ULPGC Grado en Ingeniería informática - Arquitectura de Computadores

Programación paralela:

Comparación entre OpenMP, CUDA y MPI

ULPGC
Universidad de
Las Palmas de
Gran Canaria

OpenMP

Curso 2020/21 - Rubén Santana Lorenzo

Índice

Introducción

Dado el camino actual que sigue la informática queda claro que la paralelización puede llegar a ser una herramienta muy potente. Sin embargo, para poder explotarla adecuadamente se deben conocer profundamente las distintas herramientas y tipos de sistemas a nuestra disposición.

Partiendo de esta premisa, escogemos la **multiplicación de matrices** como ejemplar sobre el que estudiar la interfaz y el rendimiento de tres importantes frameworks: OpenMP, CUDA y MPI. Estas librerías tienen usos muy distintos pero podemos realizar un estudio sobre un hardware concreto y obtener algunas conclusiones más generales sobre las ventajas y desventajas de cada una.

OpenMP, abreviado OMP, permite programar procesos ligeros con apenas unas líneas de código, permitiendo desarrollar de forma muy eficiente aprovechando las capacidades de los procesadores multinúcleo.

CUDA es un lenguaje de alto nivel que extiende C/C++ y se puede combinar con este último sin mayor complejidad. Aporta un cierto nivel de abstracción que permiten al programador explotar los recursos de la GPU de forma simple y sin la necesidad de conocer el lenguaje concreto del dispositivo.

Por último, MPI (Message Passing Interface) es un estándar de envío de mensajes entre procesos orientado a la computación paralela. Aunque también nos permite crear hilos, a lo largo de este proyecto solo se han utilizado procesos pesados. Con MPI se pueden obtener gran capacidad de cálculo a través de la unión en red de muchos computadores que se comunican entre sí haciendo uso de esta interfaz.

A lo largo de este texto usaré una nomenclatura para referirme a las matrices que forman parte de la multiplicación: Llamaré matriz A y B a las matrices multiplicando y multiplicador y C o A*B a la matriz resultado.

Descripción de la implementación

Requisitos software

A continuación se describen las herramientas software y sus versiones utilizadas para desarrollar el proyecto (Se recomienda usar estas versiones):

- GNU Compiler Collection (GCC): Versión 11.1.0
- Performance application programming interface (PAPI): Versión 6.0.0
- Nvidia CUDA Compiler (NVCC): Versión 11.3
- OpenMPI framework (MPICC, MPIRUN): Versión 4.0.5 o 4.1.1
- Python interpreter: Versión 3.9.5 (Requisito mínimo python3)
- Pandas module for python: Versión 1.2.4*
- Numpy module for python: Versión 1.20.3*
- GNU Bash: Versión 5.1.8

Los campos marcados con '*' son necesarios únicamente para el procesamiento posterior de los datos. El resto de programas son estrictamente necesarios para la correcta ejecución del benchmark.

Organización del hardware

Comenzaré describiendo el hardware utilizado durante el trabajo. En primer lugar, disponemos de dos máquinas a las cuales me referiero como manager y predator a lo largo de este texto y en algunos ficheros de configuración del proyecto.

Manager es un ordenador de sobremesa, en este se han realizado la gran mayoría de mediciones y tiene las siguientes especificaciones:

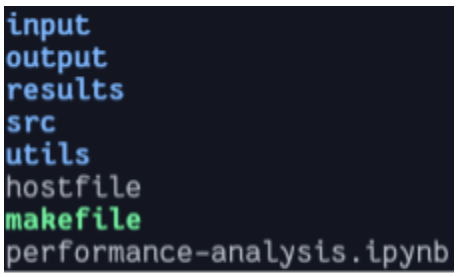
- Procesador: Intel Core i5-4460, 4 núcleos a 3.4 GHz. Un solo hilo por núcleo.
- Memoria: 8 GB de memoria DDR3
- GPU: Nvidia GeForce GTX 970, 13 multiprocesadores con 128 CUDA cores por multiprocesador.

Predator, por otro lado, es un ordenador portátil. El nombre de este nodo proviene del nombre comercial de la máquina. Se ha utilizado principalmente como “worker” en el clúster preparado para la implementación de MPI. Además se han realizado pruebas de ejecución secuencial para poder comparar su rendimiento con respecto al de manager. Estas son sus especificaciones:

- Procesador: Intel Core i7-6700HQ, 8 núcleos a 3.5 GHz. Dos hilos por núcleo.
- Memoria: 32 GB de memoria DDR4
- GPU: No utilizada.

Para la implementación de la red local necesaria para MPI se ha utilizado el switch D-Link GO-SW-5G.

Organización del repositorio



Ficheros del repositorio

El repositorio se organiza por carpetas en función de la naturaleza de los distintos ficheros. A continuación describiré en detalle cada uno de ellos.

En primer lugar, el directorio “src”, contiene todo el código fuente C. Se incluyen los ficheros de las distintas implementaciones (OpenMP, CUDA y MPI), así como dos ficheros adicionales, “matrix-utils.c”, que describe los distintos tipos de datos utilizados para representar matrices y provee funcionalidad que facilita el manejo de las mismas, y “launch.c” que actúa de módulo de control, inicializando los datos, ejecutando las rutinas paralelas y comprobando la correcta ejecución de estas últimas.

En segundo lugar, en la carpeta “utils”, disponemos de una serie de utilidades para generar ejemplares de matrices, realizar la ejecución automatizada de benchmarks y formatear los datos de salida en ficheros CSV y XLSX.

Continuando tenemos las carpeta “input”. Como su nombre implica contiene los ejemplares de las matrices utilizadas en el benchmark, estos ficheros son completamente binarios, no utilizan ningún tipo de codificación, y por lo tanto son dependientes de la arquitectura de forma que para poder ejecutar las pruebas de rendimiento adecuadamente primero se debe ejecutar el script “utils/matrix-generator.py” que genera los ficheros de datos en función de la arquitectura. El contenido de los ficheros de datos consiste de las siguientes secciones: Tres enteros, n, p y m, representando las dimensiones de las matrices dónde dim(A) = (n,p), dim(B) = (p,m) y dim(A*B) = (n,m); la matriz A; la matriz B y la matriz A*B, todas alojadas contiguamente por filas.

