Programación paralela: opciones y comparaciones con el algoritmo de multiplicación de matrices

Sergio Rodríguez Rubio

Universidad de Las Palmas de Gran Canaria

Índice

Objetivos

Este proyecto trata de mostrar diversas opciones para desarrollar aplicaciones paralelas de bajo nivel que hagan uso de la máxima cantidad de recursos eficientemente. Para ello se propone un algoritmo de multiplicación de matrices, alterado para ser usado en un multiprocesador, un multicomputador (cluster) y un procesador gráfico (gpu). En base a los respectivos código, y se obtendrán diversas métricas de la ejecución, que se utilizarán para comparar las diversas opciones de programación paralela y sus correspondientes eficacias a distintas cargas de trabajo.

Metodología

Herramientas

Para realizar este proyecto se han necesitado diversas herramientas, de las cuales se incluyen enlaces a las páginas principales y guías de uso en la bibliografía. Estas herramientas son Windows Subsystem for Linux (WSL), GNU Compiler Collection (GCC), Advanced Packaging Tool (APT), Nvidia Compiler (NVCC), Open Secure Shell Server (SSH) y Message Passing Interface Compiler (MPICC)..

WSL

Dado que algunos de los elementos necesarios para el proyecto requieren de un sistema Linux, se usa el WSL para poder desarrollar y ejecutar aplicaciones nativas de Linux desde Windows sin necesidad de crear máquinas virtuales o particiones en el disco.

GCC

El compilador de GNU que viene instalado en los sistemas Linux, se requiere para el correcto funcionamiento de las siguientes herramientas.

APT

El sistema de gestión de paquetes para sistemas Linux basados en Debian como Ubuntu. Es la principal herramienta usada para instalar y actualizar la mayoría de herramientas y librerías usadas en este trabajo.

NVCC

El compilador de Nvidia para programas que hagan uso de CUDA e interaccionen con la GPU mediante kernels. Requiere de GCC para su correcto funcionamiento.

SSH

Para poder conectar 2 computadoras de forma que se puedan ejecutar programas con el estándar MPI, estas deben poder realizar una conexión mediante SSH. Para esto, usamos Openssh-server, un paquete open source instalable en Linux que nos permite crear servidores SSH y conectarnos a ellos.

MPICC

El compilador de OpenMPI para programas que hagan uso de dicha librería.

Librerías

A su vez, para realizar el proyecto se han utilizado 4 librerías para obtener las métricas e implementar las distintas formas de paralelismo. Al igual que con las herramientas, en la bibliografía se encuentran enlaces referenciando a las páginas principales y guías de uso. Estas librerías son Performance Application Programming Interface (PAPI), Open Multi-Processing (OpenMP), Open Message Passing Interface (OpenMPI) y Compute Unified Device Architecture (CUDA).

PAPI

Es una librería open source que sólo se puede usar en Linux y MacOs que permite el perfilado de los programas mediante la creación de eventos y la lectura de contadores del hardware. Esto es significativamente más difícil de lograr en Windows, por ese motivo se usa Linux. Para poder usar la librería hay que compilarla con el código fuente en el computador.

OpenMP

Es una librería open source que nos permite añadir directivas de compilador con funcionalidades dedicadas a la paralelización de código mediante el uso de un mayor número de procesadores lógicos de la CPU. Esta librería viene incluida por defecto en el compilador GCC.

OpenMPI

Es una librería open source que nos proporciona una API de alto nivel a un protocolo de comunicación, el cual se puede usar tanto de manera local, entre procesos de un mismo computador, como online mediante computadores que se puedan conectar por SSH. Con OpenMPI, se crean diversos nodos, a los cuales se les asignan tareas. Lo bueno de OpenMPI es que la propia librería maneja la sincronización de los nodos. Para poder usar la librería hay que descargarse el paquete mediante APT.

CUDA

Aunque la arquitectura CUDA sea propietaria de Nvidia, la librería, al igual que las herramientas, es open source y nos permite interactuar con unos núcleos que la mayoría de GPUs de Nvidia recientes poseen. Estos núcleos se usan para paralelizar un gran número de operaciones con código similar, llamado **kernel**. Al igual que con OpenMPI, se debe instalar el paquete mediante APT.

El código

En total hay cuatro versiones del código, una versión básica sin ningún tipo de paralelismo, una versión que utiliza OpenMP, otra con OpenMPI y otra con CUDA. Todos los códigos tienen en común que utilizan la librería PAPI para las métricas, dividiendo el código en tres partes:

- La inicialización, donde se le dan valores iniciales a las matrices, se muestran en pantalla.
- El cómputo, donde se llama a la función Matriz_Matriz que será la encargada de realizar la multiplicación de matrices.
- El final, donde se imprimen los resultados.

Para tomar métricas más exactas y poder ver bien el desarrollo de los programas según la carga computacional, en la parte de cómputo, se llama a la función Matriz_Matriz 10.000 veces en bucle. Además, se ha ejecutado cada versión del código para matrices 10x10, 50x50, 100x100 y 250x250. Todos los códigos están subidos a un repositorio de github, cuyo enlace se encuentra en la bibliografía.

