29/5/2023

# Informe de trabajo práctico

## Régimen promoción


UNIVERSIDAD
NACIONAL
DE LA PLATA

## Sistemas distribuidos y paralelos - Ingeniería en computación


| Alumnos | N° de legajo |
|---|---|
| Dehan Lucas | 565/1 |
| Duarte Victor | 1055/7 |

# ÍNDICE

# Introducción

La cátedra nos brinda acceso al clúster de la facultad de informática, para ejecutar diversos programas secuenciales los cuales debemos paralelizar, con el fin de poder realizar métricas de rendimiento, y obtener cuál de las soluciones a los problemas presentados en base a sus tiempos de ejecución es mejor o más eficiente..

# Objetivos del proyecto primera parte

Como primer enunciado se nos pide resolver secuencialmente y utilizando la técnica por bloques la multiplicación de matrices cuadradas de N x N.

Siendo la matriz resultante C, que se obtiene multiplicando las matrices A.B. ambas de tamaño N x N. Y obtener el bloque óptimo de rendimiento.

## Ensayos y mediciones

Resolución: Para resolver dicho problema, generamos un código secuencial en el lenguaje de programación C. el cual recibe por parámetros el tamaño N de la matriz y el número de bloques Bs el cual representa porción de las matrices A y B, que calculan un bloque de C (Bs x BS), cuanto más grande Bs, más datos de A y B tendré en memoria, para realizar dichos cálculos. Cuanto más chico Bs mayor cantidad de fallos de caché tendré, dado que en memoria tendré menos datos.

La siguiente imagen muestra lo mencionado con un Bs = 2

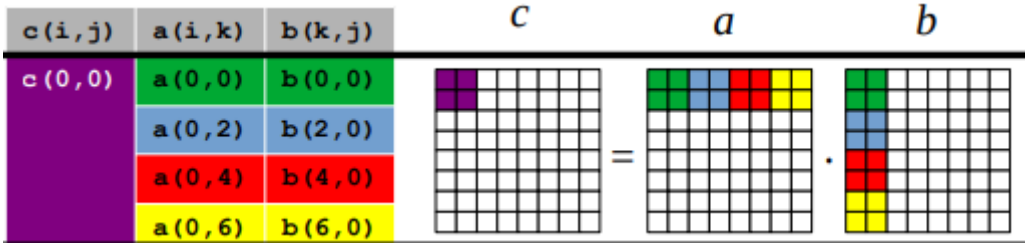


**Figura 1:** Multiplicación de matriz C=AB.

A continuación, se muestra el código implementado en C para resolver dicho problema.

```c
// Multiplicación de matrices por bloques
void matmulblks(double *a, double *b, double *c, int n, int bs){
double *ablk, *bblk, *cblk;
int I, J, K;
int i, j, k;

  for(I = 0; I < n; I += bs)
  {
    for(J = 0; J < n; J += bs)
    {
        cblk = &c[I*n + J];
        for(K = 0; K < n; K += bs)
        {
        ablk = &a[I*n + K];
        bblk = &b[J*n + K];

        for (i = 0; i < bs; i++)
            {
                for (j = 0; j < bs; j++)
                {
                    for  (k = 0; k < bs; k++)
                    {
                    cblk[i*n + j] += ablk[i*n + k] * bblk[j*n + k];
                    }
                }
            }
        }
    }
  }
}
```

**Figura 2:** Código del método para la multiplicación de matrices por bloques en C.

Para probar si el algoritmo es el correcto, se corre un programa que llama al módulo para realizar la multiplicación utilizando las matrices A y B como operandos y una matriz C donde se almacenarán los resultados.

```
Declarar las matrices A, B y C
Reservar memoria para las matrices
for (i=0 hasta N)
        for (j=0 hasta N){
                Inicializar A en la posición i,j en 1
                Inicializar B en la posición i,j en 1
                Inicializar C en la posición i,j en 0
        }
Multiplicar las matrices A y B
Verificar los resultados almacenados en C
```