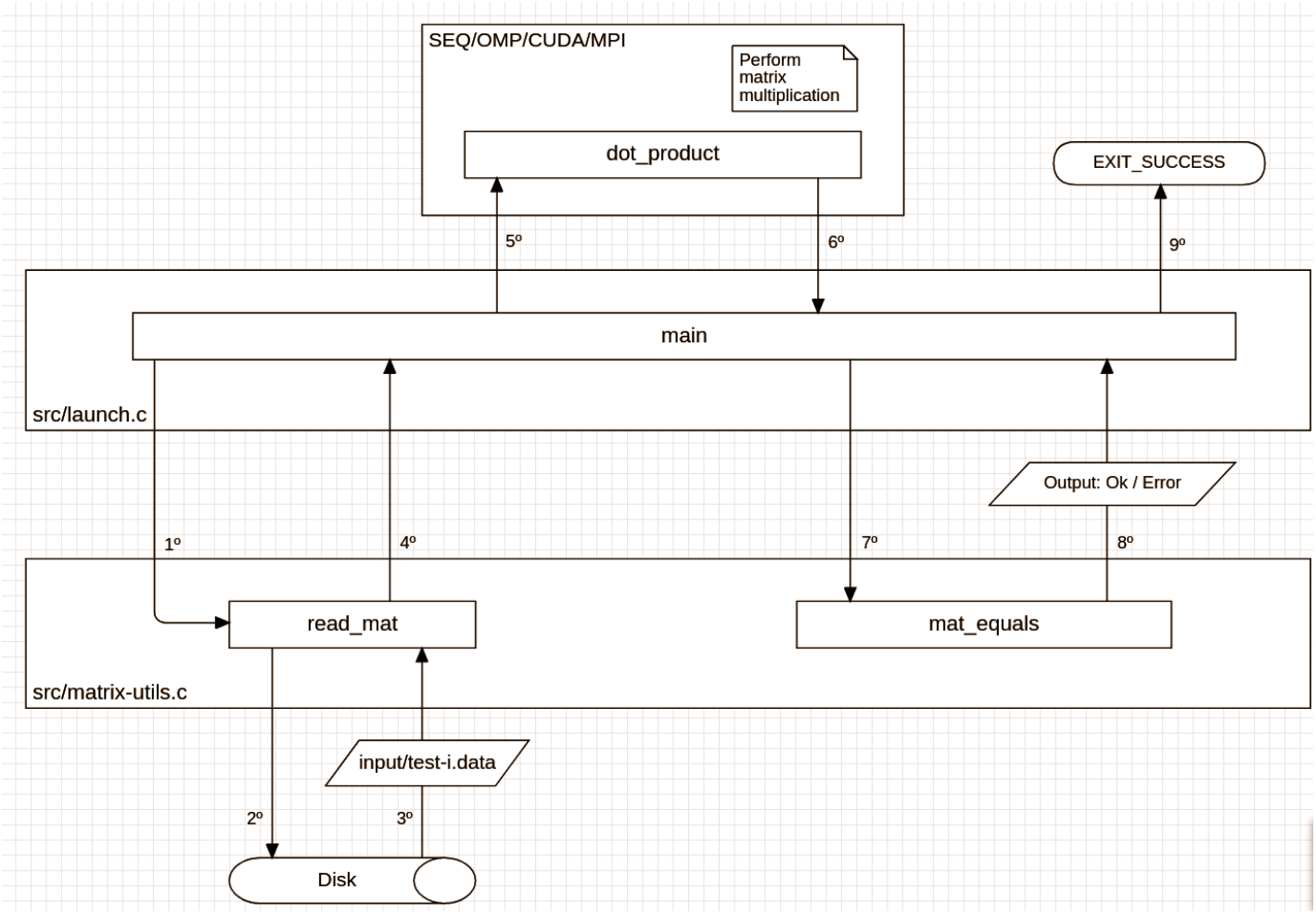
En cuarto lugar, la carpeta “output”, almacena la salida dada por la interfaz de medición de rendimiento PAPI, Performance application programming interface, que genera un fichero JSON por cada proceso pesado. Cada uno de estos archivos contiene una serie de parámetros que describen el rendimiento del programa, el más utilizado de ellos es “real_time_nsec” que representa el tiempo transcurrido en una determinada región del programa, en nanosegundos. La versión entregada está poblada con los datos generados por mi máquina

El último directorio, “results”, contiene los datos generados por la librería PAPI ya procesados en tablas con formato excel y CSV. Se incluye un script, “data_formater.py”, que permite obtener estos ficheros a partir de los ficheros del directorio “output”.

Se dispone, además, de tres ficheros adicionales. Un “Makefile” que facilita la compilación de las distintas versiones del programa, un fichero de configuración, “hostfile”, que describe la estructura del clúster MPI y un jupyter notebook, “performance-analysis.ipynb”, dónde se han llevado a cabo todo el procesamiento y representación gráfica de los resultados.

Estructura del proyecto: Flujo del benchmark

En el siguiente diagrama se modela tanto las interacciones entre los distintos módulos así como el flujo general del programa:



El fichero “launch.c” se encarga de inicializar los datos, iniciar el cómputo y, en general, dirigir el flujo del programa.

Por otro lado, “matrix-utils.c” proporciona distintas rutinas que facilitan el trabajo con matrices. Las utilizadas en el programa final son “read_mat”, que lee los ejemplares de problema del sistema de ficheros, y “mat_equals”, que compara la matriz resultado de “dot_product” con el resultado esperado en busca de igualdad.

Descripción de la implementación secuencial

Se trata de la implementación clásica del algoritmo de multiplicación de matrices. Cabe mencionar que las matrices están alojadas en el “Heap” y que la estructura “MATRIX” se utiliza para encapsular el resultado junto con sus dimensiones.

```
MATRIX dot_product(double* A, double* B, size_t n, size_t p, size_t m){
    double* res = malloc(sizeof(double)*n*m);

    for(size_t i = 0; i < n; i++){
        for(size_t j = 0; j < m; j++){
            double v = 0.0;

            for(size_t offset = 0; offset < p; offset++){
                v += *(A + i*p + offset) * *(B + offset*m + j);
            }

            *(res + i*m + j) = v;
        }
    }

    MATRIX mat = {.data=res, .n=n, .m=m};
    return mat;
}
```

Descripción de la implementación utilizando OpenMP

Una de las ventajas de OpenMP es su simpleza. Para realizar esta implementación solo es necesario añadir una directiva de precompilador al código de la versión secuencial.

La directiva permite indicar a GCC que se va a usar la extensión OpenMP para generar una región paralela, que es una sección del código que se ejecuta simultáneamente por distintos hilos. En este caso particular se añaden, además, los especificadores “for” y “collapse”.

El primero, “for”, indica que, en lugar de ejecutar todos los hilos el mismo código, las iteraciones del bucle se dividen entre todos los hilos, es decir, a cada hilo se le asignan un conjunto de valores de i para los cuales ejecuta el código del bucle. Esto implica que la unidad asignable de trabajo que entiende el programa es computar una fila, pues el índice del bucle exterior, i, indica la fila de un elemento, teniendo que calcular todos los elementos de las filas asignadas.

El segundo, “collapse(2)”, indica que el total de iteraciones de los dos bucles siguientes, es decir, todas las posibles combinaciones de i y j, se entienden como una unidad a repartir entre todos los hilos. De esta forma tenemos como unidad asignable de trabajo un elemento de la matriz, que viene definido por la tupla (i,j), en lugar de por filas como en el caso anterior. Esto permite dividir mejor el trabajo entre hilos en caso de que el número de unidades asignables de trabajo no sea múltiplo del número de núcleos disponibles evitando potenciales desbalanceos de carga.

```
18 MATRIX dot_product(double* A, double* B, size_t n, size_t p, size_t m){
19     double* res = malloc(sizeof(double)*n*m);
20 
21     #pragma omp parallel for collapse(2)
22     for(size_t i = 0; i < n; i++){
23         for(size_t j = 0; j < m; j++){
24             double v = 0.0;
25 
26             for(size_t offset = 0; offset < p; offset++){
27                 v += *(A + i*p + offset) * *(B + offset*m + j);
28             }
29 
30             *(res + i*m + j) = v;
31         }
32     }
33 
34     MATRIX mat = {.data=res, .n=n, .m=m};
35     return mat;
36 }
```

Descripción de la implementación utilizando CUDA

En el caso de CUDA he realizado dos implementaciones distintas: Una utilizando la herramienta que Nvidia provee para crear bloques y redes de bloques, “grids” con índices bidimensionales y otra sin utilizar dicha herramienta. Las llamaré “$cuda_{dim}$” y “$cuda_{ind}$” respectivamente a lo largo de este texto.

La razón por la que existen dos implementaciones es porque, originalmente, realicé la implementación sin usar bloques bidimensionales a causa de no haber asimilado completamente el concepto. Cuando realicé la segunda versión decidí que sería interesante evaluar el impacto de los operadores módulo y división, utilizados en la primera versión, sobre el rendimiento del programa.

Las dos implementaciones tienen un esqueleto común, como muchos otros programas CUDA. Este esqueleto consiste de una sección de alojamiento de espacio y envío de los datos al dispositivo CUDA (GPU) , una sección de ejecución paralela de un núcleo de cómputo y, por último, una sección de envío de resultados desde el dispositivo a memoria principal.