El código base

Del código base lo más importante es el algoritmo de multiplicación de matrices, que es el que se va a ir modificando en las siguientes versiones:

```
void Matriz_Matriz(float* x, float* y, float* z, int size) {
   for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
         float num = 0.;
         for (int k = 0; k < size; k++) {
            num += x[i*size+k] * y [k*size+j];
         }
         z[i*size+j] = num;
      }
   }
}
```

El código con OpenMP

Lo bueno de OpenMP es que al funcionar mediante directivas al compilador, la modificación del código no suele ser mucha. Cabe destacar que se utilizan dos directivas por separado, esto es para aumentar la claridad y legibilidad del código, ya que ambas directivas podrían combinarse. La primera directiva simplemente declara un bloque de código que debe ser ejecutado en paralelo, declarando con shared las variables compartidas entre hilos, y con private las variables únicas de cada hilo. La segunda directiva indica que lo que se debe paralelizar es en concreto los bucles for, con un programado estático, ya que reduce el overhead debido al organizador de hilos durante la ejecución.

```
int i, j, k;
float num;
#pragma omp parallel shared(x, y, z) private(i, j, k, num)
{
   #pragma omp for schedule(static)
   for (i = 0; i < size; i++) {
      for (j = 0; j < size; j++) {
         num = 0.;
         for (k = 0; k < size; k++) {
            num += x[i*size+k] * y [k*size+j];
         }
         z[i*size+j] = num;
      }
   }
}
```

El código con OpenMPI

Ya con OpenMPI hay que cambiar más las cosas, en primer lugar, al código de multiplicación por matrices ahora se le pasan las filas que debe ejecutar, ya que se dividirá la matriz entre los nodos disponibles. Por otro lado, el main cambia de forma que en la inicialización, el nodo maestro realiza la inicialización original y envía al resto de nodos los datos necesarios para que ejecuten su parte, mientras tanto, el resto de nodos solo deben recibir los datos proporcionados por el maestro. A su vez, en la parte final, el nodo maestro deberá recibir los datos de cada nodo e imprimirlos en pantalla, mientras que el resto de nodos deberán enviar los datos al maestro.

De esta forma, la función MatrizxMatriz se quedaría así:

```
void Matriz_Matriz(float* x, float* y, float* z, int size, int rows, int offset) {
   for (int i = offset; i < offset+rows; i++) {
      if((i+1) > size) {
         break;
      }
      for (int j = 0; j < size; j++) {
         float num = 0.;
         for (int k = 0; k < size; k++) {
            num += x[i*size+k] * y [k*size+j];
         }
         z[i*size+j] = num;
      }
   }
}
```

La inicialización en el main se quedaría de esta forma:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &processId);
MPI_Comm_size(MPI_COMM_WORLD, &processCount);
slaveTaskCount = processCount-1;
if (processId == 0) {
   Inicializar_Matrices(A, B, C, n);
   Imprimir_Inicio(A, B, n);
   rows = (n/processCount)+1;
   offset = 0;
   for(dest=1; dest <= slaveTaskCount; dest++) {
      MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(&A[offset*n], rows*n, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(&B, n*n, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
      offset += rows;
   }
} else {
   MPI_Recv(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
   MPI_Recv(&rows, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
   MPI_Recv(&A[offset*n], rows*n, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
   MPI_Recv(&B, n*n, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
}
```

Y la parte final con la sincronización sería similar a la inicialización pero el nodo maestro llama MPI_Recv y los otros nodos llaman MPI_Send.

El código con CUDA

El código con CUDA es relativamente similar al código base, lo que cambia es que la función Matriz_Matriz ahora es un kernel que se manda a cada núcleo de la GPU para calcular un solo valor de C por núcleo. También cambia el main, donde en la inicialización se debe reservar memoria en la GPU para las matrices y se deben copiar desde la memoria de CPU a la de la GPU mediante las funciones cudaMalloc y cudaMemcpy. Similarmente en la parte final, se debe copiar la matriz resultante de memoria de GPU a memoria de CPU y liberar la memoria reservada en GPU. En la parte de cómputo cambia la forma de llamar a la función, ya que ahora hay que especificar cuántos bloques de threads requerimos, y cuantos threads vamos a usar por bloque.

La función Matriz_Matriz se queda de esta forma:

```
__global__ void Matriz_Matriz_kernel(float* x, float* y, float* z, int size) {
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   float num = 0.;
   if (i < size && j < size) {
      for (int k = 0; k < size; k++) {
         num += x[i*size+k] * y [k*size+j];
      }
      z[i*size+j] = num;
   }
}
```

Datos del hardware

Los programas base, con OpenMP y con CUDA fueron ejecutados en un ordenador portátil Predator Helios PH315-51 y en el programa con OpenMPI se montó un cluster añadiendo un ordenador portátil Toshiba Satellite C55D.

Las CPUs

Como CPU principal se usó un Intel Core i7-8750H @ 2.20GHz, que posee 12 procesadores lógicos.

Como CPU adicional se usó un AMD A6-7310 @ 2.20GHz, que posee 4 procesadores lógicos.

La GPU

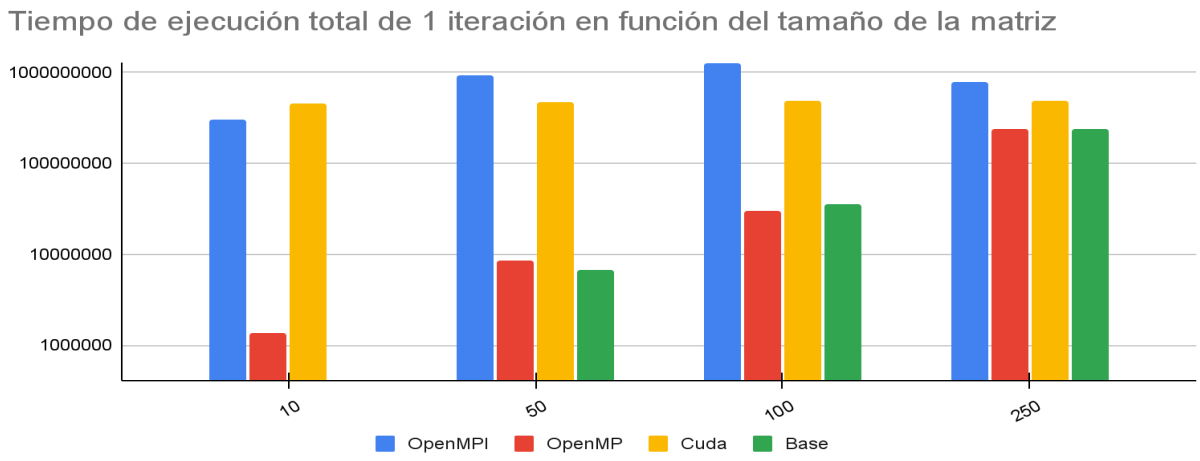
La GPU utilizada es la Nvidia GeForce GTX 1050 TI @ 1.4GHz, que posee 768 núcleos de CUDA.

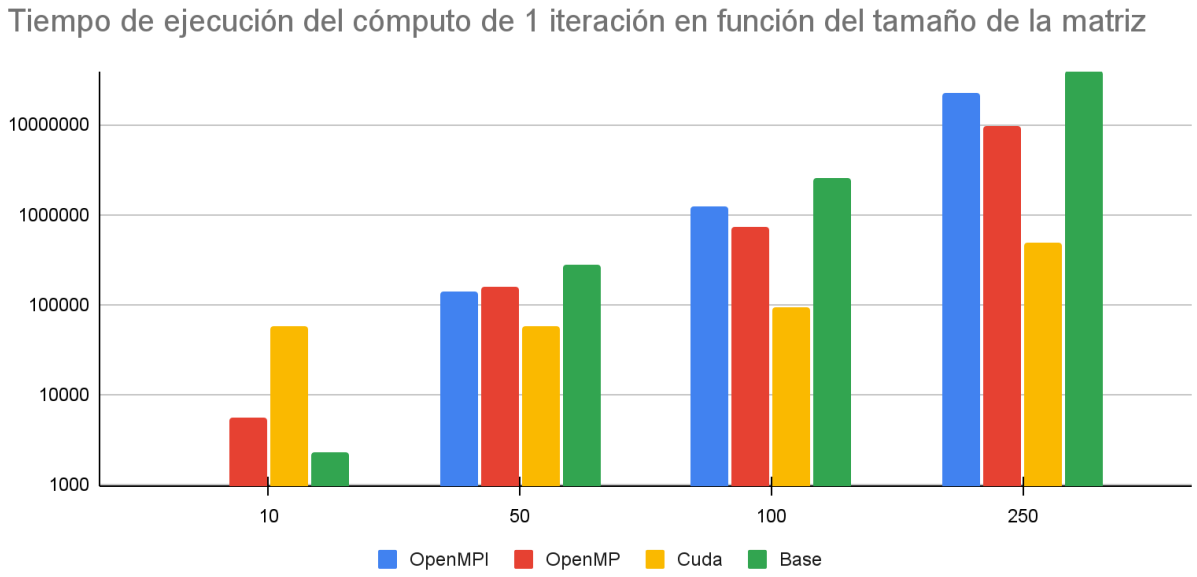
Resultados

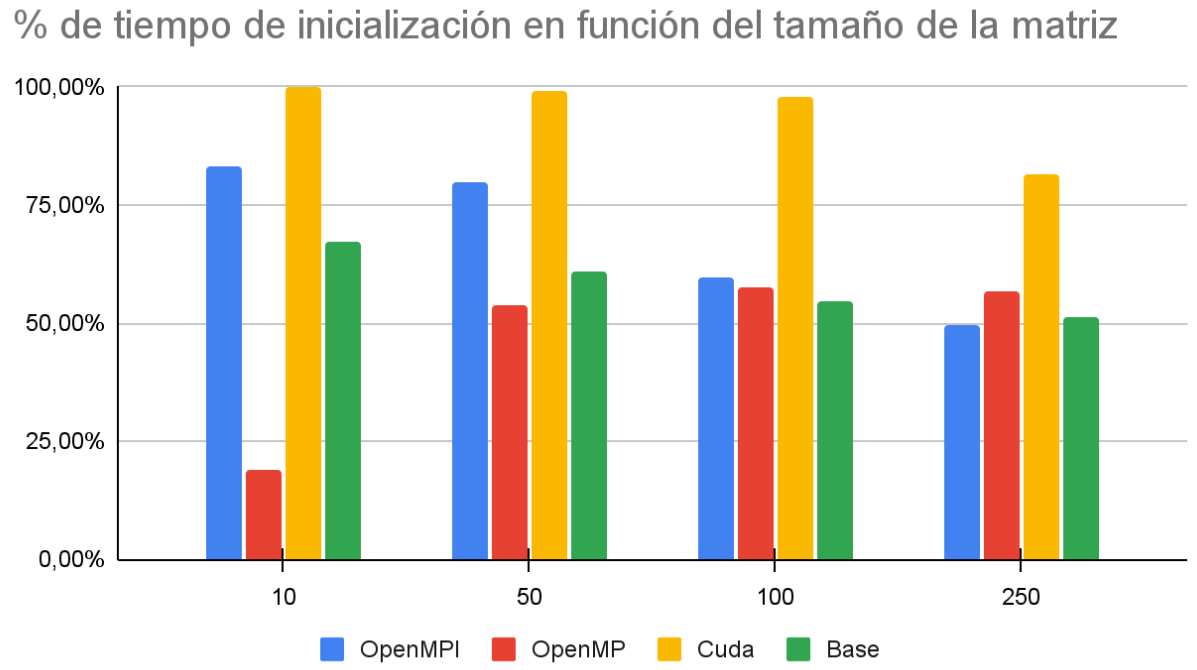
Gracias a la librería PAPI, se recogieron diversas métricas, de las cuales se ha realizado un estudio principalmente del tiempo de ejecución, pero también se han recogido datos del número de ciclos de cpu realizados y de instrucciones ejecutadas. Cabe destacar que debido a limitaciones de memoria RAM se decidió usar solo un nodo por cpu con el código de OpenMPI.