Como se observa en el pseudocódigo anterior, se realiza una verificación de los resultados, pero para ello se tiene que saber de antemano cuál será el resultado correcto de la operación por lo que se asume que toda la matriz A está cargada con el mismo número, tal como la matriz B. De está forma es posible predecir el resultado de la multiplicación para la

verificación a partir de los conceptos de la multiplicación de dos matrices de tamaño NxN: Como se realiza la multiplicación de una fila de A que contiene N elementos con una columna de B que contiene N elementos, cada posición de la matriz de resultado contendrá el número cargado en A multiplicado por el número cargado en B multiplicado por N.

$$\begin{bmatrix} a & \cdots & a \\ \vdots & \ddots & \vdots \\ a & \cdots & a \end{bmatrix} \times \begin{bmatrix} b & \cdots & b \\ \vdots & \ddots & \vdots \\ b & \cdots & b \end{bmatrix} = \begin{bmatrix} N \times a \times b & \cdots & N \times a \times b \\ \vdots & \ddots & \vdots \\ N \times a \times b & \cdots & N \times a \times b \end{bmatrix}$$

Con esto se realiza la verificación mediante el siguiente pseudocódigo:

```
for (i=0 hasta N)
        for (j=0 hasta N)
                if (C[i,j] no es igual a N.a.b)
                        Informar del error
```

Ejecutando el programa mencionado con distintos valores de N y Bs en el clúster mencionado, se obtuvieron distintos tiempos en segundos, como muestra en la siguiente figura. Con la cual se pudo determinar el bloque óptimo de rendimiento pedido.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | | | SECUENCIAL | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | TAMAÑO MATRIZ (NxN) | | |
| 6 | | | | 1024 | 2048 | 4096 |
| 7 | | | | TIEMPO (EN SEGUNDOS) | | |
| 8 | TAMAÑO DEL BLOQUE (Bs) | | 32 | 13,185 | 105,025 | 838,612 |
| 9 | | | 64 | 13,037 | 104,175 | 834,14 |
| 10 | | | 128 | 12,873 | 102,93 | 824,47 |
| 11 | | | 256 | 12,795 | 102,256 | 856,396 |
| 12 | | | 512 | 12,773 | 105,607 | |
| 13 | | | 1024 | 12,956 | 104,758 | |

**Figura 3:** Tabla de tiempos obtenidos.

## Conclusiones

Como se observa en la imagen, se marcaron para cada matriz su mejor tiempo en base a un cierto tamaño de bloque. Dado que, para la matriz de 1024, el tiempo entre el bloque de 512 y 128 no varía mucho, y algo similar pasa para la matriz de 2048 entre el bloque de 256 y 128. Observando que al aumentar el tamaño de la matriz el menor tiempo tendía a el bloque óptimo de 128 como muestra la matriz de 4096, donde sí se nota una gran diferencia de tiempo entre el bloque de 128 y su valor anterior y posterior. Por lo tanto, determinamos que el valor óptimo de bloque es de Bs = 128.

# Objetivos del proyecto segunda parte

Como segundo objetivo utilizando el bloque óptimo obtenido anteriormente, se nos pide resolver la siguiente ecuación R = PromP(P) donde P = MaxD(ABC) + MinA(DCB)

siendo A, B, C, D y P matrices N x N. Para N = 1024, 2048 y 4096 respectivamente.

Dicha resolución debe implementarse con ejecución secuencial y paralelo (utilizando memoria compartida con OpenMP y Pthreads y utilizando memoria distribuida con MPI.).

## Ensayos y mediciones en ejecución secuencial

Para resolver dicha problemática del código secuencial, primero generamos un "primer" código secuencial, que resolvió el problema correctamente y se tomaron nota de los tiempos. Pudimos mejorar la versión de dicho código secuencial en un "segundo" código. Lo que nos dio mejores tiempos de ejecución y se calculó el SPEEDUP = T1/T2 donde T1 es el tiempo antes de la mejora, y T2: tiempo después de la mejora. dichos datos los podemos observar en la siguiente imagen.

| | | SECUENCIAL | | | | | |
|---|---|---|---|---|---|---|---|
| | | Primera codigo implementado | | | Segundo codigo implementado | | |
| | | TAMAÑO MATRIZ (NxN) | | | TAMAÑO MATRIZ (NxN) | | |
| | | 1024 | 2048 | 4096 | 1024 | 2048 | 4096 |
| | | TIEMPO (EN SEGUNDOS) | | | TIEMPO (EN SEGUNDOS) | | |
| TAMAÑO DEL BLOQUE (Bs) | 128 | 61,693 | 492,66 | 3940,974 | 49,560 | 410,740 | 3290,689 |
| | SPEEDUP | 1,24481437 | 1,1994449 | 1,19761363 | | | |

