Análisis del rendimiento del programa

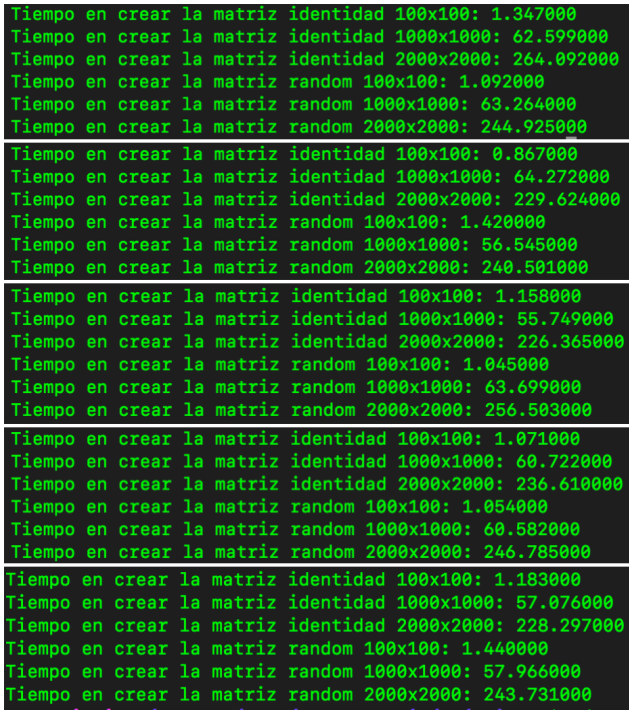
El programa consta de cuatro partes principales:

1. Crear una matriz de tamaño YxY con números decimales aleatorios y guardarla en un archivo en binario de la siguiente manera: primero se guardan como enteros las filas y las columnas y después como decimales los datos de la matriz en fila.
2. Leer la matriz que hemos creado antes y almacenarla en una estructura Matriz con las filas y columnas y los datos en un array de decimales .
3. Leer una matriz diagonal de fichero que previamente hayamos creado y almacenarla en una estructura Matriz como hemos hecho antes.
4. Multiplicar las dos matrices que hemos leído de fichero mediante un algoritmo.

Vamos a hacer una batería de pruebas para crear y multiplicar las matrices de diferentes tamaños:

Creación:

Matriz I 100x100:	1.125
Matriz R 100x100:	1.21
Matriz I 1000x1000:	60.083
Matriz R 1000x1000:	60.411
Matriz I 2000x2000:	236.997
Matriz R 2000x2000:	246.489



Multiplicación:

100x100

Leer matriz 1:	0.868
Leer matriz 2:	0.74
Multiplicar matrices:	0.983
Total:	3.801

```
Tiempo en leer la matriz 1: 1.009000
Tiempo en leer la matriz 2: 0.743000
Tiempo en multiplicar las matrices: 0.934000
Son iguales
```

```
Tiempo en leer la matriz 1: 0.881000
Tiempo en leer la matriz 2: 0.832000
Tiempo en multiplicar las matrices: 0.986000
Son iguales
```

```
Tiempo en leer la matriz 1: 0.817000
Tiempo en leer la matriz 2: 0.675000
Tiempo en multiplicar las matrices: 1.020000
Son iguales
```

```
Tiempo en leer la matriz 1: 0.742000
Tiempo en leer la matriz 2: 0.666000
Tiempo en multiplicar las matrices: 0.997000
Son iguales
```

```
Tiempo en leer la matriz 1: 0.891000
Tiempo en leer la matriz 2: 0.784000
Tiempo en multiplicar las matrices: 0.982000
Son iguales
```

1000x1000

Leer matriz 1:	56.342
Leer matriz 2:	52.505
Multiplicar matrices:	1002.614
Total:	1171.872

```
Tiempo en leer la matriz 1: 53.408000
Tiempo en leer la matriz 2: 52.681000
Tiempo en multiplicar las matrices: 1000.407000
Son iguales
```

```
Tiempo en leer la matriz 1: 53.889000
Tiempo en leer la matriz 2: 53.112000
Tiempo en multiplicar las matrices: 1039.309000
Son iguales
```

```
Tiempo en leer la matriz 1: 57.717000
Tiempo en leer la matriz 2: 52.096000
Tiempo en multiplicar las matrices: 995.398000
Son iguales
```

```
Tiempo en leer la matriz 1: 55.354000
Tiempo en leer la matriz 2: 52.181000
Tiempo en multiplicar las matrices: 968.885000
Son iguales
```

```
Tiempo en leer la matriz 1: 61.344000
Tiempo en leer la matriz 2: 52.455000
Tiempo en multiplicar las matrices: 1009.071000
Son iguales
```

