Laboratorio OPENMP - Multiplicación de Matrices

Pedro Escobar - pedro-escobar@javeriana.edu.co

***Abstract*—Se tiene un problema y es la multiplicación de matrices cuadradas y las posibles mejoras en código y rendimiento. Se propone una solución que compara el rendimiento de los algoritmos de recorrido de filas por filas y filas por columnas. Se resuelve mediante el estudio del algoritmo y la mejora de la localidad espacial. Se obtienen resultados en materia de speedup y tiempo total de ejecución y por cada hilo.**

***Index Terms*—matriz, apuntador, paralelo, ejecucion, funcion, inicializar, OpenMP.**

I. Introducción

EL presente informe contiene los resultados de la experimentación realizada para el laboratorio de la asignatura Computación de Alto Desempeño. Dicho laboratorio tiene como objetivo analizar, mejorar y probar el algoritmo de multiplicación de matrices cuadradas, implementado en el lenguaje de programación C. El informe cuenta con secciones que se encargan de caracterizar la aplicación y el entorno de trabajo, así como la definición de los componentes que lo conforman. Adicionalmente, se elaboraron dos baterías de experimentación para los dos algoritmos de multiplicación de matrices, con el objetivo de comparar el rendimiento en materia de speedup y tiempo total de ejecución. Cada uno de los resultados ejecutados en las máquinas seleccionadas es analizado y evidenciado gráficamente, con el fin de proporcionar un entendimiento suficiente del porqué de la solución propuesta. Al final, se realizan las conclusiones correspondientes y se presentan los aprendizajes obtenidos. En [5] se puede encontrar el repositorio con el código fuente y los resultados obtenidos para cada algoritmo implementado o probado.

Noviembre 13, 2023

II. Caracterización

La presente sección determina los rasgos característicos del programa aportado para el laboratorio. El programa realiza la multiplicación de matrices cuadradas, aplicando el algoritmo clásico que multiplica filas por columnas.

A. Archivo de código

El archivo de código llamado *MM1c.c* está escrito en el lenguaje de programación C, el cual, según [1], es un lenguaje orientado a la implementación de sistemas operativos. C se caracteriza por su alta eficiencia, lo que lo ubica como uno de los lenguajes más utilizados para la programación de sistemas.

B. Librerías importadas

En el archivo se encuentran las siguientes librerías o archivos de código externo incluidos:

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>
# include "sample.h"
```

A continuación, se describe brevemente cada una de las librerías incluidas.

1) stdlib: Se trata del archivo de encabezado de la biblioteca estándar de propósito general del lenguaje de programación C, que incluye los prototipos de funciones de C relacionadas con la gestión de memoria dinámica, el control de procesos y otras funcionalidades [2].

2) stdio: Este encabezado contiene varias funciones que realizan tareas de tipo input y output (por eso el sufijo *io* en su nombre). Adicionalmente, contiene tipos de variable tales como *size_t*, *FILE*, y *fpos_t* [3].

3) omp: Este encabezado llamado *Open Multi Processing* permite la ejecución de aplicaciones con memoria compartida y programación paralela en C, C++ y Fortran.

4) sample: Este encabezado contiene funciones de otro archivo proveído por el laboratorio, llamado *Otime.c*. Dicho archivo contiene las funciones necesarias para realizar las mediciones del tiempo de ejecución del código para cada uno de los hilos. El archivo contiene las siguientes funciones:

- *Sample Start:* Se encarga de obtener el tiempo en el que inicia la ejecución de un hilo en específico. Recibe el índice que representa el hilo actual, y llama a la función *gettimeofday* la cual recupera la hora actual con una precisión de microsegundos.
- *Sample Stop:* Se encarga de obtener el tiempo en el que termina la ejecución de un hilo en específico. Realiza lo mismo que la función de inicio, cambiando el arreglo en el cual se asignan los tiempos.
- *Sample Init:* Función que inicializa las variables para la ejecución en paralelo. La función recibe el número de hilos para inicializar la variable de entorno de OpenMP al llamar la función *omp_set_num_threads*.
- *Sample Par Install:* Función que retorna la cantidad de hilos establecida para el entorno de OpenMP.
- *Sample End:* Función que recorre los arreglos que almacenan los tiempos de inicio y fin de ejecución del programa que utilice estas funciones. Los arreglos de inicio y fin son paralelos, por lo que con un solo ciclo es posible recorrerlos y restar ambos valores para obtener el tiempo de ejecución de un hilo con una precisión de microsegundos.

C. Inicialización de las matrices

La inicialización de las matrices cuadradas comienza con la recepción de los argumentos del programa cuando se inicia

su ejecución. Dichos argumentos se pueden acceder con la variable *argv*, la cual, corresponde a un arreglo doble de apuntadores. La variable *argc* se encarga de determinar la cantidad de argumentos ingresados.

Para el programa del presente laboratorio se tienen los siguientes argumentos esperados:

- Nombre del ejecutable: El nombre del archivo ejecutable es el argumento cero recibido por el programa.
- Tamaño de las matrices cuadradas: Corresponde a un número entero el cual, según el código fuente, debe ser menor que 10240. Este valor debe concordar con el límite establecido por la variable *MEM_CHUNK*, la cual determina el tamaño máximo en memoria que pueden ocupar las matrices.
- Cantidad de hilos: Corresponde a un número entero que representa la cantidad de hilos en los que se dividirá la ejecución del programa.

La siguiente porción de código extrae el valor del tamaño de las matrices y modifica las variables *argv* y *argv* reduciendo en 1 su dimensión.

