DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico final

Sistemas Complejos en Máquinas Paralelas

Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Oller, Luca Esteban	667/13	ollerrr@live.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

En este informe explicaremos el desarrollo del trabajo práctico final de la materia.

Éste consiste en la simulación del crecimiento de un tumor cerebral.

Más precisamente, de los gliomas del cerebro humano, tumores que rara vez suelen ser curados.

Los gliomas son un grupo de tumores que comparten la capacidad de penetrar difusivamente el cerebro, aunque raramente sucede el fenómeno de la metástasis fuera del sistema nervioso central.

Se utilizará la matriz de Talairach y otra matriz llamada brainweb, que representa el cerebro y que contiene números que indican una escala de gris, varían de 0 a 255.

Además, se calcularán las concentraciones de las áreas de Brodmann a partir de una matriz que superpone las matrices Talairach y brainweb mencionadas anteriormente.

Estas áreas se empiezan a calcular en el momento que comienza a crecer el tumor maligno y su fin es el de predecir las funciones cerebrales que son afectadas durante el crecimiento del tumor.

Los modelos que incluyen la evolución de un 'paciente virtual' suelen predecir más fidedignamente ayudando a los médicos para elegir el tratamiento más adecuado.

Estos modelos están hechos de manera personalizada para cada paciente mediante el uso de valores obtenidos de cada uno de los mismos. Consecuentemente, los modelos matemáticos de la evolución de los gliomas humanos han sido desarrollados basados en la proliferación y la invasión de células tumorales y esto produce una mejor predicción del tumor real.

Para este trabajo se dicretizó una ecuación diferencial parcial con el método explícito utilizando diferencias finitas.

El modelo consiste en dos estados: la proliferación, que es la etapa del tumor benigno y la etapa de migración, donde crece el tumor maligno.

Durante la primera etapa, el tumor solo crece dentro de un punto inicial ubicado en la posición (32,98,85), manteniéndose allí y, una vez que la concentración de la matriz superpuesta es mayor a 10^7 células por mm^3, se pasa a la segunda etapa que consiste en el crecimiento del tumor maligno donde además de haber proliferación hay migración. En esta última etapa, el tumor comienza a expandirse a otros nodos según la ecuación de reacción-difusión que será explicada luego.

Además se observó el crecimiento de la concentración de estas células tumorales en las áreas de Brodmann del cerebro.

Se implementó una versión de C++ a partir de una versión de Matlab y luego, a partir de la versión de C++, dos versiones paralelas: una de MPI y otra de OpenMP.

Se detallarán las características de cada implementación, focalizando en las ventajas y desventajas.

Finalmente, se mostrarán los resultados conseguidos en el crecimiento del tumor y las comparaciones entre las implementaciones realizadas.

1.1. Modelo matemático

El modelo utilizado consiste en una ecuación diferencial parcial de reacción-difusión con la siguiente ecuación:

$$\frac{\partial C}{\partial t} = \rho c(1 - \frac{c}{Cm}) + D\nabla^2 c$$

Que describe el crecimiento del tumor del glioma cerebral.

Donde c es la concentración de células tumorales en un nodo ($células/mm^3$), t es el tiempo (días), ρ es el ratio de proliferación neta de las células tumorales (células/días), Cm la capacidad de carga del tumor ($células/mm^3$) y D es la tasa de difusión (mm^2/d) de células tumorales.

El código utiliza las siguientes parámetros.

Parámetro	Valor	Parámetro	Valor
$p1$	0,107 células/d	$p2$	0,0107 células/d
$Dw1$	0,255 mm^2/d	$Dw2$	0,805 mm^2/d
$Dg1$	0,051 mm^2/d	$Dg2$	0,161 mm^2/d
$d_{diag}1$	18,26mm	$d_{diag}2$	16,98 mm
d_{let}	70mm	c_{mass}	10^6 $células/mm^3$
c_{inv}	10^7 $células/mm^3$	c_{max}	10^8 $células/mm^3$

1.2. Implementaciones

El desarrollo se realizó enteramente utilizando C++.