**Figura 4:** Tiempos de código secuencial y comparación de versiones.

Como se muestra en la figura, el SPEEDUP para un tamaño de matriz de 1024 es de 1,24 esto quiere decir que la versión mejorada, es 1,24 veces más rápida que la original.

lo mismo para matriz de 2048 que se obtuvo un SPEEDUP = 1,199 y matriz de 4096 que se obtuvo un SPEEDUP = 1,197.

A continuación se muestra en la siguiente imagen el código secuencial tanto de la versión primera y segunda.

```
timetick = dwalltime();
printf("Realizando operacion: ABC... \n");
matmulblks(A, B, E, n, bs);
matmulblks(E, C, F, n, bs);
printf("Realizando operacion: DCB... \n");
matmulblks(D, C, G, n, bs);
matmulblks(G, B, H, n, bs);
printf("Realizando operacion: Hallar minimo de D... \n");
maxD = matminblks(D, n, bs);
printf("Realizando operacion: Hallar maximo de A... \n");
minA = matmaxblks(A, n, bs);
printf("Realizando operacion: MaxD(ABC) \n");
matescblks(F, I, maxD, n, bs);
printf("Realizando operacion: MinA(DCB) \n");
matescblks(H, J, minA, n, bs);
printf("Realizando operacion: P = MaxD(ABC) + MinA(DCB)... \n");
matsumablks(I, J, P, n, bs);
printf("Realizando operacion: Hallar promedio de P... \n");
promP = matpromblks(P, n, bs);
printf("Realizando operacion: R = PromP(P)... \n");
matescblks(P, R, promP, n, bs);
printf("Ejecucion finalizada.\n");
```

**Figura 5:** Código de computo de primera versión secuencial

```
timetick = dwalltime();

multibloques(A, B, C, D, E, F, G, H, &minA, &maxD, n, bs);
sumabloques(F, H, P, minA, maxD, &promP, n);
calcularR(P, R, promP, n);


double totalTime = dwalltime() - timetick;
```

**Figura 6:** Código de computo de la segunda versión secuencial

Nos dimos cuenta que en la primera versión del código se recorren las matrices más veces que lo necesario y además se realizaban otras operaciones como suma de matrices y producto de una matriz por un escalar por bloques cuando la unica operacion que se requería que fuera por bloques era la multiplicación, por lo que en la segunda versión se enfocó en reducir la cantidad de recorridos de la matriz y de modificar las operaciones en matrices que no fueran multiplicaciones entre matrices para que fuesen punto a punto. En está segunda versión, en la llamada a "multibloques" en un mismo recorrido se realizan las multiplicaciones AB, DC, la obtención de mínimo de la matriz A y el máximo de la matriz D. En "sumabloques" se obtiene P con su promedio y en el último procedimiento "calcularR" se obtiene R.

El algoritmo para verificar los resultados es una extensión del algoritmo propuesto en la primera parte con el objetivo de verificar los valores almacenados tanto en P como en R. Para predecir cuál es el valor que deben tener las matrices se asume que cada matriz está cargada

con un mismo valor en cada posición por lo que el resultado de P se obtiene a partir del siguiente desarrollo:

A partir de los resultados de la multiplicación entre las matrices A y B:

$$\begin{bmatrix} a & \cdots & a \\ \vdots & \ddots & \vdots \\ a & \cdots & a \end{bmatrix} \times \begin{bmatrix} b & \cdots & b \\ \vdots & \ddots & \vdots \\ b & \cdots & b \end{bmatrix} = \begin{bmatrix} N \times a \times b & \cdots & N \times a \times b \\ \vdots & \ddots & \vdots \\ N \times a \times b & \cdots & N \times a \times b \end{bmatrix}$$