```
N = (int) atof(argv[1]); argc--; argv++;
```

Una vez obtenidos los argumentos, se declaran y asignan las variables para la inicialización de la matriz. Para ello, se declara un arreglo llamado *MEM_CHUNK*, el cual se expande según la variable *DATA_SZ*, la cual determina la cantidad de espacios en memoria que serán ocupados por las matrices. El arreglo es de tipo *double*, lo que significa que permite números enteros y decimales, con un máximo de catorce dígitos en el extremo decimal, y que el tamaño de dicho arreglo se ajusta con el tamaño del tipo de dato (8 bytes).

```
# define DATA_SZ (1024*1024*64*3)
static double MEM_CHUNK[DATA_SZ];
```

La variable *DATA_SZ* se puede explicar de la siguiente forma: se delimita el espacio de memoria para que puedan ser multiplicadas dos matrices cuadradas con N = 8192, y pueda almacenarse el resultado en una matriz con el mismo N. Lo anterior significa que para evitar que el código termine con una excepción por el acceso a un campo de memoria inválido (*segmentation fault*), sólo se debe permitir un N menor o igual a 8192 o 1024 * 8. Si se desea ampliar el tamaño de las matrices se debe modificar esta variable.

Las variables para la inicialización de las matrices son las siguientes:

- variable *a*: Corresponde a la primera matriz cuadrada. Los valores se almacenan dentro del arreglo *MEM_CHUNK*, por tanto, se puede determinar que la variable *a* es un apuntador a las casillas del arreglo que corresponden a los valores de la primera matriz.

```
a = MEM_CHUNK;
```

- variable *b*: Corresponde a la segunda matriz cuadrada. Los valores se almacenan dentro del arreglo *MEM-CHUNK*. Sin embargo, para no sobreescribir los valores de la primera matriz, se debe mover el apuntador un total de N x N espacios, que corresponden a los espacios ocupados por la primera matriz.

```
b = a + SZ * SZ;
```

- variable *c*: Corresponde a la matriz resultante por la multiplicación de las matrices cuadradas *a* y *b*. Dicha matriz, por las propiedades de la multiplicación de matrices, tiene las mismas dimensiones que las anteriores. Dado lo anterior, para no sobreescribir los valores de ambas matrices, se desplaza el apuntador N x N espacios, dos veces.

```
c = b + SZ * SZ;
```

Luego de establecer los apuntadores para las matrices, se procede a llenar las matrices con valores numéricos. Para ello, se llama a la función *Matrix_Init_col*, la cual recibe cuatro parámetros:

- *SZ*: el tamaño de las matrices.
- *a*: el apuntador de la primera matriz.
- *b*: el apuntador de la segunda matriz.
- *c*: el apuntador de la matriz resultante.

La inicialización se realiza recorriendo cada celda de las matrices y asignando valores a las mismas. Para ello, el código define el siguiente ciclo:

```
int j,k;
for (j=0; j<SZ; j++) {
    a[j+k*SZ] = 2.0*(j+k);
    b[j+k*SZ] = 3.2*(j-k);
    c[j+k*SZ] = 1.0;
}
```

El ciclo recorre desde la primera columna hasta la última, llenando los valores en la primera fila de las matrices *a*, *b* y *c*. No obstante, se puede observar que sólo se están llenando los valores para la primera fila, puesto que no hay una modificación de la variable *k*, la cual conserva su valor inicial en cero. Se infiere que hay un error lógico en el código fuente proveído para el laboratorio, puesto que para inicializar una matriz, es necesario que haya dos ciclos anidados, uno para las filas y el otro para las columnas. Por tanto, se realizó la modificación correspondiente, resultando en:

```
int j,k;
for (k=0; k<SZ; k++) {
    for (j=0; j<SZ; j++) {
        a[j+k*SZ] = 2.0*(j+k);
        b[j+k*SZ] = 3.2*(j-k);
        c[j+k*SZ] = 1.0;
    }
}
```

Gracias a la corrección, ahora se puede inicializar cada matriz completamente. Para las matrices *a*, *b* y *c*, se accede a cada celda por medio de los índices del ciclo, donde *j* se desplaza por la porción de memoria que corresponde a la fila *k*, es decir, *j* recorre cada celda de la fila *k*, asignando un valor. El valor para las celdas en *a* se asigna con la multiplicación de 2 * (j + k), el valor para las celdas en *b* cambia el número entero por 3.2 y para la última matriz, inicializa todas sus celdas con el número 1.

Al haber pasado como parámetros los apuntadores a las matrices, no es necesario retornar la matriz resultante, puesto que los apuntadores modifican la dirección de memoria de la porción delimitada al comienzo de la ejecución. La función, por tanto, tiene el tipo de retorno vacío o *void*.

D. Multiplicación de las matrices

La multiplicación de matrices cuadradas elaborada en el presente laboratorio ejecuta el algoritmo clásico que multiplica filas por columnas. A continuación, se muestra el algoritmo:

```
for (i=0; i<SZ; i++)
    for (j=0; j<SZ; j++) {
        double *pA, *pB, S;
        S=0.0;
        pA = a+(i*SZ); pB = b+j;
        for (k=SZ; k>0; k--, pA++, pB+=SZ)
            S += (*pA * *pB);
        c[i*SZ+j]= S;
    }
```

Para poder comprender el algoritmo, se dividirá su análisis en tres secciones, una por cada ciclo presente.

- Primer ciclo: Este ciclo inicializa el índice *i*, el cual aumenta en uno hasta el tamaño de la matriz, o *SZ*. Dicho índice será utilizado para recorrer las filas de las matrices a multiplicar.
- Segundo ciclo: Este ciclo inicializa el índice *j*, el cual aumenta en uno hasta el tamaño de la matriz, o *SZ*. Dicho índice será utilizado para recorrer las columnas de las matrices a multiplicar.
  Adicionalmente, en este ciclo se inicializan los apuntadores que recorrerán cada celda y multiplicarán sus valores, almacenándose en una variable auxiliar.
  Para poder comprender las inicializaciones que serán hechas a continuación, es necesario entender la forma como dos matrices son multiplicadas. En la figura 7, se puede observar que cada celda de la matriz resultante es la suma de los productos de la fila *i* de la primera matriz por la columna *j* de la segunda.
  Se inicializa una variable *S*, la cual almacenará la suma de los productos para la iteración actual. Por otro lado, se inicializan dos apuntadores, *pA* y *pB*, para acceder a las celdas de las matrices *a* y *b*, respectivamente. El primer apuntador *pA* se inicializa de la siguiente forma:

  ```
  pA = a+(i*SZ);
  ```

  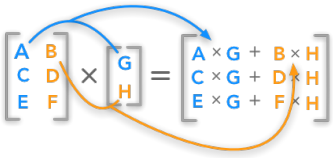

  Fig. 1: Proceso de multiplicación de matrices.

  siendo *a* el punto de partida de la primera matriz, desplazándose *i * SZ* veces en cada iteración del ciclo del índice *i* y una vez en cada iteración del ciclo del índice *k*. Con lo anterior, el apuntador es capaz de recorrer todas las celdas de la primera matriz, en un orden de tipo *row-major* o por filas.

  El segundo apuntador *pB* se inicializa de la siguiente forma:

  ```
  pB = b+j;
  ```

  siendo *b* el punto de partida de la segunda matriz en la memoria delimitada, desplazándose *j* veces en cada iteración del ciclo del índice *j*, y *SZ* veces en cada iteración del índice *k*. Con lo anterior, el apuntador es capaz de recorrer todas las celdas de la segunda matriz, en un orden de tipo *column-major* o por columnas.
- Tercer ciclo: Este ciclo se encarga de realizar las operaciones de suma y multiplicación de cada celda de las matrices y de almacenarla en la ubicación respectiva de la matriz resultante. Dicho ciclo tiene el índice *k*, el cual es ejecutado *SZ* veces. El valor de k no es utilizado ya que el acceso a las celdas de cada matriz se hace por medio de los apuntadores inicializados en el ciclo anterior.
  Cada vez que se ejecuta una iteración de este ciclo, los apuntadores *pA* y *pB* modifican su valor apuntado de acuerdo con lo explicado en el ciclo anterior.
  La variable *S* acumula los productos realizados por las celdas de las dos matrices.
  Al final de este ciclo, se regresa al segundo, donde se asigna el valor resultante a la matriz *c*, accediendo a la celda de la misma forma que se hizo en la inicialización de matrices, es decir, la celda de la fila *i*, columna *j*.

E. Ejecución en paralelo

La ejecución en paralelo se realiza con las funciones del encabezado OpenMP, en combinación con la directiva *Pragma*. Según [4], Pragma es un método especificado para el lenguaje C, que provee información adicional al compilador, más allá de lo que se especifica en el propio código. OpenMP puede realizar múltiples acciones con la directiva Pragma para separar la ejecución del código en un número de hilos determinado, organizando la ejecución en los lugares deseados, y asignando el control principal al hilo maestro.

A continuación, se explican las funciones de Pragma presentes en el código del laboratorio:

- Parallel: Ordena al compilador de forma explícita que paralelize la porción de código encerrada por este método.

Para el presente laboratorio, esta función encierra toda la ejecución de la multiplicación de matrices cuadradas, desde la inicialización, hasta la multiplicación per se.

```
#pragma omp parallel
{
    // inicializacion
    ...
    // multiplicacion
}
```

- Master: Designa una porción de código que sólo puede ser ejecutada por el hilo maestro. En el código del laboratorio, esta función permite que sólo el hilo maestro haga la inicialización de las matrices, puesto que no es necesario realizar este proceso más de una vez.

```
#pragma omp master
Matrix_Init_col(SZ, a, b, c);

Sample_Start(THR);
```

- For: Ordena al compilador distribuir las iteraciones de un ciclo entre los hilos que haya disponibles. En el código, esta función está presente encerrando los ciclos anidados que conforman el proceso de multiplicación de matrices. De esta forma, se puede paralelizar esta multiplicación, aprovechando los recursos de la máquina que ejecuta el código.