2000x2000

Leer matriz 1:	215.121
Leer matriz 2:	209.387
Multiplicar matrices:	57567.087
Total:	58238.084

```
Tiempo en leer la matriz 1: 216.308000
Tiempo en leer la matriz 2: 205.975000
Tiempo en multiplicar las matrices: 57163.806000
Son iguales
```

```
Tiempo en leer la matriz 1: 214.042000
Tiempo en leer la matriz 2: 215.602000
Tiempo en multiplicar las matrices: 57527.409000
Son iguales
```

```
Tiempo en leer la matriz 1: 215.032000
Tiempo en leer la matriz 2: 208.785000
Tiempo en multiplicar las matrices: 57515.735000
Son iguales
```

```
Tiempo en leer la matriz 1: 213.905000
Tiempo en leer la matriz 2: 208.383000
Tiempo en multiplicar las matrices: 57230.717000
Son iguales
```

```
Tiempo en leer la matriz 1: 216.322000
Tiempo en leer la matriz 2: 208.192000
Tiempo en multiplicar las matrices: 58397.771000
Son iguales
```

Vamos a calcular el porcentaje de ejecución de cada parte del programa para cada tamaño de la matriz cuadrada: 100, 1000, y 2000.
Para ello vamos a usar el tiempo de creación de la matriz aleatoria, no el de la matriz identidad, ya que suponemos que está creada.

100x100

Crear matriz random:	31.84%
Leer matriz random:	22.84%
Leer matriz identidad:	19.46%
Multiplicar matrices:	25.86%

1000x1000

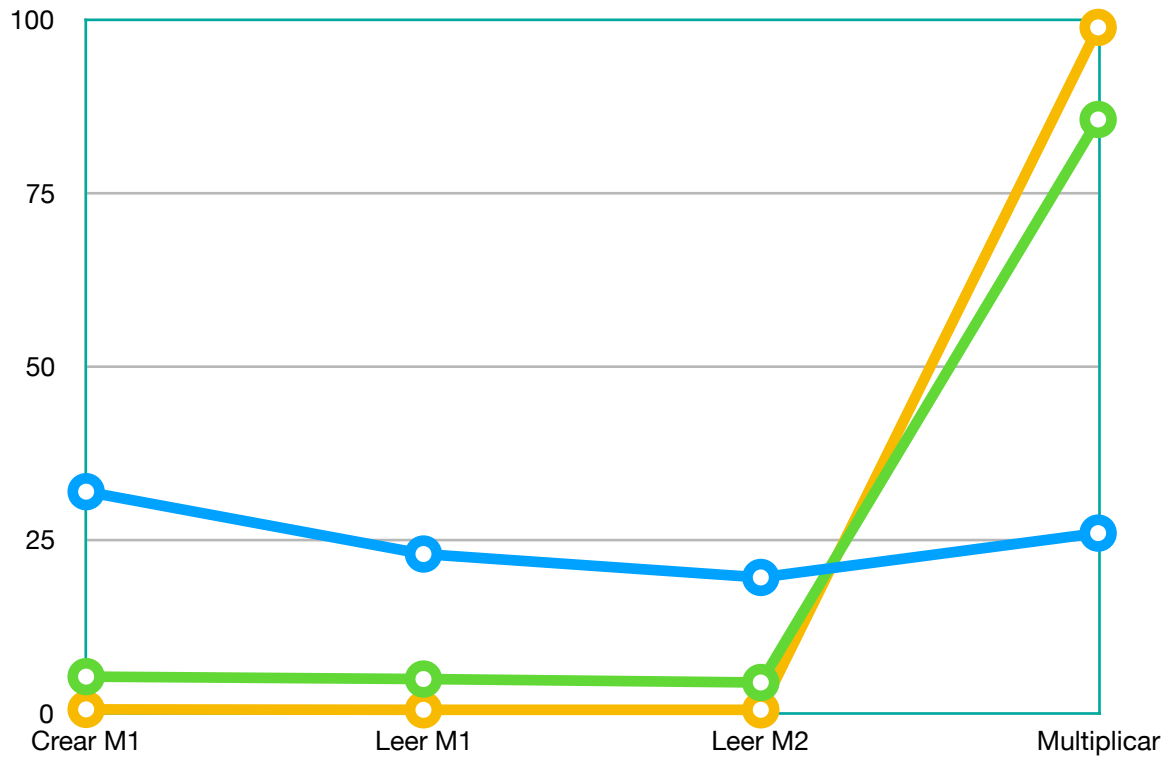
Crear matriz random:	5.15%
Leer matriz random:	4.80%
Leer matriz identidad:	4.30%
Multiplicar matrices:	85.55%