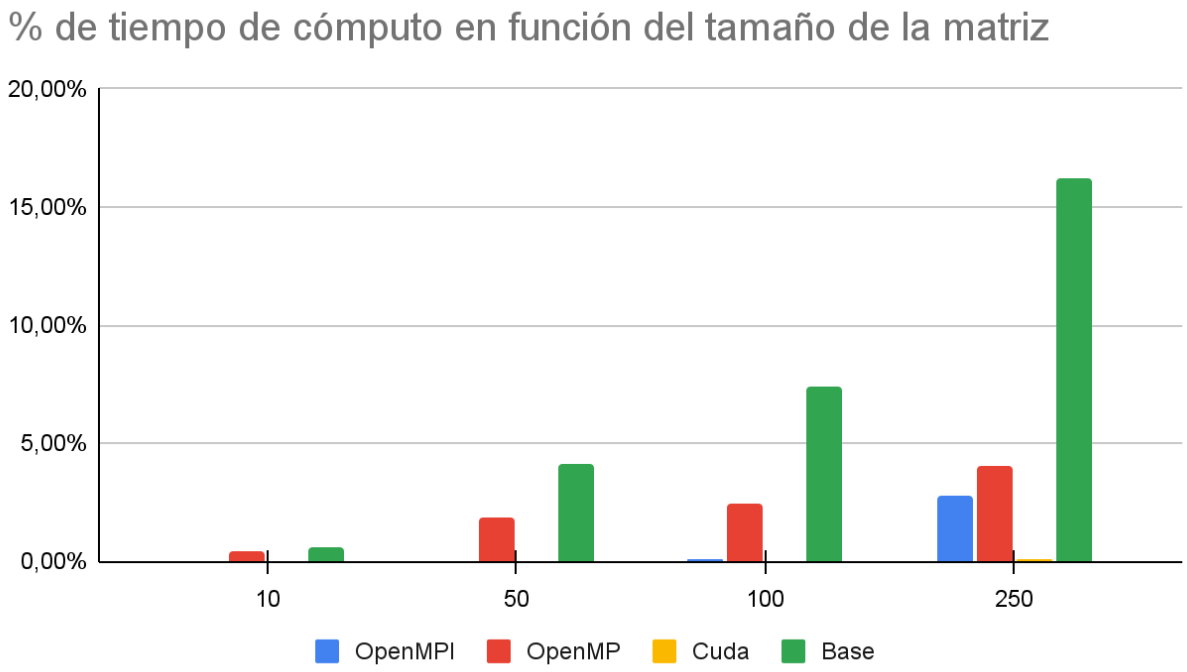
Tiempos de ejecución

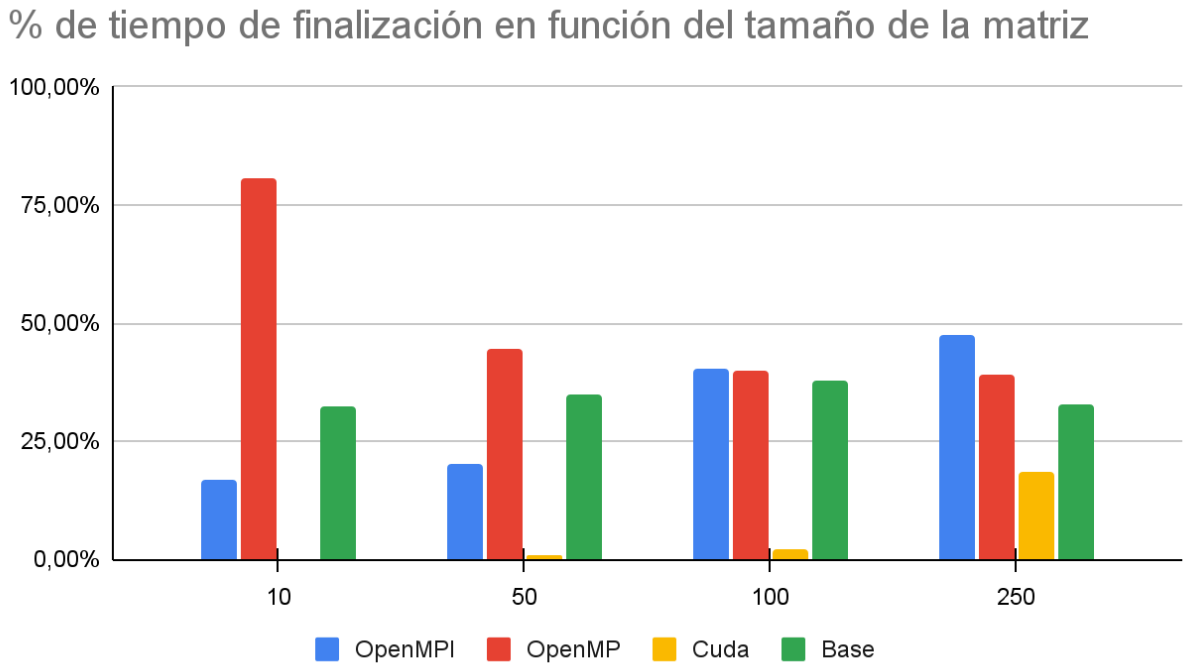
En primer lugar se comparan los tiempos (en ns) y porcentajes de tiempos de ejecución correspondientes a 1 iteración del programa en función al tamaño de la matriz. De estos tiempos podemos ver como OpenMPI y CUDA tienen un tiempo total relativamente constante. En CUDA esto ocurre porque la gran parte del programa está ocupado durante la inicialización tratando de reservar memoria en la GPU y copiar todos los valores de la CPU a la GPU. En el caso de OpenMPI se nota aún más ya que se deben sincronizar los nodos y la comunicación entre CPUs por conexión ethernet no es tan rápida como en los buses internos de cada computador. Otro detalle a anotar es el tiempo de cómputo en CUDA, que se mantiene en rangos similares de tiempo para los distintos tamaños de matrices.




Tiempo de ejecución del cómputo de 1 iteración en función del tamaño de la matriz
10000000
1000000
100000
10000
1000
10
50
100
250
OpenMPI
OpenMP
Cuda
Base


% de tiempo de inicialización en función del tamaño de la matriz
100,00%
75,00%
50,00%
25,00%
0,00%
10
50
100
250
OpenMPI
OpenMP
Cuda
Base

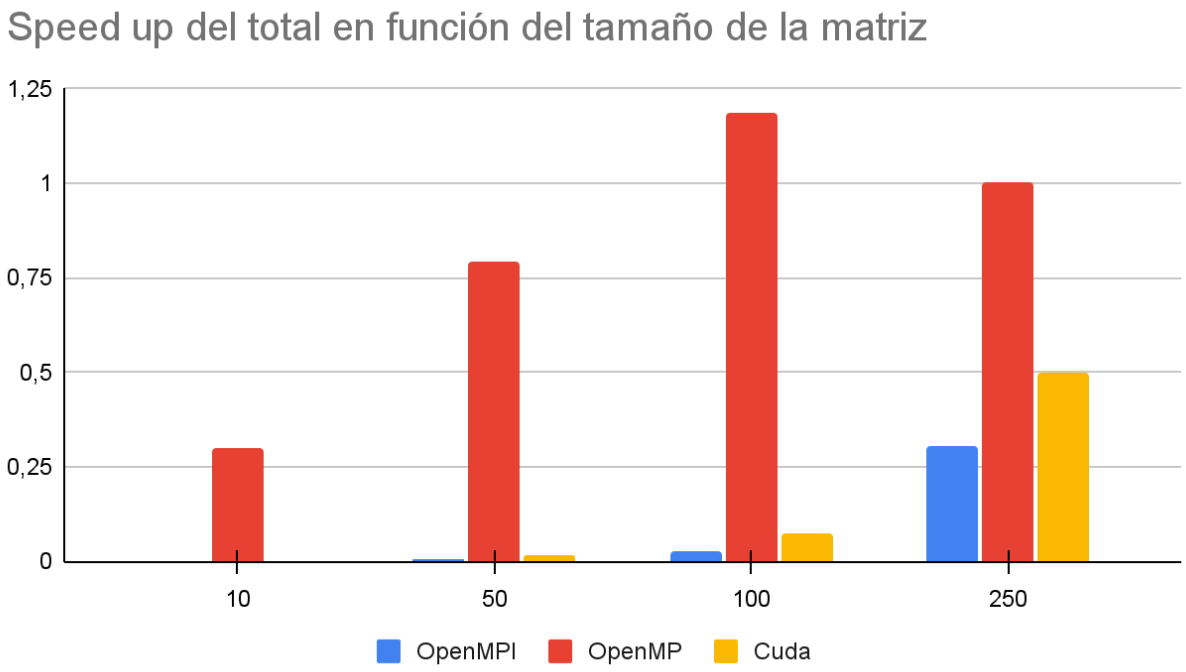

% de tiempo de cómputo en función del tamaño de la matriz
20,00%
15,00%
10,00%
5,00%
0,00%
10
50
100
250
OpenMPI
OpenMP
Cuda
Base