Es trivial que las diferencias entre ambas versiones residen tanto en la invocación del núcleo del cómputo como en el núcleo en sí mismo. Mientras que $cuda_{ind}$ mantiene un índice monodimensional para cada elemento de la matriz y obtiene la fila y columna a partir de este usando los operadores módulo y división, $cuda_{dim}$ utiliza índices bidimensionales para cada hilo y bloque que permiten triangular fácilmente la fila y columna del elemento a calcular. Por otro lado, la llamada al “kernel” se ve afectada de forma que $cuda_{dim}$ usa las estructuras “dim3” para especificar las dimensiones de un bloque y de el “grid”.

A continuación se adjuntan imágenes de ambos códigos con las diferencias más significativas señaladas dentro de rectángulos rojos:

```
extern "C" MATRIX dot_product(double* a, double* b, size_t n, size_t p, size_t m){
    double *da, *db, *dc;
    double* c = (double*)malloc(sizeof(double)*n*m);
    if(c==NULL) {
        perror("CUDA: while allocating memory");
        exit(EXIT_FAILURE);
    }

    int blocks = (n*m)/MAX_THREADS_PER_BLOCK + 1;

    cudaMalloc((void**)&da, n*p*sizeof(double));
    cudaMalloc((void**)&db, p*m*sizeof(double));
    cudaMalloc((void**)&dc, n*m*sizeof(double));

    cudaMemcpy(da, a, n*p*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(db, b, p*m*sizeof(double), cudaMemcpyHostToDevice);

    __dot_product<<<blocks,MAX_THREADS_PER_BLOCK>>>(da, db, dc, n, p, m);

    cudaMemcpy(c, dc, n*m*sizeof(double), cudaMemcpyDeviceToHost);
    MATRIX mat = {.data=c, .n=n, .m=m};

    cudaFree(da); cudaFree(db); cudaFree(dc);

    return mat;
}
```

$cuda_{ind}$: Implementación de dot_product

```
extern "C" MATRIX dot_product(double* a, double* b, size_t n, size_t p, size_t m){
    double *da, *db, *dc;
    double* c = (double*)malloc(sizeof(double)*n*m);
    if(c==NULL) {
        perror("CUDA: while allocating memory");
        exit(EXIT_FAILURE);
    }

    dim3 block_dims(THREADS_PER_DIM, THREADS_PER_DIM, 1);   /* Blocks of 32*32        */
    dim3 grid_dims((int)ceil(n/(double)THREADS_PER_DIM),    /* Blocks per grid row    */
                   (int)ceil(m/(double)THREADS_PER_DIM),1); /* Blocks per grid column */


    cudaMalloc((void**)&da, n*p*sizeof(double));
    cudaMalloc((void**)&db, p*m*sizeof(double));
    cudaMalloc((void**)&dc, n*m*sizeof(double));

    cudaMemcpy(da, a, n*p*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(db, b, p*m*sizeof(double), cudaMemcpyHostToDevice);

    __dot_product<<<grid_dims,block_dims>>>(da, db, dc, n, p, m);

    cudaMemcpy(c, dc, n*m*sizeof(double), cudaMemcpyDeviceToHost);
    MATRIX mat = {.data=c, .n=n, .m=m};

    cudaFree(da); cudaFree(db); cudaFree(dc);

    return mat;
}
```

$cuda_{dim}$: Implementación de dot_product

```cpp
__global__ void __dot_product(double* a, double* b, double* c,
                              size_t n, size_t p, size_t m){
    size_t index = threadIdx.x + blockIdx.x*blockDim.x;

    if(index >= n*m) return;

    double v = 0.0;
    size_t i = index/m, j = index%m;

    for(size_t offset = 0; offset<p; offset++){
        v += *(a + i*p + offset) * *(b + m*offset + j);
    }

    c[index] = v;
```

$cuda_{ind}$: Núcleo del cómputo

```cpp
__global__ void __dot_product(double* a, double* b, double* c,
                              size_t n, size_t p, size_t m){

    size_t i = blockIdx.y*blockDim.y + threadIdx.y;
    size_t j = blockIdx.x*blockDim.x + threadIdx.x;

    if(i >= n || j >= m){
        return;
    }

    double v = 0.0;
    for(size_t offset = 0; offset < p; offset++){
        v += a[i*p + offset] * b[m*offset + j];
    }

    c[i*m + j] = v;
}
```

$cuda_{dim}$: Núcleo del cómputo

Por la alta similitud entre el rendimiento de ambas versiones solo estudiaremos la versión $cuda_{dim}$ en la sección de análisis de los resultados

Descripción de la implementación utilizando MPI

La multiplicación usando el estándar MPI (Message passing interface), más en concreto la implementación OpenMPI, requiere, para que sea un sistema multicomputador, preparar una pequeña red local entre dos computadores. Las tareas necesarias para poner el clúster en funcionamiento son las siguientes: Incluir las direcciones IP de los dispositivos en los respectivos ficheros de configuración, habilitar una conexión segura y con autentificación automatizada (SSH) y, finalmente, montar un directorio compartido entre los nodos a través del cual se pueda compartir el ejecutable, en mi sistema este directorio se llama "cloud" y está en la carpeta HOME del usuario.

Al igual que CUDA, suministro dos versiones distintas de la implementación usando MPI. Una define la unidad de trabajo asignable como un elemento de la matriz, la otra define la unidad de trabajo como una fila. A la primera la llamaré mpi_{be} (By elements) y a la segunda mpi_{br} (By rows). La razón por la que se proveen dos implementaciones es porque es interesante ver el impacto de las dos estrategias sobre el tiempo necesario para enviar los datos y, consecuentemente, el tiempo total de ejecución.

En ambas versiones la metodología es la misma: Primero se reparte la carga computacional, ya sea por filas o elementos, entre todos los procesos disponibles, que coinciden con el número de unidades de procesamiento, y se envían los datos. En segundo lugar, se lleva a cabo el cálculo parcial de la multiplicación en cada proceso y, por último, los datos son recolectados en el hilo maestro.

Las diferencias residen entonces en la forma en la que se reparte la carga. En mpi_{be} se comparten las dos matrices de forma completa a todos los procesos colaborantes, esto supone una cierta penalización pero nos permite designar la unidad mínima de trabajo como un elemento de la matriz. Por otro lado, la metodología seguida en mpi_{br} nos obliga a definir la fila como unidad de trabajo asignable y, a cambio, podemos reducir los datos enviados a un subconjunto de filas de la matriz A y la matriz B completa.

A continuación se muestran imágenes, por simplicidad, sólo de la implementación mpi_{br}, aunque puede comprobar la versión mpi_{be} en la siguiente ruta: “src/mpi/mpi-matmul.c”.

```c
void share_work(DATA* d, int rank, int world_size, int* start, int* end){
    int dim[3];

    if(rank==dim){
        0[0] = d->n; dim[1] = d->p; dim[2] = d->m;
    }

    MPI_Bcast(dim, 3, MPI_INT, 0, MPI_COMM_WORLD);

    /* Master process */
    if(rank==0){
        int nrow = d->n;
        int base_chunk = nrow/world_size;
        int remainder =  nrow%world_size;
        int offset;
        *start = 0;

        if(remainder == 0){
            offset = *end = base_chunk;
        } else {
            offset = *end = base_chunk+1;
        }

        for(int i = 1; i < world_size; int++){
            i range[2] = {offset, offset+base_chunk};
            offset += base_chunk;

            if(i < remainder){
                range[1]++;
                offset++;
            }

            MPI_Send(range, 2, MPI_INT, i, 0, MPI_COMM_WORLD);      /* Send assigned range of rows */
            MPI_Send(d->A+range[0]*d->p, (range[1]-range[0])*d->p, /* Send assigned rows */
                     MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        }

    /* Slave process */
    }else{
        int range[2];
        MPI_Recv(range, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        *start = range[0];
        *end = range[1];
        d->n = dim[0]; d->p = dim[1]; d->m = dim[2];
        d->A = malloc((*end-*start) * d->p * sizeof(double)); /* Allocate space for assigned rows */
        d->B = malloc(d->p * d->m * sizeof(double));

        MPI_Recv(d->A, (*end-*start)*d->p, MPI_DOUBLE, 0, 0,  /* Recive assigned rows */
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Bcast(d->B, d->p*d->m, MPI_DOUBLE, 0, MPI_COMM_WORLD); /* Broadcast matrix B */
}
```