2000x2000

Crear matriz random:	0.42%
Leer matriz random:	0.36%
Leer matriz identidad:	0.36%
Multiplicar matrices:	98.84%

Observamos como según aumentan los parámetros de entrada, aumenta el tiempo de ejecución de las 3 primeras partes de forma lineal (O(n)), y el tiempo de ejecución de la multiplicación aumenta de manera exponencial (O(n^2)).
Esto provoca que según aumenta el tamaño de la matriz, las primeras partes llegan a ser despreciables: ≈1%.

Podemos ver gráficamente como cambian los tiempos de ejecución en porcentaje según el tamaño de la matriz:



Según aumenta el tamaño de la matriz el tiempo relativo de multiplicar tiende al 100%.

La parte del programa que podemos paralelizar es la multiplicación, así que el tiempo relativo de esa fase del programa será nuestra fracción de mejora.

$$\text{Speedup}_{\text{global}} = \frac{1}{(1 - \text{Fracción}_{\text{mejora}}) + \frac{\text{Fracción}_{\text{mejora}}}{\text{Speedup}_{\text{mejora}}}}$$

100x100

S = 1 / ((1- 0.2586) + (0.2586 / 2)) = 1.1485
S = 1 / ((1- 0.2586) + (0.2586 / 4)) = 1.2406
S = 1 / ((1- 0.2586) + (0.2586 / 6)) = 1.2746
S = 1 / ((1- 0.2586) + (0.2586 / 8)) = 1.2924
S = 1 / ((1- 0.2586) + (0.2586 / 10)) = 1.3033
S = 1 / ((1- 0.2586) + (0.2586 / 100)) = 1.3441
S = 1 / ((1- 0.2586) + (0.2586 / MAX)) = 1.3487

1000x1000

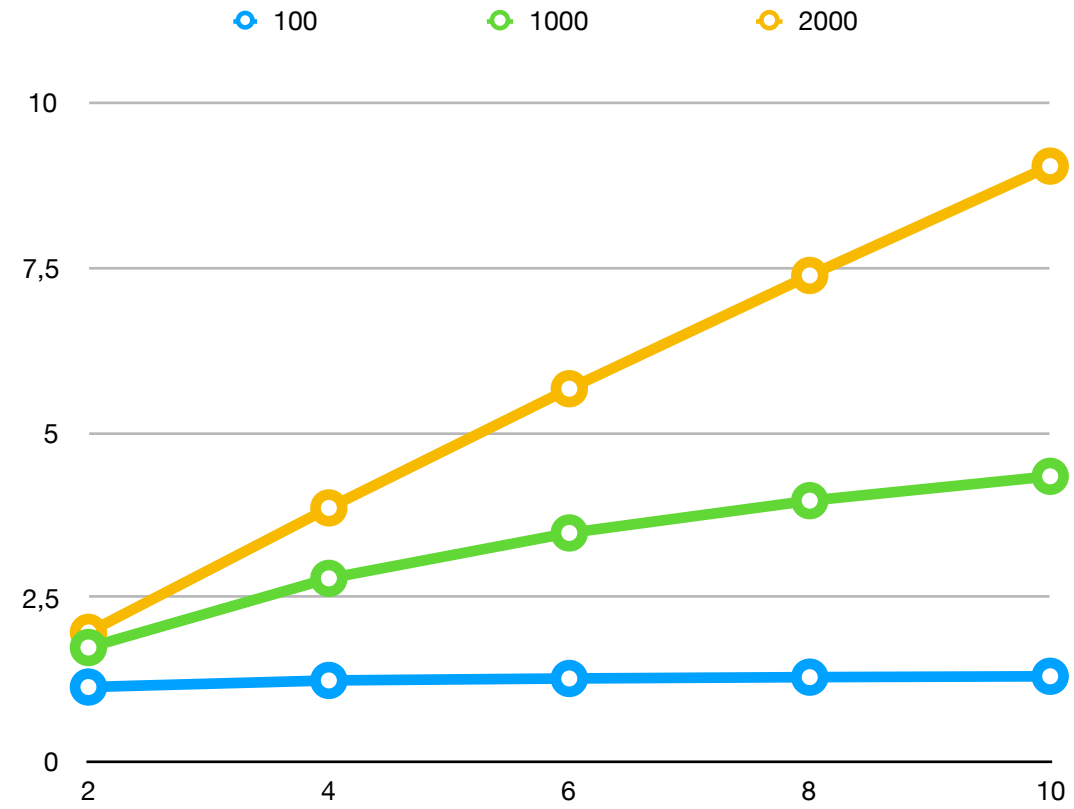
S = 1 / ((1- 0.8555) + (0.8555 / 2)) = 1.7474
S = 1 / ((1- 0.8555) + (0.8555 / 4)) = 2.7903
S = 1 / ((1- 0.8555) + (0.8555 / 6)) = 3.4833
S = 1 / ((1- 0.8555) + (0.8555 / 8)) = 3.9771
S = 1 / ((1- 0.8555) + (0.8555 / 10)) = 4.3468
S = 1 / ((1- 0.8555) + (0.8555 / 100)) = 6.5335
S = 1 / ((1- 0.8555) + (0.8555 / MAX)) = 6.9204