Las dos versiones paralelizadas se desarrollaron utilizando OpenMP y MPI. Para graficar la matriz 3D de concentraciones se utilizó Paraview sobre archivos VTK generados por el código a intervalos regulares de iteraciones temporales.

El video se realizó con Paraview utilizando los VTKs anteriormente mencionados.

Las mediciones de tiempo se realizaron utilizando la librería estándar Chrono de C++.

1.3. Estructuras de datos utilizadas

El problema propuesto por el trabajo práctico requiere la representación de una matriz tridimensional para alojar los resultados de las concentraciones de células tumorales calculadas así como los coeficientes de difusión y sus derivadas.

Previendo que el programa iba a tener que paralelizarse utilizando MPI, decidimos utilizar una estructura lo más sencilla posible: Representar la matriz como un vector de TAM_X x TAM_Y x TAM_Z, donde estas tres últimas son las dimensiones del paralelepípedo que representa el dominio del problema. Las posiciones x, y, z de la matriz se guardan en el orden que denota la figura 1.

```
    [(0,0,0 - TAMX-1,0,0),(0,1,0 - TAMX-1,1,0),...((0,(TAMY-1),0 -
(TAMX-1),(TAMY-1),0))],

    [(0,0,1 - TAMX-1,0,1),(0,1,1 - TAMX-1,1,1),...((0,(TAMY-1),1 -
(TAMX-1),(TAMY-1),1))],

    .

    .

    .

    [(0,0,(TAMZ-1) - TAMX-1,0,(TAMZ-1)),(0,1,(TAMZ-1) -
TAMX-1,1,(TAMZ-1)),...((0,(TAMY-1),1 - (TAMX-1),(TAMY-1),(TAMZ-1)))]
```

Figura 1: Dimensiones.

De esta manera, el elemento (i,j,k) se corresponde a la posición k * (TAM_X * TAM_Y) + j * TAM_X + i

En la primera implementación habíamos utilizado una clase que funcionaba como un wrapper del vector de datos. De esta manera, las posiciones se accedían mediante setters y getters dados x,y,z. Sin embargo, se descubrió que esto suponía cierto overhead debido al context-switching del llamado a funciones de la clase. Si bien esto puede parecer un tiempo insignificante, debemos tener en cuenta que se ejecuta decenas de millones de veces por iteración. Debido a lo expuesto, se toma la decisión de utilizar el array directamente y accederlo mediante macros como se puede ver en la figura 2.

```
#define G3D(V,X,Y,Z)  V[(Z) * ((TAM_X) * (TAM_Y)) + (Y) * (TAM_X)
+ (X)]      // Hace el GET de una posición

#define S3D(V,X,Y,Z,S)  V[(Z) * ((TAM_X) * (TAM_Y)) + (Y) * TAM_X
+ (X)]=S   // Hacel el SET de una posición
```

Figura 2: Definiciones utilizadas en la implementación.

Se comparó la performance de ambas implementaciones, obteniendo con esta última una mejora significativa de tiempos.

Cuando se implementaron los ciclos anidados sobre las matrices, tuvimos en cuenta que siempre el ciclo externo itere sobre Z, luego el del medio sobre Y el el interno sobre X. De esta manera, los arrays se recorren secuencialmente aprovechando el principio de localidad espacial. La única matriz que varía en el tiempo es la que contiene las concentraciones que se desean calcular (C(x,y,z)). Tanto la matriz de coeficientes de difusión D como la matriz M se mantienen constantes una vez que se inicializan sus valores, es decir no son modificadas durante las iteraciones temporales ni espaciales. Por este motivo, para la versiòn de C++ y la de openmp tanto C, como M y P son variables globales, mientras que en MPI sólo C se divide en slices a ser utilizada para cada proceso, dándonos además de C, C_slice la matriz que

es modificada por cada proceso independientemente, mientras que luego se mezclan todos los resultados en la matriz C respetando los índices.