mpi_{br}: Rutina de división del trabajo y envío de datos

```c
double* __dot_product(double* a, double* b, size_t n, size_t p, size_t m, int start, int end){
    double* c = malloc(n*m*sizeof(double));
    if(c == NULL){
        perror("Error creating tmp array to contain matrix multiplication partial results");
        abort();
    }

    for(int i = 0; i < end-start; i++){
        for(int j = 0; j < m; j++){
            double v = 0.0;

            for(int offset = 0; offset < p; offset++){
                v += a[i*p + offset] * b[offset*m + j];
            }

            c[i*m + j] = v;
        }
    }

    return c;
}
```

mpi_{br}: Cómputo parcial de las filas asignadas

```c
double* collect_outcome(double* c, DATA d, int rank, int world_size, int start, int end){

    size_t csize = d.n*d.m;

    if(rank==0){
        c = realloc(c, csize*sizeof(double));
        if(c == NULL){
            perror("Not enough memory to store multiplication's outcome.");
            abort();
        }

        size_t offset = end*d.m;

        for(int i = 1; i < world_size; i++){
            MPI_Status status;
            int step;

            MPI_Recv(c+offset, csize, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_DOUBLE, &step);

            offset += step;
        }

        return c;
    }else {
        MPI_Send(c, (end-start)*d.m, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        return NULL;
    }
}
```

mpi_{br}: Recolección de los datos en el proceso maestro tras los cómputos parciales

Análisis de las prestaciones

Metodología utilizada

Para realizar el estudio del rendimiento de las distintas versiones del programa se utiliza la interfaz de programación PAPI. Esta interfaz usa contadores hardware, dependientes de la arquitectura utilizada, que nos permiten contabilizar ciertos eventos como fallos de cache o ciclos de reloj. La versión utilizada durante el desarrollo de este proyecto, 6.0.0, nos permite además obtener el tiempo transcurrido en una determinada región del programa, este último parámetro es el que se usa a lo largo del resto de esta sección. La forma en la que PAPI permite tomar medidas de una cierta región del programa es a través de las funciones PAPI_hl_region_begin y PAPI_hl_region_end que se sitúan al principio y al final de dicha región respectivamente.

Se han tomado mediciones en la región de la multiplicación de matrices, excluyendo la carga de los datos y la comprobación del resultado. En el caso de CUDA y MPI esta región incluye los tiempos de envío y recepción de datos. Además, en el caso de MPI, se han tomado mediciones explícitas del tiempo de envío y recepción de datos.

Para evaluar el rendimiento en distintos escenarios se han llevado a cabo un total de once tests con distintos tamaños de matrices. A continuación se listan las dimensiones específicas de cada test:

[250x250, 500x500, 750x750, 1000x1000, 1500x1500, 2000x2000, 2500x2500, 3000x3000, 3500x3500, 4000x4000, 4500x4500]

Cada uno de estos tests se ha llevado a cabo un total de siete veces para cada una de las versiones suministradas para asegurar la consistencia de los datos. Después las distintas instancias de cada test se han agregado usando media geométrica para favorecer los valores más comunes sobre las posibles imperfecciones de las medidas a causa del entorno de prueba.

La forma en la que se han obtenido las distintas gráficas mostradas durante la siguiente sección se han obtenido usando un jupyter notebook (python) que también se adjunta en el proyecto.

Análisis de los resultados

Empezamos por introducir los datos en una tabla para simplificar la operación sobre los mismos. Para hacer esto hacemos uso del script “utils/data_formater.py”. A continuación muestro una previsualización de dicha tabla.

		test250	test500	test750	test1	test1.5	test2	test2.5	test3	test3.5	test4	test4.5
seq	cycles	1.673387e+08	1.400288e+09	5.168005e+09	1.433770e+10	6.279546e+10	1.261305e+11	2.900260e+11	4.165637e+11	9.056659e+11	1.244813e+12	2.003902e+12
	real_time_nsec	5.232133e+07	4.378439e+08	1.615942e+09	4.483145e+09	1.963483e+10	3.943876e+10	9.068526e+10	1.302521e+11	2.831838e+11	3.892223e+11	6.265785e+11
	PAPI_TOT_INS	4.075696e+08	3.255260e+09	1.098058e+10	2.602102e+10	8.779729e+10	2.080841e+11	4.063814e+11	7.021892e+11	1.115008e+12	1.664336e+12	2.369676e+12
	PAPI_TOT_CYC	1.738307e+08	1.451016e+09	5.384591e+09	1.499463e+10	6.261429e+10	1.321494e+11	2.894857e+11	4.365239e+11	9.085283e+11	1.303387e+12	2.018170e+12
omp	cycles	4.444127e+07	3.716906e+08	1.349542e+09	3.549884e+09	1.616566e+10	3.267203e+10	7.747911e+10	1.020482e+11	2.853180e+11	3.579617e+11	6.151306e+11
	real_time_nsec	1.389178e+07	1.162130e+08	4.219572e+08	1.109934e+09	5.054658e+09	1.021551e+10	2.422608e+10	3.190734e+10	8.921292e+10	1.119246e+11	1.923386e+11
	PAPI_TOT_INS	1.061682e+08	8.466837e+08	2.851691e+09	6.756734e+09	2.279524e+10	5.402473e+10	1.055051e+11	1.823011e+11	2.894765e+11	4.320903e+11	6.152067e+11
	PAPI_TOT_CYC	4.347740e+07	3.532998e+08	1.321587e+09	3.422319e+09	1.470458e+10	3.189985e+10	7.112760e+10	9.965204e+10	2.592454e+11	3.485590e+11	5.558442e+11
cuda	cycles	2.916182e+08	5.709952e+08	1.499148e+09	3.280228e+09	1.060037e+10	2.481087e+10	4.822894e+10	8.322385e+10	1.318241e+11	1.966044e+11	2.797454e+11
	real_time_nsec	9.118645e+07	1.785693e+08	4.688333e+08	1.025828e+09	3.314976e+09	7.758596e+09	1.508095e+10	2.602294e+10	4.121771e+10	6.147379e+10	8.747216e+10
	PAPI_TOT_INS	1.377691e+08	8.879972e+08	2.877731e+09	6.599040e+09	2.228104e+10	5.399393e+10	1.054249e+11	1.841654e+11	2.961474e+11	4.495676e+11	6.387541e+11
	PAPI_TOT_CYC	7.518569e+07	4.468246e+08	1.423504e+09	3.294044e+09	1.101387e+10	2.597769e+10	5.062260e+10	8.748920e+10	1.386815e+11	2.069673e+11	2.945669e+11
mpi	cycles	4.757996e+08	1.465509e+09	3.515197e+09	6.100154e+09	1.865240e+10	3.770652e+10	7.468561e+10	1.215490e+11	2.421526e+11	4.002678e+11	5.969782e+11
	real_time_nsec	1.487698e+08	4.582320e+08	1.099129e+09	1.907385e+09	5.832204e+09	1.179002e+10	2.335260e+10	3.800579e+10	7.571595e+10	1.251553e+11	1.866627e+11
	PAPI_TOT_INS	2.018291e+08	6.024138e+08	1.564688e+09	3.083125e+09	9.930011e+09	2.255157e+10	4.379357e+10	7.560092e+10	1.242034e+11	1.950080e+11	2.816250e+11
	PAPI_TOT_CYC	9.586840e+07	3.082886e+08	8.974080e+08	1.853989e+09	7.035095e+09	1.522107e+10	3.274160e+10	4.923946e+10	1.080343e+11	1.569442e+11	2.268255e+11
mpi_br	cycles	3.551366e+08	1.098973e+09	2.648657e+09	4.789416e+09	1.539711e+10	3.206241e+10	6.607131e+10	1.085975e+11	2.261708e+11	3.817246e+11	5.714693e+11
	real_time_nsec	1.110391e+08	3.436221e+08	8.281764e+08	1.497546e+09	4.814351e+09	1.002524e+10	2.065910e+10	3.395611e+10	7.071869e+10	1.193568e+11	1.786867e+11
	PAPI_TOT_INS	4.350385e+08	8.859271e+08	1.828781e+09	3.655085e+09	1.062880e+10	2.425053e+10	4.720118e+10	8.137260e+10	1.326453e+11	2.008686e+11	2.922428e+11
	PAPI_TOT_CYC	1.627402e+08	3.521958e+08	8.881653e+08	1.871298e+09	6.492021e+09	1.412671e+10	3.027757e+10	4.532631e+10	1.022360e+11	1.764620e+11	2.348842e+11
cuda_dim	cycles	2.243211e+08	5.750426e+08	1.501570e+09	3.281782e+09	1.061658e+10	2.483884e+10	4.823284e+10	8.317207e+10	1.318842e+11	1.966660e+11	2.797856e+11
	real_time_nsec	7.013842e+07	1.798019e+08	4.695089e+08	1.026145e+09	3.319590e+09	7.766610e+09	1.508145e+10	2.600625e+10	4.123756e+10	6.149353e+10	8.748331e+10
	PAPI_TOT_INS	1.355680e+08	8.871592e+08	2.820509e+09	6.529599e+09	2.225264e+10	5.300248e+10	1.026552e+11	1.766559e+11	2.809745e+11	4.225587e+11	6.044879e+11
	PAPI_TOT_CYC	7.475563e+07	4.464020e+08	1.409520e+09	3.259254e+09	1.093097e+10	2.586892e+10	5.039167e+10	8.701954e+10	1.381268e+11	2.060745e+11	2.932216e+11