2000x2000

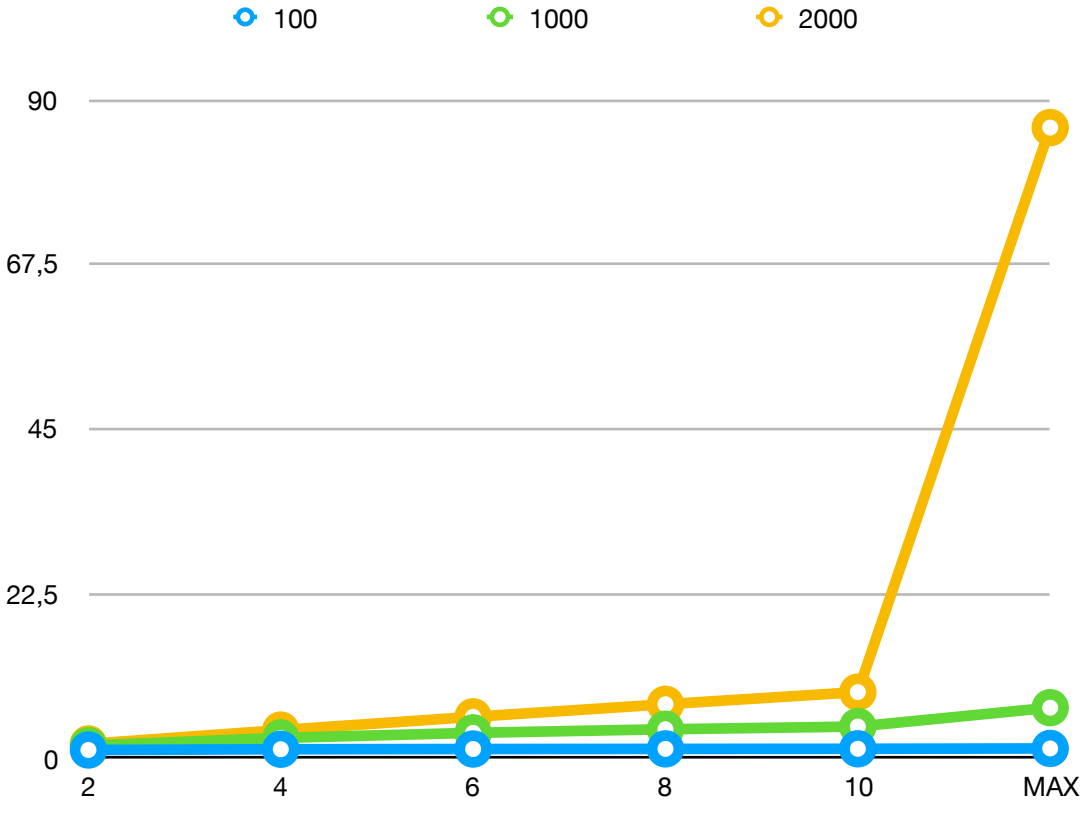
S = 1 / ((1- 0.9884) + (0.9884 / 2)) = 1.9770
S = 1 / ((1- 0.9884) + (0.9884 / 4)) = 3.8654
S = 1 / ((1- 0.9884) + (0.9884 / 6)) = 5.6710
S = 1 / ((1- 0.9884) + (0.9884 / 8)) = 7.3991
S = 1 / ((1- 0.9884) + (0.9884 / 10)) = 9.0546
S = 1 / ((1- 0.9884) + (0.9884 / 100)) = 46.5462
S = 1 / ((1- 0.9884) + (0.9884 / MAX)) = 86.2068