1.4. Diferencias entre Matlab y C++

Para realizar la implementación de C++ fue necesario adaptar el código de matlab.

Se dividió el código en funciones para el mejor entendimiento del mismo, se realizó una documentación de cada función.

Se modificaron los switch y case de la versión de matlab y se conviertieron en ifs en C++ utilizando vectores y viendo la pertenencia del elemento en dicho vector, porque C++ no permite colocar más de un entero en el mismo case.

Además se observó que en la iteración de convergencia había 3 for anidados que recorrían toda la matriz, pero para la primer etapa (la etapa de migración) solo se iba a modificar el valor del nodo inicia, teniendo un impacto negativo en la performance del programa.

Por lo tanto, la solución implementada fue la de chequear si no se estaba en migración, es decir si estábamos en la etapa de proliferación y si el if daba verdadero, solo se veía ese nodo y no toda la matriz, en caso de dar falso se verificaban todos los nodos de la matriz, algo que iba a suceder de todas formas por estar en la etapa de migración del tumor.

1.5. Paralelización utilizando OpenMP

Para paralelizar con OpenMP utilicé las directivas #pragma correspondientes sobre el loop principal de la siguiente manera:

```
#pragma omp parallel for collapse(3)  schedule(static) num_threads(threads)
       for (k=1;k<kk-1;k++){
           for (j=1;j<jj-1;j++){
               for (i=1;i<ii-1;i++){
```

Donde threads se usa para definir la cantidad de threads que se utiliza al ejecutar esta versión.

1.6. Paralelización utilizando MPI

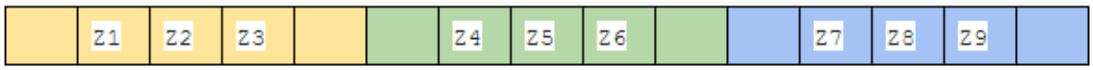
Para la versión de MPI se decidió acortar el alcance del problema para simplificar la paralelización, el código consiste en una etapa de tumor benigno, donde el tumor nunca pasa a la etapa maligna. De esta forma, se evita tener que chequear si el nodo inicial, llamado (io,jo,ko) en la implementación alcanza el punto en el que se debe comenzar la migración obteniendo así un código más simple para su paralelización.

El esquema de paralelización utilizado se basa en dividir el dominio 3D del problema en rodajas iguales(salvo el último proceso) y asignar cada una de estas a un proceso. Se optó por la división en rodajas por su simplicidad y porque solo requiere el pasaje entre procesos de las fronteras izquierda y derecha. Y considero conveniente llevar a cabo esta división a lo largo del eje Z por dos motivos:

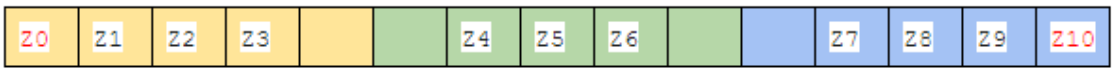
- En nuestra implementación sobre un vector de doubles de la matriz tridimensional, todas las celdas pertenecientes a una misma coordenada Z se encuentran guardados en forma contigua y ordenada por Z. Esto simplifica enormemente la división del espacio de memoria, simplemente debe asignarse a cada proceso una porción del arreglo de C * (Tamaño_X * Tamaño_Y), donde C es la cantidad de rodajas asignadas a cada proceso.
- Al trabajar con rodajas del plano X-Y para un Z fijo, aprovechamos el principio de localidad espacial, dado que cada plano XY se encuentra guardado en forma contigua en el arreglo. Esto supone una mejora en el uso de la cache de los procesadores locales.

