que la repartición cíclica de batches no añade un overhead apreciable en la ejecución del programa.

El resto de métricas que se pueden obtener, cómo el coste y el overhead, no las vamos a comentar ya que no aportan nueva información interesante más allá de confirmar que la versión con balanceo de carga reduce en gran medida el coste y overhead debido a que la carga queda bien distribuida. Estas métricas se pueden consultar en el apéndice, pero tal y como acabamos de comentar, no hay nada interesante que destacar.

3 Instrucciones de compilación y ejecución

Este proyecto emplea CMake con el objetivo de hacer el código más portable y simplificar el proceso de compilación. Se recomienda utilizar CMake 3.13 cómo versión mínima para poder compilar el proyecto. Además, hemos distinguido dos formas diferentes de compilar el proyecto en función de si se desea ejecutar el programa localmente o a través de un clúster SLURM.

La compilación del proyecto sigue un patrón llamado *out-of-source build*, en el que se almacenan los archivos resultantes de la compilación en un directorio específico para no mezclar estos archivos con el código fuente. Además, este directorio se excluye del VCS (Git en nuestro caso) para evitar problemas.

3.1 Ejecución/Compilación en local

En local basta con ejecutar el script de compilación llamado *compile.sh* y automáticamente este script ya hace todas las llamadas CMake necesarias para compilar y dejar listo el ejecutable. Este sería un ejemplo de su uso:

```
$ ./compile.sh
```

3.2 Ejecución/Compilación en WILMA (mediante SLURM)

Para ejecutar el programa en el clúster WILMA se debe utilizar el script llamado *submit.sh*.
A continuación, se presenta un ejemplo de uso:

```
$ ./submit.sh myFile
```

Este script genera los archivos *myFile.out/err/slog2*, lanzando como sbatch otro script que se halla en la raíz del directorio del proyecto. Los archivos **.out* contienen la salida del programa. Los archivos **.err* contienen errores encontrados durante la compilación o ejecución del trabajo. Los archivos **.slog2* contienen la información del trazado de TAU, estos archivos se pueden visualizar con Jumpshot.

Para configurar el número de nodos y procesos se utilizan las opciones nativas de sbatch, por lo que se recomienda revisar el contenido del script *submit.sh* para modificar la configuración de los recursos de ejecución en caso de que sea necesario.

Además, se pueden pasar distintos parámetros al script para configurar la ejecución del programa. Se puede encontrar más documentación al respecto en el ReadMe del repositorio de este proyecto y dentro del propio script.

4 Apéndice

Ver.	NProcs	Tiempo ej. (s)	Epochs
Secuencial	-	293,00	1148
Baseline	8	87,00	1223
	24	174,00	1644
	38	23,00	1749
Load-Balanced	8	46,60	1166
	24	33,41	1550
	38	22,82	1704

FIGURE 3. *Datos de rendimiento en crudo*

Ver.	NProcs	Throughput (epochs/s)	Throughput delta	SpeedUp	Coste (s)	Overhead (s)
Secuencial	-	3,92	-	-	-	-
Baseline	8	14,06	3,59	3,37	696,00	403,00
	24	9,45	2,41	1,68	4176,00	3883,00
	38	76,04	19,41	12,74	874,00	581,00
Load-Balanced	8	25,02	6,39	6,29	372,80	79,80
	24	46,39	11,84	8,77	801,84	508,84
	38	74,67	19,06	12,84	867,16	574,16

FIGURE 4. *Métricas procesadas*