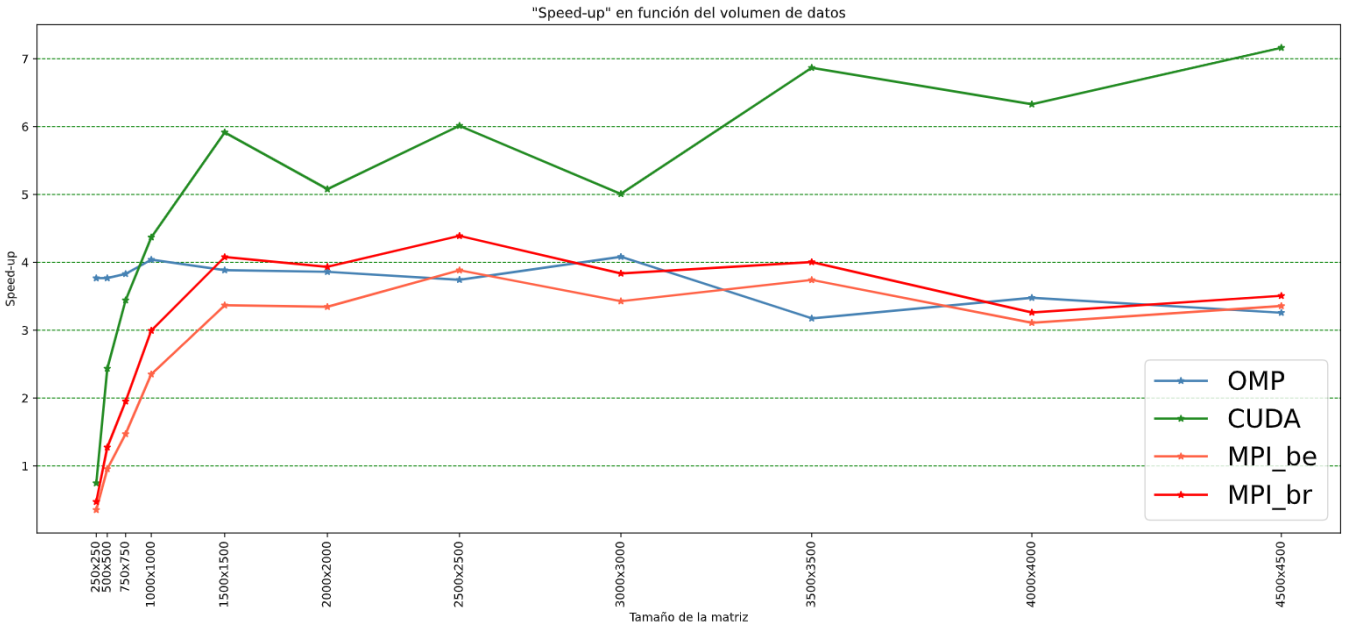
A continuación es interesante obtener el la aceleración o “speed-up” del programa para cada una de las técnicas utilizadas, este parámetro nos permite identificar qué programas han tenido mejor rendimiento para cada uno de los tests. La fórmula de la aceleración es la siguiente:

$$Speedup \ = \ tref \ / \ t_{\ i}$$

Donde $i \ \in \{omp, cuda, \ cuda_{dim}, \ mpi, \ mpi_{br}\}$ y

$$tref \ \equiv Tiempo\ de\ ejecución\ de\ la\ versión\ secuencial$$

Calculado para cada test y mostrado frente al tamaño del test ($cuda_{dim}$ no se incluye por su alto parecido con $cuda_{ind}$):



En esta gráfica se pueden observar dos comportamientos radicalmente distintos. El primero es el exhibido por OpenMP (OMP) cuya aceleración se mantiene estable en el rango [3, 4]. Por otro lado $cuda_{ind}$ y ambas versiones de MPI tienen peores speed-ups para los tests de menor magnitud y ganan rendimiento a medida que las dimensiones de las matrices se hacen más grandes.

Para poder explicar este fenómeno hay que entender la naturaleza de las distintas técnicas. OpenMP crea procesos ligeros que comparten parcialmente un mismo espacio de direccionamiento, esto evita la necesidad de copiar datos de un proceso a otro lo que produce que el coste inicial por utilizar esta técnica sea mínimo y produzca resultados similares para todas las matrices. El factor de mejora obtenido por usar OMP es relativamente proporcional al número de núcleos disponibles en el procesador, en este caso cuatro.