Al multiplicar AB por C el resultado es el siguiente:

$$\begin{bmatrix} N \times a \times b & \cdots & N \times a \times b \\ \vdots & \ddots & \vdots \\ N \times a \times b & \cdots & N \times a \times b \end{bmatrix} \times \begin{bmatrix} c & \cdots & c \\ \vdots & \ddots & \vdots \\ c & \cdots & c \end{bmatrix} = \begin{bmatrix} N^2 \times a \times b \times c & \cdots & N^2 \times a \times b \times c \\ \vdots & \ddots & \vdots \\ N^2 \times a \times b \times c & \cdots & N^2 \times a \times b \times c \end{bmatrix}$$

De manera análoga, DCB es una matriz NxN cuyo valor en cada posición es $N^2 \times d \times c \times b$.

Como la matriz A y D están cargados con un mismo valor en cada posición de la matriz, el valor mínimo de A es $a$, y el valor máximo de D es $d$. Por lo tanto $MaxD(ABC) = a \times b \times c \times d = MinA(DCB)$ entonces se puede afirmar que P será una matriz NxN cargada con el valor $p = 2 \times a \times b \times c \times d$. Dado que el promedio una matriz cargada con el mismo número será ese número, el promedio de P será $p$ y se puede calcular que R será una matriz NxN cargada con el valor $p^2$.

## Ensayos y mediciones en OpenMP y Pthreads

Luego de obtener lo que consideramos el mejor código secuencial que resuelve dicha problemática, se implementó tanto en Pthreads como en OpenMP una versión paralela del código que realiza todo el trabajo de cómputo en un único llamado al procedimiento "calculoTotal" que paraleliza el cómputo en 4 y 8 hilos (Ver figuras 7 y 8).

Se calcularon promedios de los tiempos de 3 ejecuciones para los 3 tamaños de matices pedidas, se calcularon los SPEEDUP = Ts/Tp, siendo Ts = Tiempo secuencial. y Tp = tiempo paralelo promedio, obteniendo como parámetro cuanto más rápido es el paralelo del secuencial en dichas versiones, también se calculó la EFICIENCIA = S/P siendo S = SPEEDUP y P = N° de unidades de procesamiento.

| | | PTHREADS | | | | | |
|---|---|---|---|---|---|---|---|
| | | 4 THREADS | | | 8 THREADS | | |
| | | TAMAÑO MATRIZ (NxN) | | | TAMAÑO MATRIZ (NxN) | | |
| | | 1024 | 2048 | 4096 | 1024 | 2048 | 4096 |
| | | TIEMPO (EN SEGUNDOS) | | | TIEMPO (EN SEGUNDOS) | | |
| TAMAÑO DEL BLOQUE (Bs) 128 | | 12,302 | 102,237 | 817,208 | 6,162 | 51,22 | 419,256 |
| | | 12,304 | 102,264 | 817,37 | 6,162 | 51,231 | 421,701 |
| | | 12,303 | 103,773 | 816,874 | 6,164 | 51,215 | 415,483 |
| | Promedio | 12,303 | 102,758 | 817,150667 | 6,16266667 | 51,222 | 418,813333 |
| | | | | | | | |
| | SPEEDUP | 4,02617248 | 3,99305164 | 4,02671274 | 8,03775422 | 8,01058139 | 7,85655742 |
| | | | | | | | |
| | EFICIENCIA | 1,00654312 | 0,99826291 | 1,00667819 | 1,00471928 | 1,00132267 | 0,98206968 |

**Figura 7:** Tiempos de código paralelo en Pthreads.

| | | OPENMP | | | | | |
|---|---|---|---|---|---|---|---|
| | | 4 THREADS | | | 8 THREADS | | |
| | | TAMAÑO MATRIZ (NxN) | | | TAMAÑO MATRIZ (NxN) | | |
| | | 1024 | 2048 | 4096 | 1024 | 2048 | 4096 |
| | | TIEMPO (EN SEGUNDOS) | | | TIEMPO (EN SEGUNDOS) | | |
| TAMAÑO DEL BLOQUE (Bs) 128 | | 12,257 | 101,908 | 814,45 | 6,141 | 51,049 | 418,916 |
| | | 12,266 | 101,886 | 814,493 | 6,145 | 51,7 | 419,886 |
| | | 12,258 | 101,893 | 814,335 | 6,151 | 51,052 | 414,447 |
| | Promedio | 12,2603333 | 101,895667 | 814,426 | 6,14566667 | 51,267 | 417,749667 |
| | | | | | | | |
| | SPEEDUP | 4,04230445 | 4,03098594 | 4,04050092 | 8,06421869 | 8,01178146 | 7,87717924 |
| | | | | | | | |
| | EFICIENCIA | 1,01057611 | 1,00774649 | 1,01012523 | 1,00802734 | 1,00147268 | 0,9846474 |