Veamos un ejemplo de cómo se lleva a cabo el proceso paralelizado. Definimos Zi como la rodaja que representa el plano X-Y para Z = i. De esta forma, nuestra matriz 3D se compone un arreglo de K rodajas. Supongamos K = 11, P = 3 procesos:

Z0 y Z10 son valores borde y no deben ser procesados. Esto nos deja con 9 rodajas para procesar, 3 por proceso. Sin embargo, los procesos necesitan los valores de frontera de los procesos vecinos por lo que se cada proceso aloca espacio para 5 rodajas. Luego, el proceso P0 lleva a cabo un MPI_Scatter, transmitiendo los siguientes valores a cada proceso:



Para este problema puntual, las condiciones de borde Z0 y Z10 son 0 por lo que estos datos se pueden enviar a los procesos de las puntas P0 y P2 (de hecho no es necesario dado que 0 es el default de los arrays)

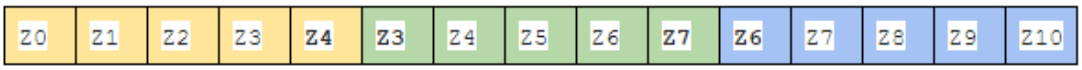


Por último, en cada iteración se ejecuta una función ”halo”que se encarga de que los procesos contiguos se intercambien los resultados de los bordes. Este procedimiento se lleva a cabo al inicio de cada iteración de la siguiente manera:

```
Shift left
P0[4] <- P1[1]
P1[4] <- P2[1]

Shift right
P1[0] <- P0[3]
P2[0] <- P1[3]
```

Resultando en el siguiente arreglo:



Con estos datos en cada proceso ya es posible realizar los cálculos locales. Cada proceso ejecuta su propia iteración. Al final de cada iteración existe una sincronización entre procesos (barrier). De esta forma se puede asegurar que el intercambio de bordes se haga con datos coherentes (no desfasados en el tiempo)

Finalmente, luego de la última iteración, se ejecuta MPI_Gather para volver a componer la matriz 3D de concentraciones.

2. Resultados obtenidos

2.1. Resultados

La simulación fue realizada sobre un cubo de 181x217x181 mm. Donde el tumor tiene origen en la posición (32,98,85).

Estas imágenes fueron realizadas con Paraview utilizando los vtk de salida que provee el código.

Se realizó una superposición del tumor en dos etapas. Para el comienzo del diagnóstico y para la muerte del paciente, cada uno con el cerebro y otro con la matriz Talairach.

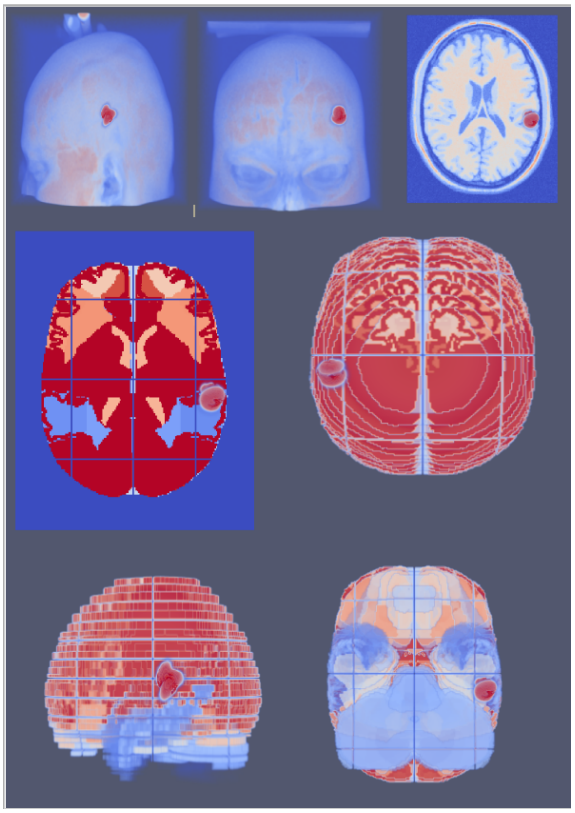


Figura 3: Diágnostico en cerebro y talairach, distintas vistas.