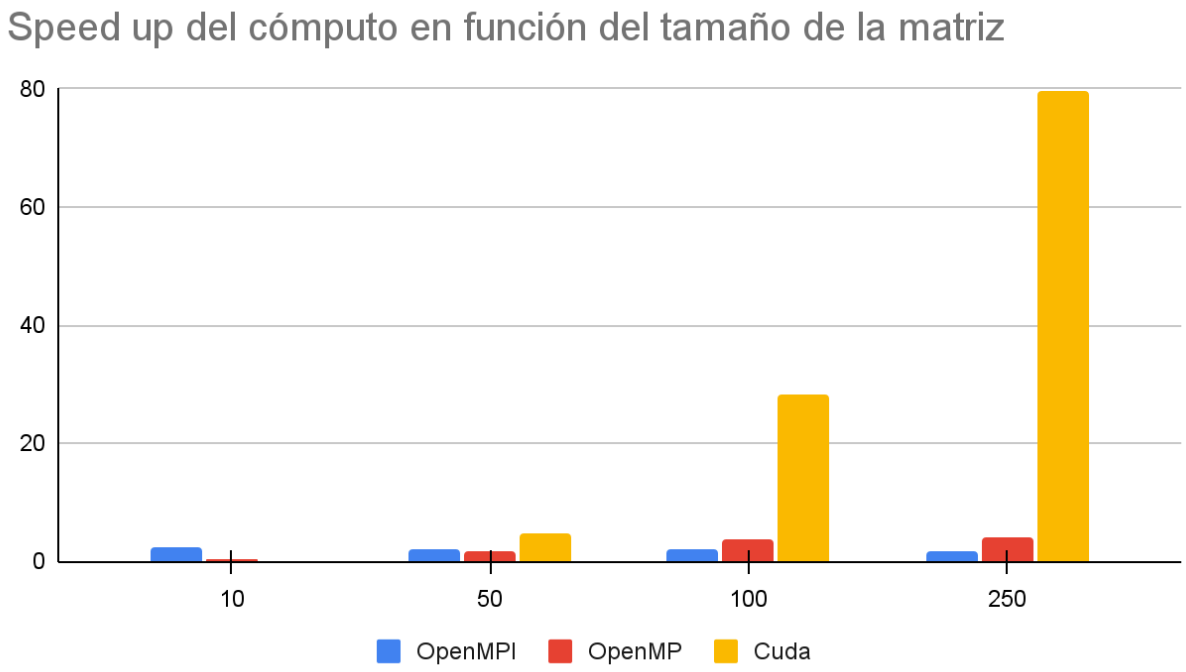

% de tiempo de finalización en función del tamaño de la matriz
100,00%
75,00%
50,00%
25,00%
0,00%
10
50
100
250
OpenMPI
OpenMP
Cuda
Base

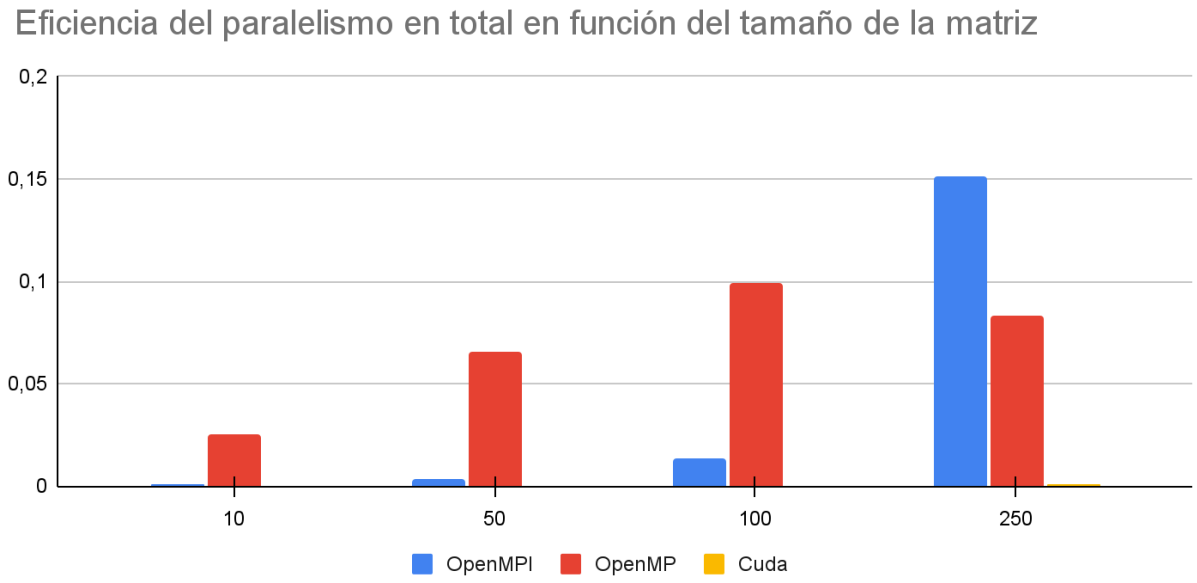
Speedups y eficiencias de los paralelismos

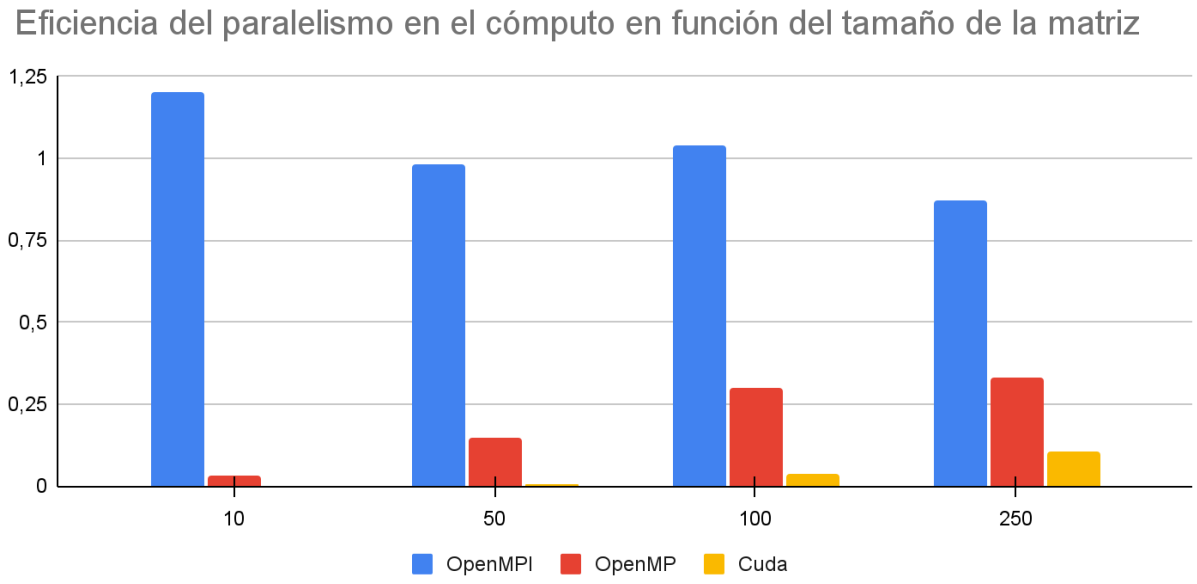
Por mucho que muestren los tiempos de ejecución, lo que se busca en el paralelismo es una mejora en función de los núcleos/nodos que uno use, es por eso que se debe analizar los speed ups y las eficiencias de los paralelismos. Es importante acentuar que la eficiencia del paralelismo para el programa con CUDA está basada en el número de núcleos de CUDA utilizados, que se obtiene con la siguiente fórmula: $\min(768, \text{ceil}(N/b)^2 * b^2$ donde N es el tamaño de la matriz (10, 50, 100, 250) y b es el tamaño de bloque utilizado por CUDA, que para el programa se puso a 5.

Con los datos obtenidos, se puede observar como los programas paralelos van viendo un mayor speedup según aumentan los tamaños de las matrices, principalmente porque el cómputo incrementa exponencialmente más rápido que las otras secciones del código y el cómputo es lo único que acelera. En concreto, el programa basado en CUDA tiene el mayor speedup del cómputo, llegando hasta un speedup de x79,64 con respecto al programa base gracias a la gran capacidad de procesado de datos de la