Las matrices pequeñas necesitan muy pocos procesadores para acercarse al speed up teórico máximo, mientras que las matrices mas grandes necesitan muchos.

Podemos ver visualmente la escalabilidad del speed up según el tamaño de la matriz y el número de procesadores:



Si incluimos el speed up teórico máximo en la gráfica:



Parte 2

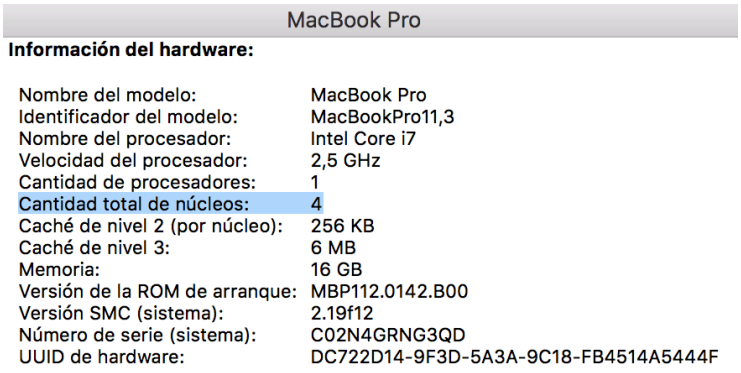
Ahora vamos a hacer en análisis utilizando std::thread para multiplicar las matrices: concurrencia vs computación secuencial.

Vamos a utilizar tres métodos para multiplicar las matrices.
El primer método consistirá en crear un threads para cada uno de los vectores que haya que multiplicar, superando con creces el numero de procesadores que tenemos en el ordenador.
El problema de este método es que gastamos una cantidad enorme de tiempo de ejecución tanto en crear como en gestionar esos threads.

El segundo método trata de crear un thread para cada uno de los procesadores que tengamos en el ordenador, dividiendo las matrices en tantos trozos como procesadores tengamos.
La ventaja que obtenemos con este método es que casi no gastamos tiempo en la creación y gestión de esos procesos.
Sin embargo la desventaja que podemos encontrar es que, al ser procesos tan largos, alguno puede terminar antes que los demás, provocando que tengamos alguno de los procesadores inactivo durante la ejecución.

El tercer método trata de solucionar el problema que podíamos encontrar en el segundo método.
Para ello vamos a generar una lista de X trabajos, siendo los trabajos divisiones más pequeñas de la matriz, y desde cada uno de los threads iremos accediendo a esa lista y procesándola hasta que quede vacía.
Esto nos permite poder dividir el trabajo en muchas mas partes que threads tengamos sin tenemos el problema de la carga computacional en creación y gestión de threads.

El ordenador que voy a estar usando tiene 4 procesadores, así que no debería mostrar una gran mejora cuando usemos más de 4 threads.



Método “a lo loco”

Para este método no puedo separar en 1, 2, 4 y 8 procesadores porque el método consiste en crear threads y lanzarlos sin tener en cuenta qué procesadores están libres ni cuantos tenemos.