```
#pragma omp for
{
    for (i=0; i<SZ; i++)
        for (j=0; j<SZ; j++)
            ...
}
```

- Barrier: Designa un punto de sincronización en el cual los hilos de una región paralela no pueden continuar su ejecución hasta que todos los demás hilos lleguen al mismo punto. En el código del laboratorio esta función está presente dentro de la función *Sample_Start*, con el fin de que los hilos al iniciar, lo hagan al mismo tiempo.

```
void Sample_Start(int THR) {
#pragma omp barrier
  gettimeofday(...);
}
```

Para determinar la cantidad de hilos que serán ejecutados, OpenMP necesita que se establezca este número utilizando la siguiente función:

```
omp_set_num_threads (N_THREADS);
```

Con el llamado anterior, OpenMP conoce el número de hilos, y podrá dividir la ejecución en los lugares designados. Para cada hilo, este puede obtener su identificador utilizando la siguiente función:

```
int THR = omp_get_thread_num();
```

La cual es utilizada en el código para establecer el tiempo de inicio y final de la ejecución de un hilo dado.

F. Compilación del código fuente

Para poder compilar el código y generar el archivo ejecutable, es necesario que el sistema cuente con el compilador del código en C, más conocido como GCC. Dicho compilador se encuentra previamente instalado en los sistemas operativos basados en Linux. En caso de no tenerlo, debe ser instalado utilizando el comando `sudo apt install GCC`.

No obstante, para el presente laboratorio se cuenta con un archivo denominado *Makefile*, el cual posee un conjunto de palabras clave para poder automatizar el proceso de compilación. Entre las palabras claves y sus valores se encuentran:

- GCC: Es utilizado para designar el compilador del código fuente. Su valor es `gcc`.
- oT: Sirve para adicionar parámetros extra a la compilación. Su valor es `-fopenmp -O3`, lo que significa que está utilizando OpenMP para su ejecución.
- CFLAGS: Agrega banderas a la compilación. En este caso, se agrega la bandera `-lm` la cual indica que se enlaza la librería de matemáticas al código.
- oL: Vincula archivos adicionales al proceso de compilación. En el laboratorio se tiene el archivo de *Otime.c*, por lo que esta palabra clave lo agrega con el valor `Otime.c`
- BINDIR: Dirección de destino del archivo ejecutable. Para el laboratorio se tiene la dirección `../BIN/`
- PROGS: Los nombres de los programas ejecutables junto con la dirección de estos. El valor puesto es `$(BINDIR)MM1c`

Luego de determinar las palabras clave para el archivo *Makefile*, se procede a armar el comando que será ejecutado cuando se ingrese `make` desde la terminal. Para ello, se cuenta con las siguientes palabras clave: *all*, con la cual se pueden encadenar compilaciones de múltiples archivos con distintos encabezados, *clean*, con la cual se pueden eliminar los archivos ejecutables y *MM1c*, el cual corresponde a la compilación específica para el código fuente de la multiplicación de matrices proveído para el laboratorio. Dicho comando está conformado por las siguientes palabras clave:

```
$(GCC) $(oT) $(oL) $@.c \\
-o $(BINDIR)$@ $(CFLAGS)
```

G. Ejecución del programa

La ejecución del programa se realiza al ejecutar el comando `./MM1c` desde el directorio donde se encuentran los archivos ejecutables, en este caso es el directorio `../BIN/`. Para una ejecución exitosa, es necesario agregar argumentos junto al comando mostrado. Los siguientes son los parámetros solicitados:

- tamaño de la matriz: El tamaño de la matriz es un número entero no negativo, el cual debe ser menor que 1024 * 10. Este límite fue mencionado en la sección C del presente documento. No obstante, se evidencia un error en la implementación puesto que este límite no concuerda con el valor máximo establecido en memoria para las tres matrices. Esto es debido a que el valor límite *DATA_SZ* sólo aloja en memoria matrices de un N menor o igual a 8192. El error se puede observar en la fórmula *1024*1024*64*3*, donde el valor 64 es el cuadrado de 8, no de 10. Por lo tanto, para corregir este error y admitir matrices de 1024 * 10 de tamaño, debe modificarse la variable *DATA_SZ* a *1024*1024*100*3*, ya que 100 es el cuadrado de 10, lo cual concuerda con el límite. Con esta solución, se previene el error *segmentation fault* si el programa tiene como valor N un número mayor a 8192.

- número de hilos: El número de hilos que será creado para la ejecución paralela del programa. El número máximo de hilos establecido es de 20. Sin embargo, se conoce que este número de hilos no puede ser ejecutado en paralelo por todas las máquinas, puesto que algunas de ellas tienen procesadores con una capacidad mucho menor, cuando esto ocurre, los hilos que no se pueden ejecutar en paralelo se ejecutan en secuencia, una vez un hilo termine de ejecutarse.

  Para valores de N pequeños, el procedimiento de multiplicación de matrices tarda menos de un segundo en ejecutarse completamente. No obstante, entre mayor sea el tamaño de la matriz, más operaciones se realiza, resultando en tiempos de ejecución mucho más grandes. Es por esta razón que se propone realizar una optimización al algoritmo.

III. Primera batería de experimentación

La batería de experimentación para el algoritmo *MM1c* es la siguiente:

- número de hilos: variar la cantidad de hilos en 1, 2, 4, 8, 10, 14, 16 y 20.
- tamaño de la matriz: variar el tamaño en 100, 200, 400, 600, 800, 1000, 1500 y 2000.

La ejecución se llevó a cabo en tres máquinas diferentes, para probar la capacidad de cada una en cuanto a la ejecución del código. Las máquinas son:

- Ubuntu 22.4, ocho procesadores. La máquina cuenta con ocho procesadores Intel i7 de doceava generación, y una memoria RAM de 8 gigabytes y de 4800 MHz de capacidad.
- Windows 11, dieciséis procesadores. La máquina cuenta con dieciséis procesadores Intel i7 de doceava generación, y una memoria RAM de 8 gigabytes y de 4800 MHz de capacidad.

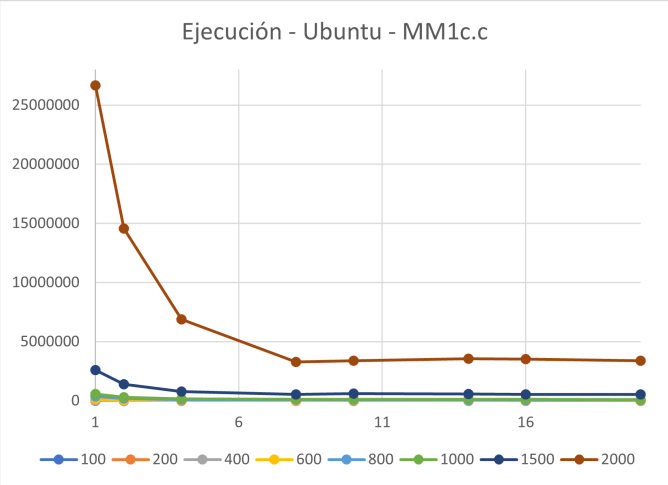

Fig. 2: Resultado de MM1c en Ubuntu

A. *Ejecución máquina Ubuntu*

A continuación se describen los resultados para la primera máquina:

Como se puede observar en la figura, la ejecución del programa muestra la mayor diferencia de tiempo cuando se ejecuta con un sólo hilo. Esto ocurre debido a que un hilo ejecuta toda la multiplicación de matrices, por tanto, consume todo su tiempo de ejecución. A medida aumenta la cantidad de hilos, se observa una reducción drástica en el tiempo, siendo los números dos y cuatro los que reducen sustancialmente el tiempo de ejecución. No obstante, es destacable que a partir de los ocho hilos, el programa llega a un punto de equilibrio, puesto que no mejora el tiempo de ejecución.

El tiempo de ejecución con más de ocho hilos no mejora, esto puede deberse a las siguientes razones:

- El tiempo que consume la paralelización del algoritmo es casi igual al tiempo de mejora de ejecución del algoritmo: Esto significa que al dividir más las tareas, el proceso de división tiene un tiempo t, el cual es similar al tiempo de ejecución de un hilo.