**Figura 8:** Tiempos de código paralelo en OpenMP.

Para implementar la estrategia de paralelización para Pthread y OpenMP, primero se codificó un método para que realice todo el cómputo de forma secuencial para luego implementar las directivas que usan tanto Pthread como OpenMP para paralelizar la ejecución del método a partir de dividir la matriz NxN en partes acorde a la cantidad de procesos con la que se ejecute el programa. A continuación se describe el pseudocódigo de calculoTotal:

```
for (I= inicio hasta fin )
        for(J=0 hasta N)
                for (K=0 hasta N)
                        for (i hasta bs)
                                for (j hasta bs)
                                        for (k hasta bs){
                                                Realizar multiplicación AB
                                                Realizar multiplicación DC
                                                Hallar mínimo local de A
                                                Hallar máximo local de D
                                        }
Definir el mínimo absoluto de A para todos los procesos
Definir el máximo absoluto de D para todos los procesos
for (I= inicio hasta fin )
        for(J=0 hasta N)
                for (K=0 hasta N)
                        for (i hasta bs)
                                for (j hasta bs)
                                        for (k hasta bs){
                                                Realizar multiplicación ABC
                                                Realizar multiplicación DCB
                                        }
for (I= inicio hasta fin )
        for(J=0 hasta N){
                Calcular P
                Calcular sumatoria local de P
        }
```

La estrategia que se implementó en OpenMP fue utilizar las directivas para indicar que la ejecución de calculoTotal sea en paralelo por el número de hilos: en cada for que recorre la matriz se aplican directivas para dividir las iteraciones que cada proceso le corresponderá para dividir la tarea de cómputo, y en las partes en las que se tiene que acceder a una variable global como el máximo y mínimo se hace uso de una directiva para que cada proceso realiza la tarea de forma privada con las partes de la matrices que le fueron asignadas y al terminar juntar sus resultados para reducir el acceso a la variable global.

## Conclusiones

Como se puede observar en las imágenes anteriores, los tiempo mejoraron notoriamente, si bien entre ambos códigos paralelos no se observa diferencia notorias, comparando cada con el secuencial, podemos ver que para 4 Threads con una matriz de 1024 el SPEEDUP > P, esto pasa en la mayoría de casos, tanto para tamaños de matriz 2048 y 4096 tanto en 4 como 8 Threads. Esto se lo conoce como SPEEDUP superlineal, quiere decir que el algoritmo

secuencial que consideramos mejor versión nuestra, no es el más óptimo para resolver dicho trabajo o la versiones paralelas del algoritmo realizan menos trabajo que la versión secuencial, por consecuencia de la arquitectura utilizada.

En cuanto a la EFICIENCIA (E) cómo se calcula en base a el SPEEDUP, esta debería dar valores $0 < E < 1$, pero como nuestro speedup es superlineal, algunos resultados dieron mayor a 1.

## Ensayos y mediciones en MPI

MPI trabaja sobre memoria distribuida por lo que cada "core" ejecuta una copia del programa y tiene una memoria independiente del resto, y como tal no se puede trabajar con variables globales. Entonces, para que todos los procesos trabajen en sincronía es necesario que exista una comunicación entre los procesos, esto se logra utilizando las siguientes sentencias que ofrece MPI:

MPI_Scatter: Un proceso raíz trocea un mensaje en partes iguales y los envía individualmente al resto de procesos y a sí mismo.

MPI_Gather: Recoge una serie de datos de varios procesos en un único proceso raíz (operación en la cual interviene también el propio proceso raíz).

MPI_Bcast: Envía un mensaje desde un proceso origen a todos los procesos pertenecientes al mismo grupo.