GPU. En cambio, el programa basado en OpenMP es el que ve un mayor speedup total de x1,19 debido a que provoca un menor overhead que el resto de programas. OpenMPI es el que destaca en cuanto a la eficiencia del paralelismo en el cómputo, ya que logra dividir el cómputo entre 2 CPUs con poco overhead durante el cómputo. Por último, tanto OpenMPI como OpenMP mejoran significativamente la eficiencia de paralelismo a medida que las matrices aumentan de tamaño.


Speed up del total en función del tamaño de la matriz
1,25
1
0,75
0,5
0,25
0
10
50
100
250
OpenMPI
OpenMP
Cuda


Speed up del cómputo en función del tamaño de la matriz
80
60
40
20
0
10
50
100
250
OpenMPI
OpenMP
Cuda

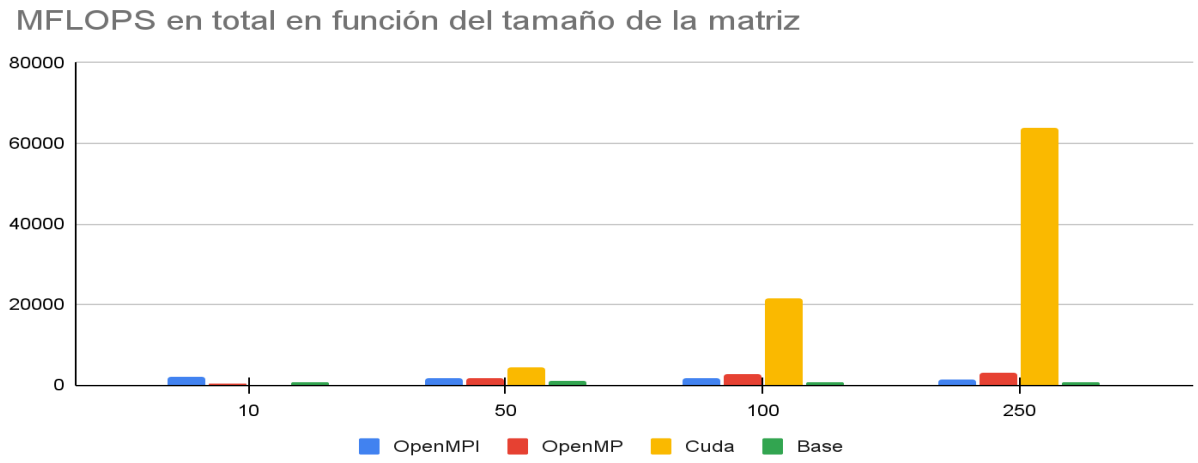

Eficiencia del paralelismo en total en función del tamaño de la matriz
0,2
0,15
0,1
0,05
0
10
50
100
250
OpenMPI
OpenMP
Cuda


Eficiencia del paralelismo en el cómputo en función del tamaño de la matriz
1,25
1
0,75
0,5
0,25
0
10
50
100
250
OpenMPI
OpenMP
Cuda

¡¡¡¡Millones de operaciones por segundo!!!!

Por último, se han analizado las operaciones por segundo realizadas por los diversos programas. La métrica es una estimación de los FLOP donde se cuentan únicamente las sumas y las multiplicaciones de tipo **float** que se ejecutan en la multiplicación de matrices, que se puede hallar con la siguiente fórmula: $2n^3-n^2$ donde n