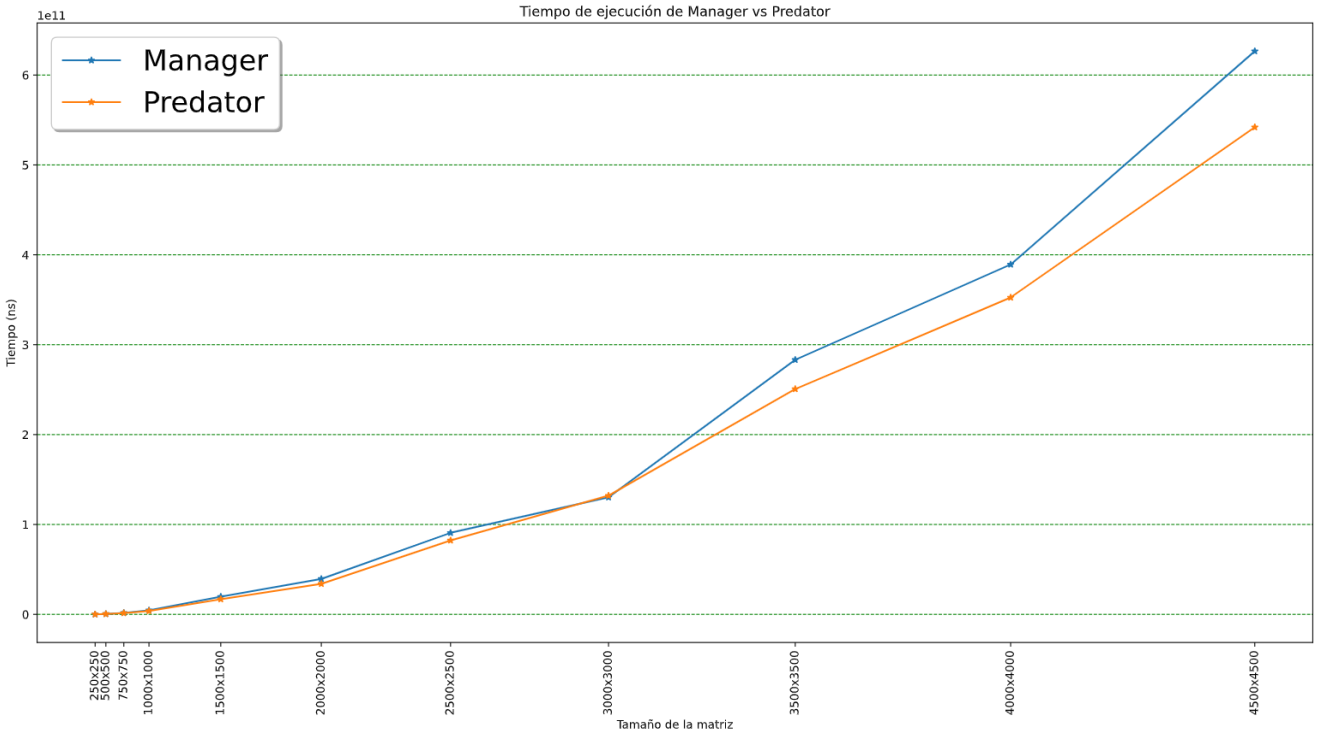
Por otro lado, CUDA y MPI tienen un coste de inicialización mucho más caro. Los programas CUDA tienen que copiar los datos de memoria principal a la memoria de la GPU antes de poder iniciar el cómputo y, finalmente, se deben transferir los resultados de nuevo a la memoria principal. Análogamente, los programas MPI tienen que enviar los datos a todos los procesos colaborantes y, después del cómputo, recolectar los resultados.

¿Con estos costes adicionales como pueden estas dos técnicas igualar o incluso superar a OpenMP? La clave reside en la capacidad de cómputo, las arquitecturas de GPU vienen preparadas para procesar eficientemente grandes cantidades de operaciones paralelizables, como en este caso, y los clúster de MPI se pueden hacer tan grandes como se requiera y con las especificaciones de cada nodo tan buenas como se necesite.

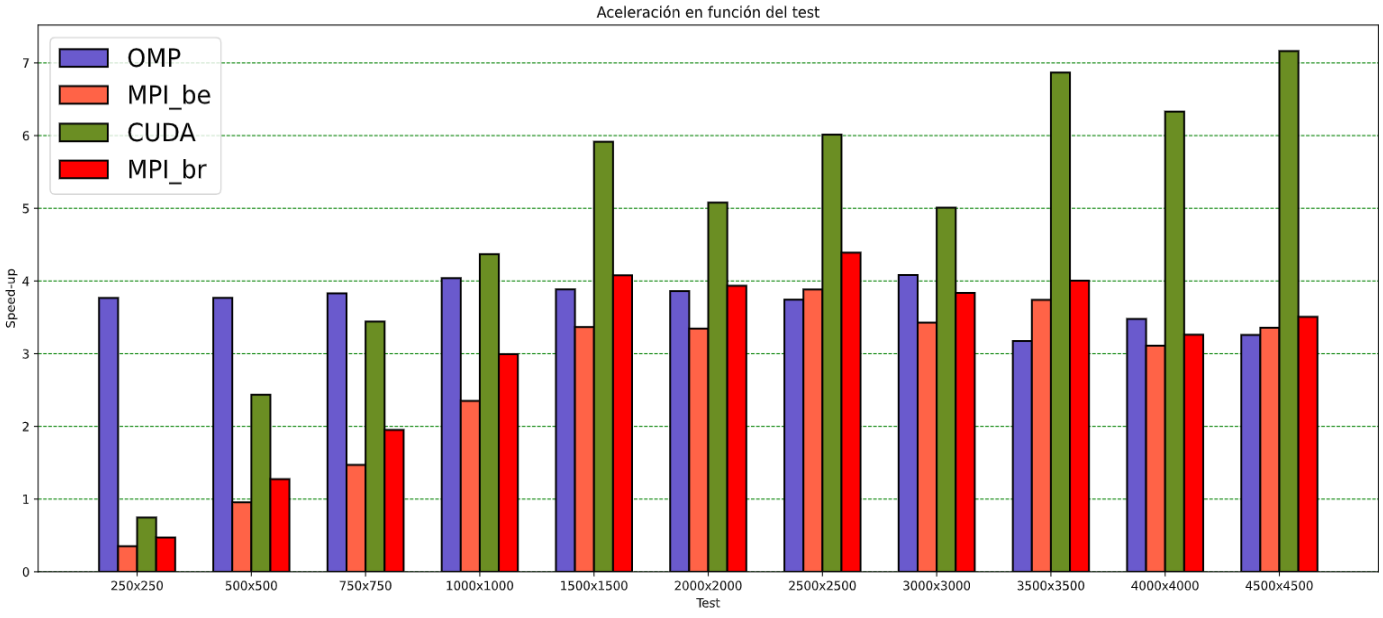
Para problemas como este que tienen complejidad $O(n^3)$ se tarda poco en observar que el tiempo de cómputo es mucho mayor que el tiempo necesario para enviar los datos, normalmente $O(n)$. Es cuando el porcentaje de tiempo total ocupado por el envío de datos se empieza a reducir que se notan las ventajas de las arquitecturas CUDA y MPI.

Sin embargo, CUDA y MPI no tienen un comportamiento exactamente igual. Por un lado el speed-up de CUDA crece más rápidamente y es considerablemente mejor en todos los tests; probablemente porque los datos solo se deben copiar una vez a través de un bus interno y la GPU tiene mayor capacidad de cómputo que el total del clúster MPI. Por otro lado, existen más factores condicionantes para MPI: En primer lugar, las interfaces de red de los distintos computadores y el switch utilizado pueden hacer cuello de botella en el envío y recepción de datos; en segundo lugar, los datos se deben copiar en el espacio de direccionamiento de cada proceso y, por último, las diferencias en las especificaciones de los procesadores puede afectar negativamente.

Como se puede ver en la siguiente gráfica, para mi sistema se demuestra que los procesadores de los dos nodos del clúster tienen capacidades ligeramente distintas lo que podría producir, al asignar equitativamente la carga total de trabajo, pérdidas de aceleración por procesos que tardan más que otros en terminar.



Adjunto, además, una versión alternativa de la gráfica del speed-up que permite comparar más fácilmente el rendimiento de los distintos programas.