MPI_Allreduce: Reduce un valor de un grupo de procesos y lo redistribuye entre todos.

Para las primeras 3 sentencias, los mensajes serán las matrices y para el Allreduce el valor serán los máximo valor de D y mínimo valor de A.

Dado que necesitaremos un proceso raíz para los mensajes, nuestra estrategia será separar la ejecución del cómputo según el ID: El proceso con el ID 0 será designado como el proceso raíz y ejecutará el computo se realizará mediante el método "Proceso0" mientras que los otros procesos ejecutarán el método "Proceso1" para realizar el cómputo.

La principal diferencia entre los métodos está en el contenido de los mensajes que se envían y la memoria que alocan los procesos. El proceso 0 alocara memoria para las matrices de tamaño NxN y dos buffer de tamaño reducido acorde a la cantidad de procesos corriendo, mientras que los otros procesos alocan memoria para dos buffers y matrices de un tamaño reducido acorde a la cantidad de procesos ya que estos solo tienen que trabajar con las

matrices de entrada que les designa el proceso 0, solo este último tiene que tener la memoria alocada suficiente para alocar los resultados.

A continuación se muestran el pseudocódigo que realiza el cómputo en el proceso 0:

```
Alocar memoria para almacenar las matrices
Inicializar las matrices
Distribuir los elementos A y D al resto de los procesos y a sí mismo para
trabajar
Envía las matrices B y C completas
Realizar la multiplicación AB y DC y, obtener un minA y maxD locales
Comunicar hacia los otros procesos el minimo y máximo local hallado
para consensuar cuál es el minimo y maximo absoluto
Realizar la multiplicación ABC y DCB
Calcular P
Juntar todos los resultados de los otros procesos y almacenarlos
Calcular la sumatoria de la parte asignada de P
Juntar los resultados de la sumatoria de los otros procesos
Calcular promedio de P
Calcular R
Juntar todos los resultados de los otros procesos y almacenarlos
Validar los resultados
Liberar memoria
```

A continuación, el pseudocódigo que realiza el cómputo en los otros procesos:

```
Alocar memoria para almacenar los elementos de la matriz que enviara el
proceso raíz
Recibir los elementos A y D  para trabajar
Recibir las matrices B y C completas
Realizar la multiplicación AB y DC y, obtener un minA y maxD locales
Comunicar hacia los otros procesos el mínimo y máximo local hallado
para consensuar cuál es el minimo y maximo absoluto
Realizar la multiplicación ABC y DCB
Calcular P
Enviar el resultado del cálculo de P al proceso raíz
Calcular la sumatoria de la parte asignada de P
Enviar el resultado de la sumatoria de P al proceso raíz y esperar a recibir
el resultado final
Calcular promedio de P
Calcular R
Enviar el resultado del cálculo de R al proceso raíz
Liberar memoria
```

A partir de estos algoritmos, se corre el programa para ejecutarse en 4, 8 y 16 procesos en el cluster, tomando el recaudo durante la ejecución con 16 procesos de reducir el tamaño del

bloque ya que uno de 128 es demasiado grande para una matriz de 1024 x 1024 por lo que se reduce el tamaño de bloque en este caso a un tamaño de 32 x 32. A continuación se presentan los resultados:

| | | MPI | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 4 CORES | | | 8 CORES | | |
| | | | TAMAÑO | | | TAMAÑO MATRIZ (NxN) | | |
| | | | 1024 | 2048 | 4096 | 1024 | 2048 | 4096 |
| | | | TIEMPO (EN SEGUNDOS) | | | TIEMPO (EN SEGUNDOS) | | |
| TAMAÑO DEL BLOQUE (Bs) | | 128 | 12,773 | 102,939 | 868,558 | 6,562 | 52,196 | 429,725 |
| | | | 12,773 | 103,026 | 840,916 | 6,555 | 52,171 | 424,937 |
| | | | 12,767 | 103,035 | 863,046 | 6,563 | 52,195 | 419,382 |
| | Promedio | | 12,771 | 103,000 | 857,507 | 6,560 | 52,187 | 424,681 |
| | SPEEDUP | | 3,88066714 | 3,98776699 | 3,83750836 | 7,55487805 | 7,8704922 | 7,74860758 |
| | EFICIENCIA | | 0,97016678 | 0,99694175 | 0,95937709 | 0,94435976 | 0,98381153 | 0,96857595 |