En la imagen se puede ver el tumor en el instante que es diagnosticado. Está superpuesto en una vista transversal, lateral y frontal para poder apreciar el fenómeno desde distintos ángulos.

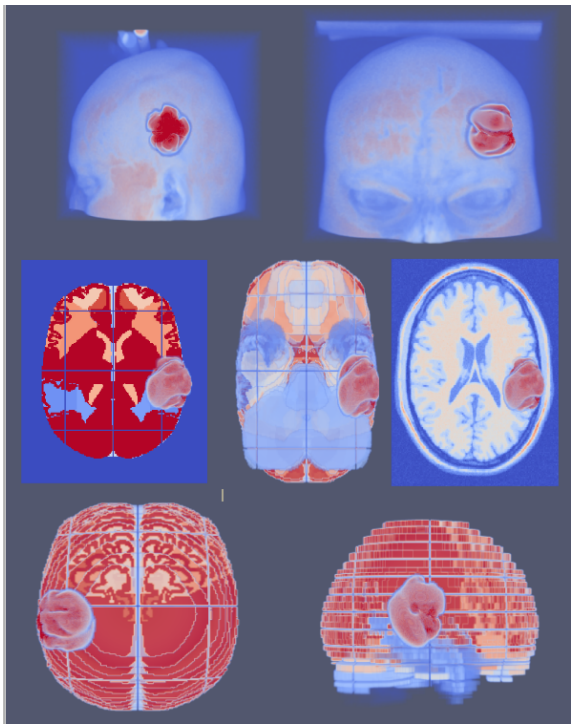


Figura 4: Muerte del paciente en cerebro y talairach, distintas vistas.

Mientras que en esta imagen se puede ver el tumor en el instante que el paciente muere, con las mismas vistas y superposiciones que la figura 3.

2.2. Mediciones de tiempo y comparaciones

Las implementaciones, serial, OpenMP difieren solamente en la paralelización de las iteraciones del loop temporal. Por tal motivo, lo más relevante es tomar las mediciones de tiempo como lo que tarda en ejecutar dicho loop.

De especial interés es determinar cómo escalan las implementaciones paralelas a medida que se van agregando procesos o threads.

Hay que destacar que todas las implementaciones fueron compiladas utilizando la opción -O3 del compilador.

Les mediciones de tiempo fueron realizados con las computadoras de los laboratorios del Departamento de Computación.

Se partió de la implementación de matlab que tardaba aproximadamente 12 horas para una ejecución de 3000 iteraciones, es decir 300 días de simulación.

La implementación serial tardó 1072 segundos en completar en las computadoras del DC, utilizando optimización O3, aproximadamente 18 minutos, es decir obtenemos una mejora de rendimiento sustancial ya con sólo pasar de Matlab a C++.

2.2.1. Openmp

Para estudiar los tiempos en OpenMp se realizó un análisis de Gustavson en el cluster del departamento Cecar.

Se pudo analizar la escalabilidad hasta 4 threads, dado que para 8 ya no se podía mantener la eficiencia constante porque el dominio de la matriz crecía demasiado y había que pedir memoria adicional al cluster, es decir más de 4 GB de memoria para alojar las matrices. Se cree que el fenómeno de la disminución del speedup ocurre porque se hacen más misses a la caché al aumentar el tamaño de las matrices del problema.

La siguiente tabla muestra los tiempos medidos, cuanto se aumentó a cada coordenada de la matriz y el speedup máximo obtenido.

#Procesos	Tiempo 3000 It (seg.)	tiempo +50	tiempo +100	tiempo +200	tiempo +300	Speed-Up
1	483	570	589	1751	4088	-
2	380	-	-	-	-	0.64
4	214	222	222	-	-	0.64
8	150	155	162	1011	2216	-
16	225	230	232	-	1955	-
32	200	207	208	951	1894	-

Tiempo empleado para 3000 iteraciones, versión OpenMP

Se puede ver que para 8 procesos, se tiene que 483/150 = 3.22 y 3.22/8= 0.4025 para la matriz de tamaño original, mientras que para el tiempo obtenido para +200 ya se puede observar que 1751/1011=1.73 que es menor a 3.22, por lo que el speedup decrece al dividir a ambos números por 8.