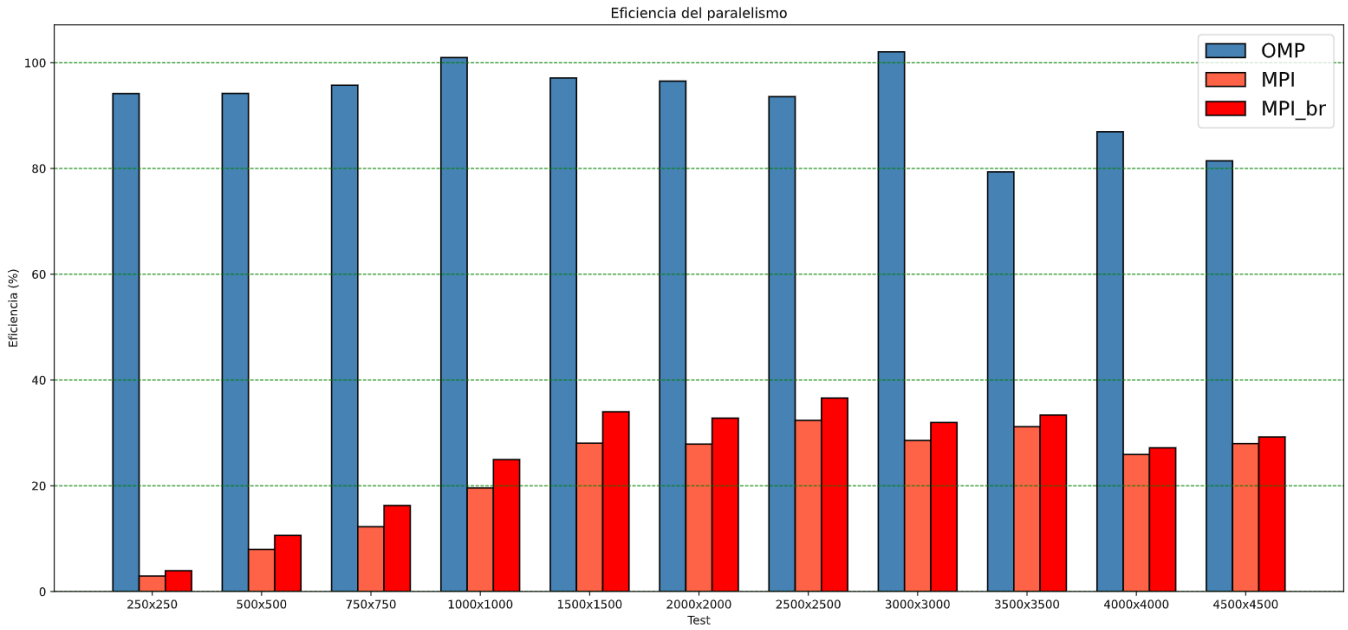


En siguiente lugar, evaluaremos la eficiencia de haber aplicado paralelismo usando OpenMP y MPI. La eficiencia del paralelismo se define como:

$$Eficiencia\ del\ paralelismo\ (\%)\ =\ speedup\ /\ N$$

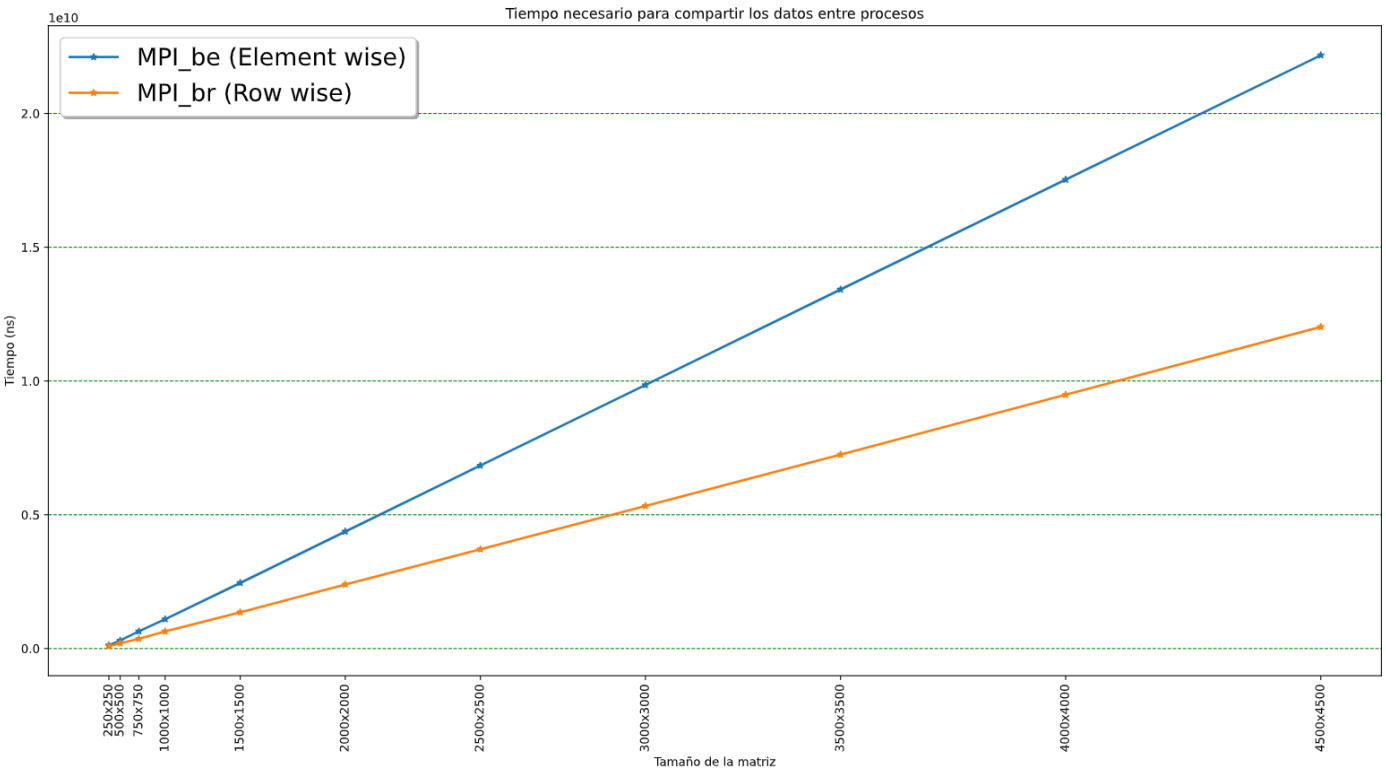
$$donde\ N\ \equiv\ Número\ de\ procesadores\ utilizados$$

Este parámetro nos permite identificar el grado de éxito que ha tenido nuestra paralelización.