**Figura 9:** Tiempos de código paralelo en MPI para 4 y 8 cores.

| | | | 16 CORES | | |
|---|---|---|---|---|---|
| | | | TAMAÑO MATRIZ (NxN) | | |
| | | | 1024 | 2048 | 4096 |
| | | | TIEMPO (EN SEGUNDOS) | | |
| TAMAÑO DEL BLOQUE (Bs) | | 32 | 3,566 | 27,085 | 210,653 |
| | | | 3,565 | 26,974 | 211,196 |
| | | | 3,569 | 27,09 | 210,579 |
| | | | 3,567 | 27,050 | 210,809 |
| | | | 13,8953271 | 15,1846603 | 15,609788 |
| | | | 0,86845794 | 0,94904127 | 0,97561175 |

**Figura 10:** Tiempos de código paralelo en MPI para 16 cores.

## Conclusiones

Como se puede observar en las imágenes anteriores, los tiempos de 4 y 8 cores son similares a los de memoria compartida pero no dan un speedup superlineal sino uno cercano al número de cores en las que fue ejecutada el programa.

En cuanto a la EFICIENCIA (E) cómo se calcula en base a el SPEEDUP, esta debería dar valores $0 < E < 1$, lo cual entra dentro del rango y se obtiene una eficiencia entre 0.86 y 0.99 por lo que podemos decir que este algoritmo es bastante eficiente.

# Escalabilidad

| ESCALABILIDAD PTHREAD | | | |
|---|---|---|---|
| | TAMAÑO MATRIZ (NxN) | | |
| | 1024 | 2048 | 4096 |
| 4 | 1,00707145 | 0,99928959 | 1,00675712 |
| 8 | 1,00524665 | 1,0023525 | 0,98214668 |

| ESCALABILIDAD OPENMP | | | |
|---|---|---|---|
| | TAMAÑO MATRIZ (NxN) | | |
| | 1024 | 2048 | 4096 |
| 4 | 1,01057611 | 1,00774649 | 1,01012523 |
| 8 | 1,00802734 | 1,00147268 | 0,9846474 |

| ESCALABILIDAD MPI | | | |
|---|---|---|---|
| | TAMAÑO MATRIZ (NxN) | | |
| | 1024 | 2048 | 4096 |
| 4 | 0,97016678 | 0,99694175 | 0,95937709 |
| 8 | 0,94435976 | 0,98381153 | 0,96857595 |
| 16 | 0,86845794 | 0,94904127 | 0,97561175 |

**Figura 11:** Tablas de escalabilidad

## Conclusiones

Viendo las tres gráficas de escalabilidad al observar la diagonal que se forma si aumentamos linealmente el tamaño de la matriz en conjunto con hilos o cores. podemos ver que el valor de eficiencia se mantiene relativamente constante por lo que el programa paralelo es débilmente escalable. Esto quiere decir que nuestro programa mantendrá una eficiencia constante al incrementar el tamaño del problema y la cantidad de unidades de procesamiento a la vez.

Por otro lado, si observamos para un tamaño específico del problema, el comportamiento de la eficiencia al ir aumentando cantidad de hilos o cores podemos ver qué para pthreads y openmp, que para cualquier tamaño de matriz fijo la eficiencia se mantiene relativamente constante al aumentar hilos o cores, por lo que podríamos suponer que posiblemente sea fuertemente escalable, en cambio si observamos la tabla de MPI sólo a partir de la 3ra columna, donde la matriz es de 4096 la eficiencia se mantiene constante al aumentar los cores, es posible que sea fuertemente escalable a partir de este tamaño de matriz.

# Extra

Para Pthread se adjuntan dos códigos ya que se realizó una mejora al implementar reduce para el cálculo del máximo, mínimo y de la suma pero no se pudo medir los tiempos de ejecución de está versión ya que en nuestras computadoras ocurre un error en ejecución (Floating point exception (core dumped)) del cual no pudimos encontrar la fuente del problema dado que

durante el debug pudimos determinar que el error ocurre en el método reduce_valores pero no hallamos la causa.