- La distribución paralela del código hecha para el laboratorio tiene esta capacidad máxima de eficiencia.
- La máquina tiene un número de procesadores máximo de ocho, por tanto, no es posible optimizar más el algoritmo debido a las limitaciones físicas del sistema. Esta puede ser la razón más importante, puesto que al haber más hilos que procesadores, estos ejecutan el proceso una vez haya uno libre, entonces el tiempo promediado no mide exactamente el tiempo total, el cual debería ser dos veces el tiempo con 8 procesadores para el caso de 16 hilos, y 3 veces el tiempo con 8 procesadores para el caso de 20 hilos.

En la tabla se pueden observar los valores en microsegundos, que determinan el tiempo de ejecución para la máquina *Ubuntu*. Se observa un incremento en el tiempo entre mayor es el tamaño de la matriz.

	1	2	4	8	10	14	16	20
100	411.43	158.08	88.47	709.03	1664.86	496.82	514.42	713.27
200	3127.8	1555.64	826.05	1298.91	1336.69	1217.33	1165.18	1236.96
400	33280.67	17224.83	9165.08	12751.96	13676.86	9180.06	9803.77	8526.41
600	115076.3	58354.92	31734.65	35075.67	38401.14	23833.81	23793.3	21473.65
800	372400.33	185186.2	99538.96	86099.25	86586.11	70318.87	69022.77	64512.13
1000	573742.8	287822.7	154784.9	131619.4	132077.5	114928	106743.4	104233.6
1500	2591813.37	1387011	782440.7	543869.1	615492.1	568939.6	532486.1	535002.9
2000	26665157.3	14551988	6862504	3292444	3366775	3562335	3517405	3372490

Fig. 3: Resultados de MM1c en Ubuntu

B. Ejecución máquina Windows

A continuación se describen los resultados para la segunda máquina:

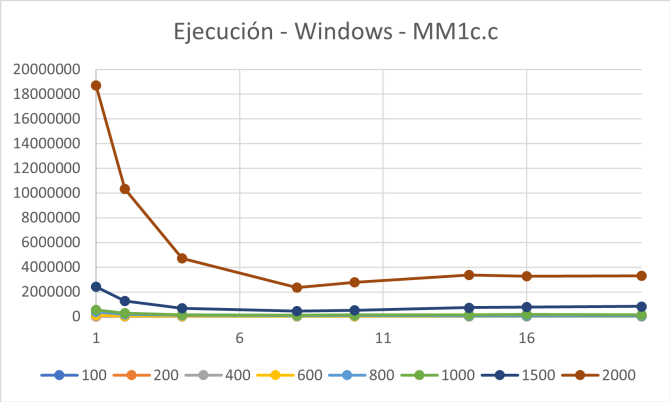

Fig. 4: Resultado de MM1c en Windows

Como se puede observar en la figura, la ejecución del programa muestra la mayor diferencia de tiempo cuando se ejecuta con un sólo hilo al igual que la ejecución en el sistema Ubuntu. Se observa un comportamiento muy similar, pero los tiempos en general son más rápidos, siendo el tiempo ocho segundos más rápido en Windows para su peor caso, comparado con Ubuntu. El peor tiempo de Windows fue de 18 segundos, mientras que el de Ubuntu fueron 26 segundos. Esta mejora puede deberse a que se cuenta con un mayor número de procesadores, o porque la ejecución de Ubuntu era realizada con una máquina virtual, lo cual puede reducir la capacidad de procesamiento de los procesadores.

Adicionalmente, se observa que luego de llegar a 8 hilos, la ejecución del programa no mejora, por tanto es posible deducir que no se debe a la cantidad de procesadores, puesto que este sistema tiene dieciséis, el doble que los de la máquina anterior.

	1	2	4	8	10	14	16	20
100	533.57	349.54	310.85	235.26	201.8	206.69	227.94	245.11
200	4006.47	2254.6	1460.94	1123.14	1045.44	684.17	828.12	568.38
400	32833.67	15863.36	11545.53	13097.69	10761.9	9968.61	8529.22	8782.98
600	112366.4	57048.02	32294.93	33712.24	28675.62	23251.51	22474.21	20035.51
800	368772.9	183432.6	99299.36	78005.68	68247.21	59466.7	55763.62	51541.82
1000	556301	276367.6	146913.9	125329.9	129299.2	157144.1	171456	154940.9
1500	2397380	1264732	665187.5	431619	506164.5	720166.4	782698.5	817962.8
2000	18699529	10327208	4700496	2342002	2780570	3366797	3261338	3306168

Fig. 5: Resultados de MM1c en Windows

En la tabla se pueden observar los valores en microsegundos, que determinan el tiempo de ejecución para la máquina *Windows*. Se observa un incremento en el tiempo entre mayor es el tamaño de la matriz, lo cual es esperable.

IV. Aplicación del algoritmo con recorrido de filas

Con el fin de mejorar la localidad espacial del algoritmo *MM1c*, es posible cambiar la forma como los apuntadores que acceden a las celdas de las matrices se desplazan, incrementando la localidad espacial, puesto que estarían accediendo al mismo bloque o a bloques más cercanos que los del primer algoritmo. El siguiente es el algoritmo resultante, optimizando el acceso de los punteros, el cual ahora es filas por filas:

```
for (i=0; i<SZ; i++){
    double *pA = a+(i*SZ);
    for (j=0; j<SZ; j++) {
        double *pB;
        pB = b + (j*SZ);

        for (k=SZ; k>0; k--, pB++)
            c[i*SZ+j] += (*pA * *pB);
        pA++;
    }
}
```

En el código se puede observar que se inicializa el apuntador *pA* de la misma forma que en el código de filas por columnas. No obstante, esta se asigna en el primer ciclo, para poder reiniciar su valor cada vez que cambia el índice *i*. Adicionalmente, este valor cambia de a uno cada vez que el tercer ciclo termina todas sus iteraciones, de esta manera se conserva la localidad para la matriz *a*. Para la segunda matriz, su recorrido es optimizado al asignar *pB* de la misma forma que el primer apuntador, recorriendo filas por filas.

No obstante, el cambio más importante es en la asignación de los valores de las celdas, puesto que ya no se tiene un valor auxiliar, sino que se modifica el valor de la matriz resultante, puesto que al modificar la posición del apuntador de la primera matriz, ahora es posible volver a acceder a la misma posición varias veces. Se incrementa la localidad espacial con este algoritmo, puesto que ambos apuntadores cambian su valor de a uno, en vez de dar saltos que ignoran el contenido enviado por un bloque al procesador en el caché.

El resto del código funciona exactamente igual. A continuación se muestran los resultados obtenidos para las máquinas seleccionadas.

V. Segunda batería de experimentación

La batería de experimentación para el algoritmo *MM1f* es la misma que para la del algoritmo de la primera batería de experimentación. Se utilizan las mismas máquinas y las mismas configuraciones para su ejecución. Por consiguiente, se procede a mostrar los resultados obtenidos para cada una.

A. *Ejecución máquina Ubuntu*

A continuación se describen los resultados para la primera máquina:

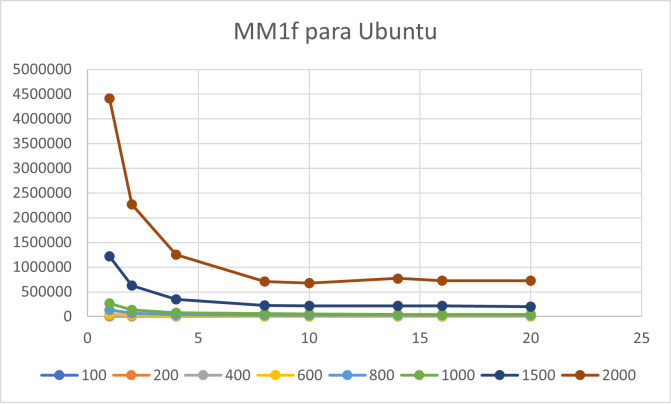

Fig. 6: Resultado de MM1c en Ubuntu

Como se puede observar en la figura, la ejecución del programa muestra una mejora sustancial en la ejecución del código. Ahora la multiplicación de 2000 de tamaño de matriz tiene una ejecución de 4.5 segundos, comparado con los 26 que había en el primer algoritmo de filas por columnas. La localidad espacial en el código se ve reflejada en los tiempos de ejecución de cada hilo, puesto que cada apuntador sólo incrementa en uno, y no en una porción completa de una fila o columna.

	1	2	4	8	10	14	16	20
100	279.13	149.12	99.87	143.77	4066.43	560.51	1662.14	1830.02
200	2162.33	1109.64	725.6	2672.7	6531.68	1488.05	1286.72	1326.47
400	18373.37	9600.9	5471.36	4108.05	8382.34	5695.47	4862.11	5127.82
600	56223.63	28515.5	18164.19	14352.31	15628.49	12943.23	14441.65	12705.81
800	131868.4	67130.9	37918.27	30098.47	29234.41	28363.9	25452.17	26483.66
1000	265392.7	135927	74187.5	58541.4	53448.06	47783.99	46204.97	47851.44
1500	1221351	626320.4	344646	228751.8	213915.4	214844.6	213745.1	202705.4
2000	4416734	2273990	1250056	711129.1	675006.7	771371.9	727686.7	724087.8

Fig. 7: Resultados de MM1f en Ubuntu

En la tabla se pueden observar los valores en microsegundos, que determinan el tiempo de ejecución para la máquina *Ubuntu*. Si bien el comportamiento en esta máquina para los algoritmos es prácticamente el mismo, los tiempos disminuyen en todas las ejecuciones de los hilos. Se sigue reflejando en los resultados que al llegar a los ocho hilos, se alcanza el equilibrio en el tiempo y la eficiencia. Posiblemente para tamaños más grandes de las matrices se puede observar un cambio, o no.

B. *Ejecución máquina Windows*

A continuación se describen los resultados para la segunda máquina:

Como se puede observar en la figura, la ejecución tiene una mejora sustancial en comparación con la ejecución con filas por columnas. No obstante, se esperaba que se mantuviera la mejora en comparación con el código de Ubuntu, pero no ha sido así. El peor tiempo marca aproximadamente 7 segundos para Windows y 16 procesadores, y 4.5 segundos para Ubuntu

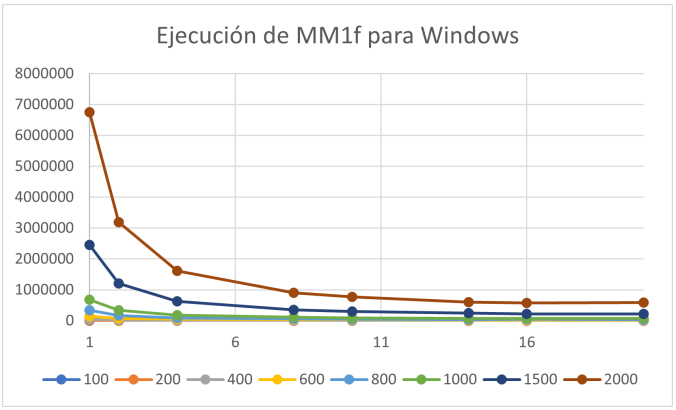

Fig. 8: Gráfica de MM1f en Windows

con 8 procesadores. Esto puede deberse a que el sistema operativo de Windows se ejecuta con otros programas en ejecución, tales como buscadores, hojas de cálculo, mientras que Ubuntu puede ejecutarse como consola solamente, reduciendo el consumo al máximo, para que sean los programas de multiplicación de matrices los encargados de gastar recursos.

	1	2	4	8	10	14	16	20
100	640.27	185.4	244.88	29.74	0	595.16	683.42	859.9
200	6302.53	2499.87	1427.28	1557.99	624.09	1410.27	748.56	1235.82
400	44450.67	21755.47	13971	11044.53	12855.18	9207.85	9501.15	6782.17
600	142985.8	73178	44528.79	33111.77	29402.3	22977.62	23529.13	20841.9
800	342875.9	172233.7	92478.8	62608.73	53750.19	39675.48	42422.41	36991.58
1000	674530.4	337340.9	177445.6	111310.5	92298.99	71660.84	69641.05	68409.37
1500	2459712	1205547	627534.8	350544.5	301939.1	241074.6	223186.5	219818.5
2000	6745670	3183292	1609047	907918.9	766938.8	597317	578236.4	583380