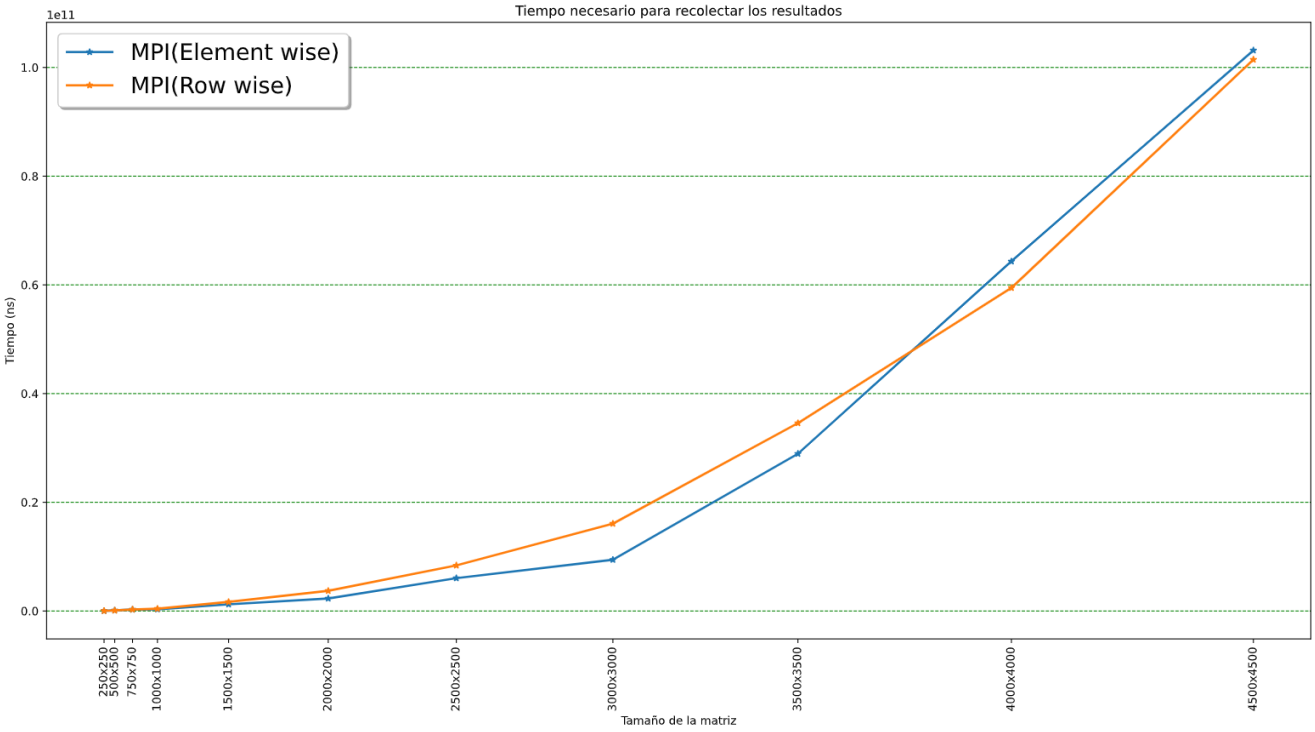


Como se puede ver el grado de éxito de OpenMP es mucho mayor que el de MPI pues su speed-up es relativamente proporcional a la inversión proporcionada (Cuatro núcleos). Por otro lado los programas MPI han tenido un peor grado de eficiencia pues su aceleración se ha visto condicionada por los factores mencionados anteriormente, no alcanzando el nivel de mejora esperado dada la inversión (Doce núcleos).

Finalmente, observaremos datos que justifican las diferencias de rendimiento entre mpi_{be} y mpi_{br}. Utilizando las mediciones del tiempo de envío y el tiempo de recolección de datos construimos las siguientes gráficas:



Se observa, como se esperaba, que el tiempo para repartir los datos es lineal y que mpi_{br} tarda significativamente menos en compartir los datos pues envía un subconjunto de filas de la matriz A frente a mpi_{be} que debe enviarla completa. Este es el motivo por el que el rendimiento de mpi_{br} es mayor, sin embargo, la mejora no es tanta como se esperaba por los otros condicionantes mencionados como el ritmo de transferencia y el desbalance de carga.



En la gráfica de recolección de resultados podemos observar que no existe apenas diferencia entre ambas versiones siendo, en promedio, mpi_{br} alrededor de un 15% mayor que mpi_{be}. La ligera diferencia de tiempo se podría atribuir al desbalance de carga y la unidad de trabajo asignable considerablemente mayor de la metodología por filas, aunque hace falta más investigación para llegar a una conclusión clara.

Por otro lado, se observa que en lugar de ser lineal, el tiempo describe una curva en función del tamaño de la matriz. Esto se debe a la forma en la que se toman las medidas pues incluye el tiempo durante el cual el proceso maestro espera a que los procesos esclavos terminen su computación, que tiene carácter cúbico. La curva en sí no tiene carácter cúbico porque el proceso maestro también realiza el cómputo de una porción de la matriz reduciendo considerablemente la cantidad de tiempo que está ocioso a la espera de que otro proceso termine.

Conclusiones

A la luz de los resultados podemos obtener algunas conclusiones que nos permitan identificar que tipo de tecnología es mejor para qué situación.

En primer lugar, OpenMP y CUDA se pueden utilizar para acelerar aplicaciones de un usuario medio que requieran de ciertas secciones de procesamiento intensivo como puede ser la edición de fotos y vídeos o los videojuegos. OMP debería priorizarse para conjuntos de datos pequeños mientras que a CUDA se le deberían asignar las cargas más pesadas. Cabe mencionar que OpenMP es más viable de implementar en una aplicación común pues no requiere de hardware especializado más allá de un procesador multinúcleo que, a día de hoy, es muy común en cualquier tipo de computador. Por otro lado, a CUDA se le pueden asignar aplicaciones más específicas de tipo científico muy intensivas en CPU y altamente paralelizables

Finalmente, el estándar MPI está muy dedicado a un tipo de aplicación especializada donde los nodos y la estructura de la red se escogen cuidadosamente para sacar el máximo partido al clúster, sin embargo, para clusters preparados con componentes no específicos el rendimiento parece verse muy afectado. Podemos concluir entonces que MPI se debería usar exclusivamente, al menos como conjunto de procesos pesados en distintos computadores, para aplicaciones de tipo científicas u otras altamente intensivas en CPU y donde sea el interés de la organización respaldante invertir en un sistema preparado y eficiente. Es importante mencionar que MPI se podría utilizar en conjunción con CUDA u OpenCL para explotar la capacidad de cómputo de un clúster con GPU, hoy día se ven algunas aplicaciones de clusters como este en el minado de criptomonedas y otros cripto objetos.

Interfaz de las utilidades suministradas

- utils/matrix-generator.py [Nombre del fichero.] [Tamaño de la matriz]

  El tamaño de la matriz es un único número que define ambas dimensiones, siempre crea matrices cuadradas con todos los elementos inicializados con el valor cinco. Un ejemplo de invocación puede ser con nombre de fichero: input/test1.data y tamaño de matriz: 1000.

- utils/generate-input.sh

  Ejecuta la utilidad anterior adecuadamente para crear los mismos ficheros de entrada usados para el análisis de resultados en este fichero.

- utils/launch_benchmark.sh

  Funcionará adecuadamente con las versiones recomendadas y habiéndose ejecutado utils/generate-input.sh previamente. Además es necesario que la carpeta compartida entre nodos del clúster mpi esté en la carpeta de usuario y se llame “cloud” y que el fichero de configuración “hostfile” esté adecuadamente configurado con los nombres y las IP de los nodos del clúster. Más información acerca de esta configuración se puede encontrar en el siguiente enlace: Running and MPI Cluster within a LAN

- utils/data_formater.py

  Funcionará adecuadamente si además de las versiones recomendadas se tiene instalados pandas y numpy para python. Generará un fichero excel y csv y los guardará en la carpeta “results” que tiene que existir previamente.

Opinión y valoración de la práctica

Es una práctica muy interesante. Me ha permitido salir del esquema habitual y explorar un tema que, hasta el momento, no está muy presente en el resto de asignaturas como es el diseño y estudio de un cierto sistema. Además me otorga una mejor perspectiva de la utilidad de la programación paralela en el mundo real y añade una nueva dimensión a mi manera de afrontar un problema de desarrollo.

Gracias a la gran comunidad alrededor de estos frameworks la mayor dificultad ha sido, muchas veces, instalar y preparar todo el entorno para poder desarrollar el programa. Aunque ha sido frustrante en ocasiones, he logrado solucionar la mayoría de problemas con relativa facilidad y aprender muchas cosas nuevas por el camino, entre ellas puedo destacar algunos conocimientos básicos sobre redes y la capacidad de instalar un programa a partir de su código fuente en lugar de usando un paquete o instalador.

Si hay algo que no me haya terminado de convencer probablemente es el guión de la práctica. Por un lado, que no incluya demasiada información me parece una buena idea porque fomenta que el alumno busque y descubra por su cuenta además de la creatividad a la hora de plantear las mediciones y evaluar los resultados. Por otro lado, creo que se podrían añadir ciertos elementos claves para facilitar la labor del estudiante como pueden ser enlaces a ciertas guías o documentación como, por ejemplo, la de PAPI que, más allá de las “man pages”, me fue imposible encontrar documentación para la versión que instalé.

Bibliografía

- Mattson and Meadows's "Hands-On Introduction to OpenMP"
- Mattson and Meadows's "Hands-On Introduction to OpenMP Exercises"
- CUDA Programming guide
- CUDA C/C++ Basic tutorial
- MPI Tutorial Introduction
- Running and MPI Cluster within a LAN
- MPI Hello World · MPI Tutorial
- MPI Send and Receive · MPI Tutorial
- Dynamic Receiving with MPI Probe (and MPI Status) · MPI Tutorial
- MPI Broadcast and Collective Communication · MPI Tutorial
- Downloading and installing PAPI
- PAPI man page
- Pandas reference page
- Numpy reference page
- Imagen del logo de OpenMP
- Imagen del logo de Nvidia CUDA
- Imagen del logo de MPI
- Imagen del logo de la ULPGC