Para 32 procesos se puede ver el mismo efecto: 483/200 =2.415 y 2.415/32 =0.075 y para le tiempo +300 4088/1894=2.16 que es claramente menor a 2.415 por lo que se puede apreciar el mismo fenómeno. Mientras que para 16 procesos ocurre lo mismo, es decir, se tiene que 483/225= 2.147 y para tiempo +300 es 4088/1955=2.09.

En el siguiente gráfico se pueden ver los tiempos obtenidos en el cluster de la versión de OpenMP.

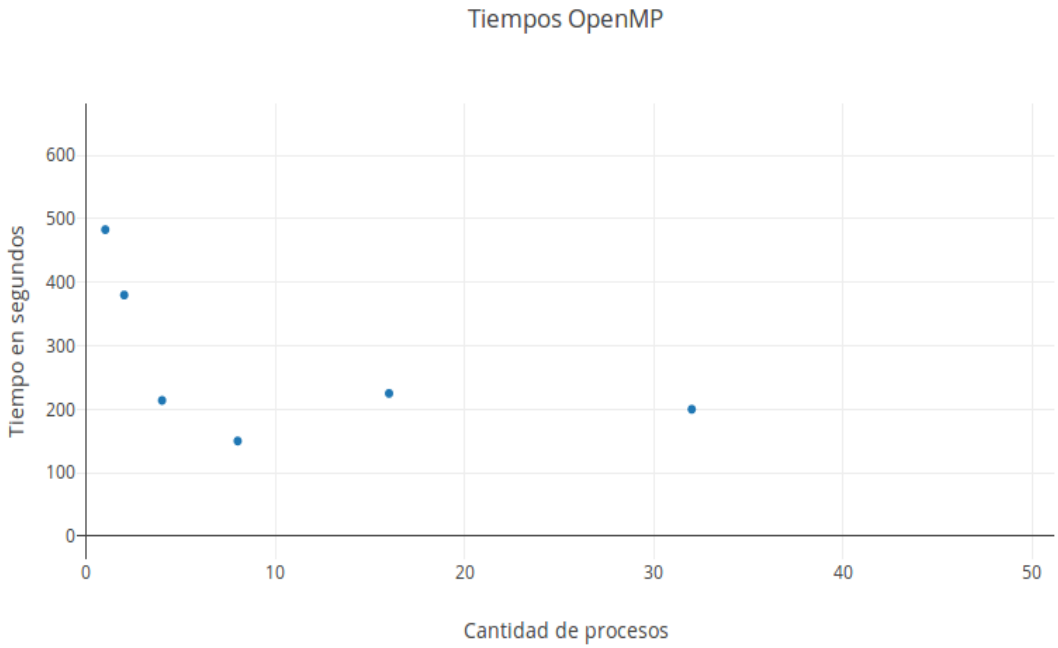


Figura 5: Speed up MPI.

2.2.2. MPI

Hay que destacar que al ser distintas las implementaciones de MPI y openMP no son comparables entre sí.

Mientras que para poder comparar la versión serial con la de MPI, se realizó una versión serial similar a la de MPI, es decir sin crecimiento de tumor maligno y solo con difusión.

Esta versión serial tardó 1577.96 segundos en ejecutarse y la mejora obtenida utilizando MPI fue de un 80 %

Para MPI se optó por otro enfoque, realizar el análisis Amdahl.

#Procesos	Tiempo 3000 It (seg.)	Speed-Up
1	684.983	-
2	431.873	1.59
3	329.701	2.08
4	311.568	2.20
5	313.961	2.18
10	329.379	2.08
20	330.368	2.07
30	376.667	1.82
40	412.211	1.93

Tiempo empleado para 3000 iteraciones, versión MPI

Puede observarse que, a partir de los 4 procesos, seguir agregando procesos solo empeora la performance, es decir, que para 4 procesos se obtuvo el menor tiempo de ejecución.