Fig. 9: Resultado de MM1f en Windows

La tabla muestra por primera vez en toda la ejecución del laboratorio, valores en cero, lo que significa que posiblemente la ejecución promedio fue en nanosegundos, pero como se estableció la medición en microsegundos, no se pudo capturar el valor exacto. Las ejecuciones de Windows muestran resultados mucho mejores que los resultados de Ubuntu para los tamaños pequeños de la matriz, pero para tamaños mayores, Ubuntu recupera ventaja. Lo anterior puede deberse a que Ubuntu maneja los recursos de una forma más eficiente, y porque se ejecuta sólo la consola de comandos, mientras que Windows tiene su interfaz gráfica siempre desplegada.

VI. Comparación del rendimiento

A continuación, se compara el rendimiento de los dos algoritmos de multiplicación de matrices. Durante el laboratorio se ha mostrado y analizado la métrica de tiempo de ejecución, donde se evidencia claramente que el algoritmo *MM1f* mejora la localidad espacial del código, y, por consiguiente, mejora el tiempo de ejecución. No obstante, ahora es necesario analizar el *speedup* de estos algoritmos, para comprobar exactamente cómo se comporta la ejecución paralela en comparación con la ejecución secuencial (un sólo hilo).

La siguiente tabla muestra el speedup promediado para las ejecuciones de los algoritmos de *MM1c*:

	1	2	4	8	10	14	16	20
100	1	1.53	1.72	2.27	2.64	2.58	2.34	2.18
200	1	1.78	2.74	3.57	3.83	5.86	4.84	7.05
400	1	2.07	2.84	2.51	3.05	3.29	3.85	3.74
600	1	1.97	3.48	3.33	3.92	4.83	5.00	5.61
800	1	2.01	3.71	4.73	5.40	6.20	6.61	7.15
1000	1	2.01	3.79	4.44	4.30	3.54	3.24	3.59
1500	1	1.90	3.60	5.55	4.74	3.33	3.06	2.93
2000	1	1.81	3.98	7.98	6.73	5.55	5.73	5.66

Fig. 10: Speedup recopilado por la ejecución de MM1c

Aquí se puede observar valores de speedup de hasta 8 veces más eficiencia. El speedup permite analizar la aceleración del algoritmo al aplicar paralelismo, comparando su eficiencia contra el modelo secuencial. Se puede observar que para los tamaños más grandes de las matrices, mejor aceleración y aprovechamiento del paralelismo se obtiene. No obstante, se puede observar que desde los ocho hilos el speedup no mejora, más bien, empeora lentamente. Por tanto, es importante concluir que con ocho hilos es suficiente para ejecutar este algoritmo, de esta forma se puede reducir costos de ejecución y tiempo.

	1	2	4	8	10	14	16	20
100	1	3.5	2.6	21.5	IND	1.1	0.9	0.7
200	1	2.5	4.4	4.0	10.1	4.5	8.4	5.1
400	1	2.0	3.2	4.0	3.5	4.8	4.7	6.6
600	1	2.0	3.2	4.3	4.9	6.2	6.1	6.9
800	1	2.0	3.7	5.5	6.4	8.6	8.1	9.3
1000	1	2.0	3.8	6.1	7.3	9.4	9.7	9.9
1500	1	2.0	3.9	7.0	8.1	10.2	11.0	11.2
2000	1	2.1	4.2	7.4	8.8	11.3	11.7	11.6

Fig. 11: Speedup recopilado por la ejecución de MM1f

En esta segunda tabla se pueden observar aceleraciones mucho más grandes que las presentes en la ejecución de *MM1c*. Una de las celdas contiene un valor indeterminado, puesto que la aceleración fue tan grande, que la ejecución del algoritmo fue en menos de un microsegundo, por lo que se puede deducir que para tamaños pequeños, la aceleración del algoritmo *MM1f* es exponencial.

Para ejecuciones más grandes, se tiene un speedup concurrente de alrededor de 10 veces más aceleración, lo cual prueba una vez más la importancia del manejo adecuado del espacio y el tiempo, en especial referencia a los apuntadores utilizados en la multiplicación de matrices. El mayor valor no indeterminado obtenido fue de 21, con lo cual se concluye que se ha logrado mejorar la ejecución satisfactoriamente. Un detalle importante es que a pesar de tener un valor indeterminado, el resto de hilos con cien de tamaño de matriz, muestran un speedup con una aceleración muy pequeña, por lo que se infiere que la ejecución mejora mucho más en tamaños de matrices más grandes.

VII. Conclusiones

Se puede concluir lo siguiente:

- Existe una mejora considerable al aplicar la localidad espacial al algoritmo de multiplicación de matrices. La localidad espacial incrementa el rendimiento puesto que se accede al mismo bloque de información proveída al procesador.
- El paralelismo mejora el tiempo de ejecución del algoritmo, pero siempre se llega a un punto de equilibrio.
- es posible realizar mejoras al código reduciendo la complejidad del mismo e incluso incorporando más máquinas para su ejecución.
- métricas de rendimiento como el Speedup y el tiempo de ejecución son determinísticos al momento de querer asignar recursos, ya sea locales o interconectando máquinas en un clúster. Es necesario mantener los procesadores ocupados el mayor tiempo posible, ya que cada tiempo de ocio es dinero que se desperdicia, o tiempo para poder determinar decisiones de gran valor.

References

[1] https://en.wikipedia.org/wiki/C_(programming_language)
[2] https://es.wikipedia.org/wiki/Stdlib.h
[3] https://www.tutorialspoint.com/c_standard_library/stdio_h.htm
[4] https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html
[5] https://github.com/pedro-escobar/laboratorio-openmp