es el tamaño de las matrices. Estos FLOP son divididos entre los segundos que se tardó en realizar 1 iteración de la etapa de cómputo para obtener los FLOPS, pero como se quedan números muy grandes se van a usar los MFLOPS que son 1.000.000 FLOPS.

En comparación con el programa base, el programa con OpenMPI aproximadamente lo duplica como se esperaría de hacer uso de 2 CPUs. El programa con OpenMP empieza peor que el programa base, pero acaba llegando a realizar 8 veces más MFLOPS que el programa base aunque no sean las 12 veces más que se esperaría. El programa con CUDA es el que mayor MFLOPS tiene, más de 90 veces más que el programa base. Con 63753 MFLOPS, se puede observar por qué las GPUs se usan de manera extensa en las computaciones con grandes números de datos.



Conclusiones

Con este trabajo se han visto 3 formas de implementar programas con paralelismo mediante multiprocesadores (OpenMP), multicomputadores (OpenMPI) y GPUs (CUDA). Y tras ver los resultados está claro que hay ciertas diferencias y que cada forma tiene su caso de uso.

Empezando con OpenMP, es el ejemplo claro de paralelismo ligero y disponible con facilidad. Ya que hace un mayor uso de hilos y no crea muchos procesos nuevos, los programas son ligeros en el uso de memoria, pero siguen siendo eficientes en el uso de CPU.