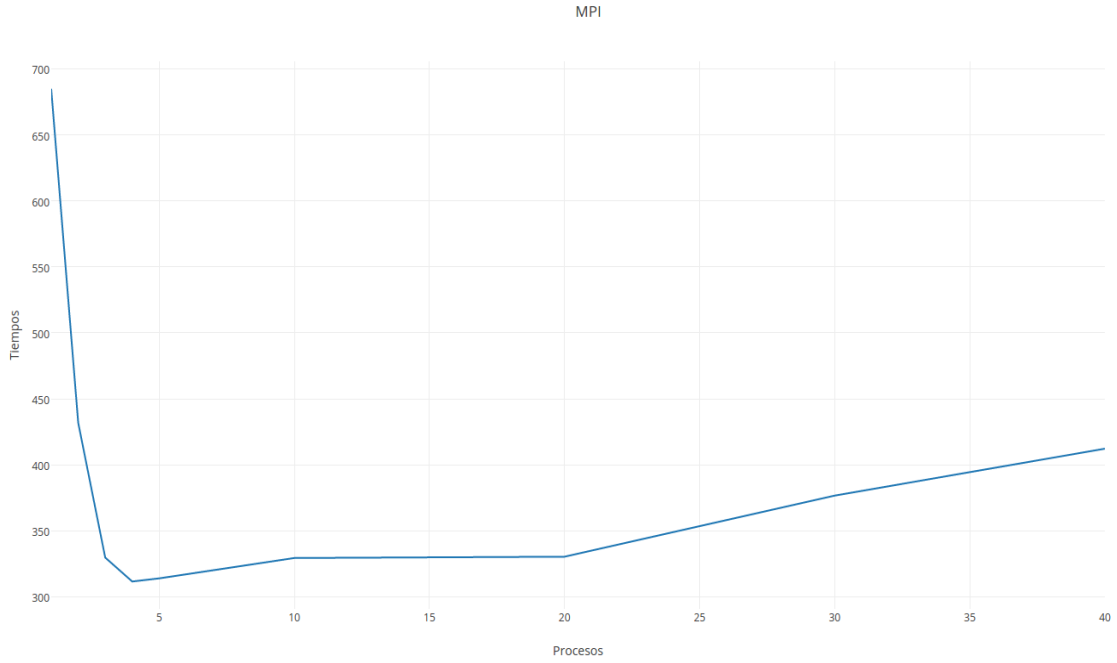


Figura 6: Tiempo de procesamiento (MPI)

La figura 8 grafica el Speed-Up logrado respecto a un proceso. Se utilizó una escala logarítmica para una mejor apreciación de la curva.

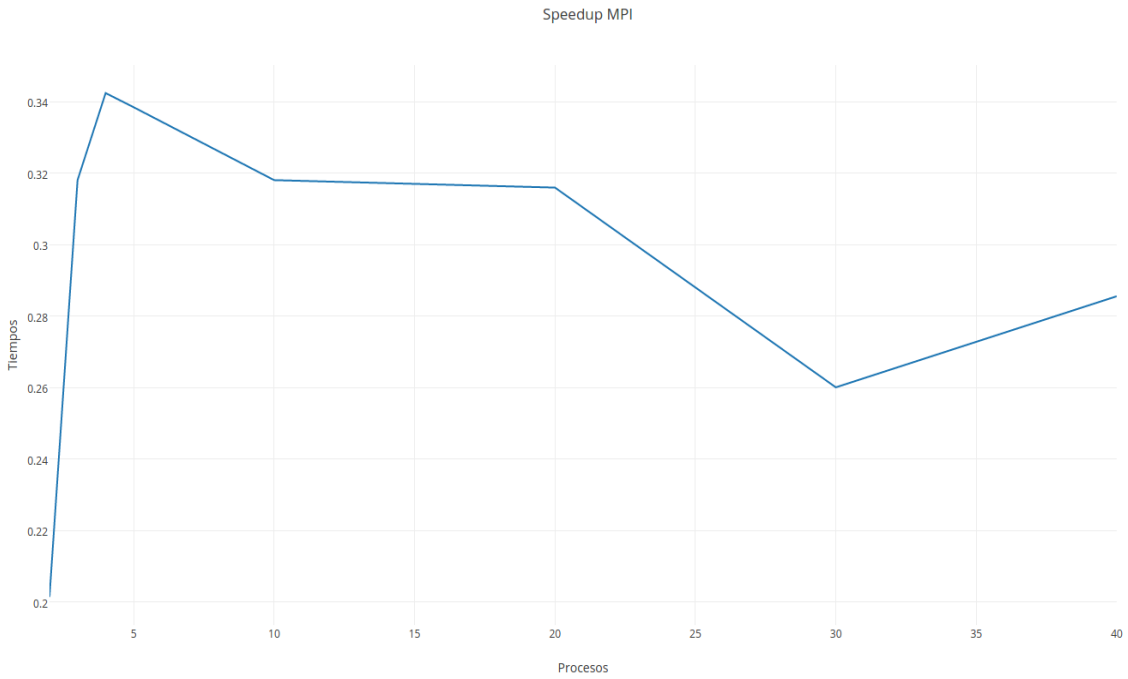


Figura 7: Speed up MPI.

3. Conclusiones

- De la versión de C++ a la versión de matlab hay un salto muy grande de tiempos, estimo que esto se debe a cuestiones de como se optimiza el código con 03 y que C++ es un lenguaje de más bajo nivel que matlab. Además considero que la implementación de la matriz de una sola dimensión ayuda mucho a mejorar el rendimiento, queda pendiente hacer la comparación entre esta versión con otra implementación de una matriz de tres dimensiones como se utiliza usualmente para verificar si esta suposición es verdadera.
- Si bien de la versión de C++ a la versión de OpenMP la diferencia de tiempos no es tan importante, sigue siendo considerable dado que la mejor performance se obtuvo en los 10 procesos y para los 50 el tiempo de ejecución seguía siendo menor al tiempo de ejecución de C++.
- La paralelización con MPI aumenta drásticamente la performance de la simulación. Sin embargo, para la instancia de prueba no parece escalar mas allá de los 10 procesos simultáneos. Se cree que esto es debido mayormente a que, al aumentar la cantidad de procesos, disminuye la cantidad de procesamiento propio (rodajas interiores de cada proceso) respecto al overhead (mensajes de intercambio de bordes). Por lo tanto, debe buscarse un compromiso entre la cantidad de procesos y la cantidad de rodajas asignadas a los mismos.
- Es importante prestar atención a las posibles optimizaciones de los procesos seriales. En el caso actual, un buen uso de la propiedad de localidad espacial de la cache significó un aumento de performance comparable con el de la paralelización.

Referencias

[1] Colonna M Breitburd K Marshall G Suarez C, Maglietti F. Mathematical Modeling of Human Glioma Growth Based on Brain Topological Structures: Study of Two Clinical Cases. . *PLoS ONE 7(6): e39616*, 7(10):1–10, 2012.

[2] Marshall G. Solución numérica de ecuaciones diferenciales. pages 12–21, 1986.

[3] Pacheco. Parallel programming with MPI. *Morgan Kaufmann*, 1997.

[4] A. Skjellum W. Gropp, E. Lusk. Using MPI: Portable Parallel Programming with Message Passing Interface. *MIT Press*, 1999.

[5] D. Kohr D Maydan J McDonald R Menosh R Chandra, L. Dagum. Parallel programming in OpenMP. 1:12–21, 1986.

[6] G. F. Pinder L. Lapidus. Numerical Solution of Partial Differential Equations in Science and Engineering. 1982.