Continuando con OpenMPI, aunque ha sido un área nueva de conocimiento con los clusters de ordenadores, OpenMPI demuestra su utilidad no tanto en estos programas que buscan puramente aumentar el número de operaciones, sino en programas con lógicas más complejas, que requieran de mucho cómputo de CPU. El principal problema de OpenMPI es el extenso uso de procesos nuevos, ya que consume una gran cantidad de memoria, y en este caso limitó la posibilidad de obtener una mayor cantidad de cómputo.

Por último, CUDA, que ha sido con diferencia el que mejor resultados ha dado y con razón, ya que las GPUs fueron diseñadas para este tipo de tareas. Si es cierto que tras utilizar más la librería, aumentan mis dudas de si funcionará igual de bien que el resto en programas con flujos más complejos.

En conclusión, se ha visto como el paralelismo afecta a la ejecución del programa para distintas cargas computacionales, y los grandes beneficios que aporta en programas computacionalmente intensos, pero también se a visto como en programas que no requieren de mucho cómputo, paralelizarlos hace más daño de lo que aporta.

Bibliografía

- Página de WSL
- Página de GCC
- Página de APT de Ubuntu
- Página de OpenSSH
- Página de PAPI
- Guía de instalación de PAPI
- Página de OpenMP
- OpenMPI
- Guía de cómo crear un cluster de ordenadores con Ubuntu y OpenMPI
- Página de CUDA
- Github del Proyecto