Mientras programaba esta parte me he dado cuenta de que el tiempo de la creación de los threads era extremadamente grande.
Al principio quise medir exactamente cuanto tardaba en crear todos estos threads, pero cuando me puse a crear grandes cantidades de threads (1000*1000) sin ir ejecutándolos, cada vez se creaban mas lentos hasta que el proceso se estancaba cerca del 12% y dejaba de avanzar, así que las pruebas están realizadas creando y ejecutando cada thread a la vez, en vez de separar estos dos procesos.

100x100

```
Tiempo en leer la matriz 1: 0.730000
Tiempo en leer la matriz 2: 0.790000
Tiempo total en multiplicar las matrices: 272.219000
Son iguales
```

1000x1000

```
Tiempo en leer la matriz 1: 60.462000
Tiempo en leer la matriz 2: 72.416000
Tiempo total en multiplicar las matrices: 24532.463000
Son iguales
```

2000x2000

```
Tiempo en leer la matriz 1: 238.597000
Tiempo en leer la matriz 2: 275.799000
Tiempo total en multiplicar las matrices: 104226.055000
Son iguales
```

10000x10000

```
Tiempo en leer la matriz 1: 5738.560000
Tiempo en leer la matriz 2: 7124.380000
100.0%
Tiempo total en multiplicar las matrices: 3464952.316000
Son iguales
```

57 minutos

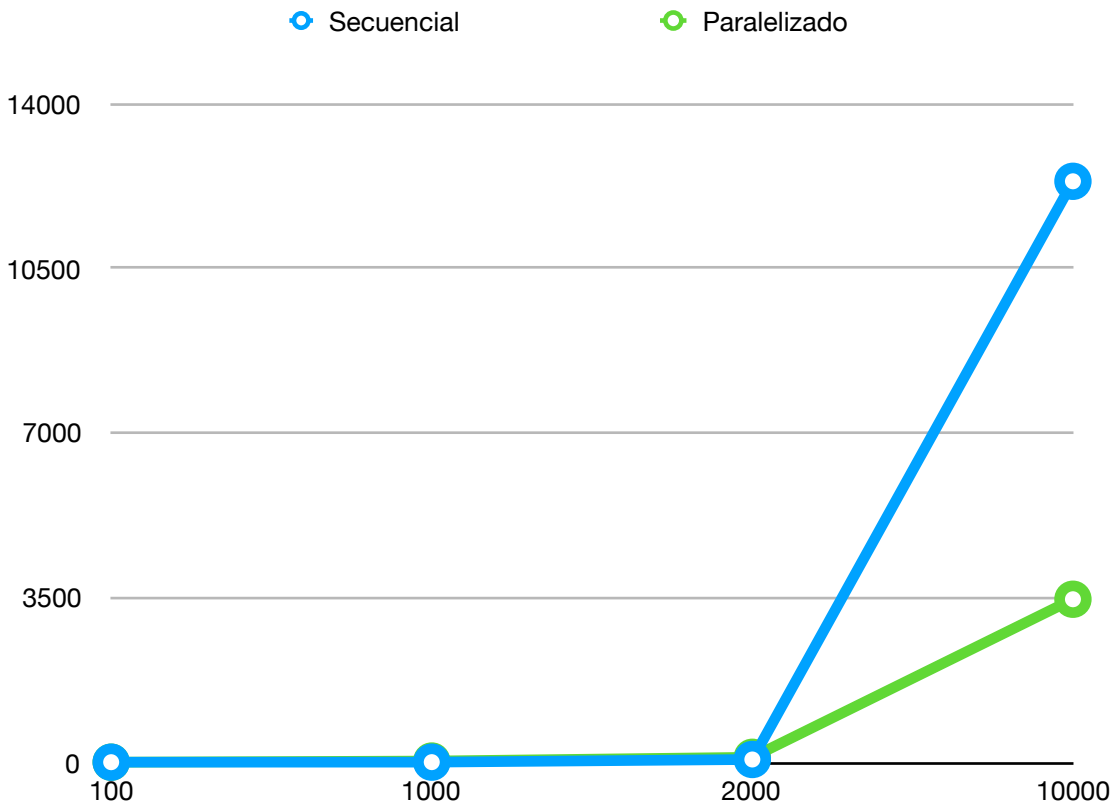
Y ahora vamos a ver la matriz de 10000x10000 sin utilizar threads:

```
Tiempo en leer la matriz 1: 5562.131000
Tiempo en leer la matriz 2: 5538.957000
100.0%
Tiempo total en multiplicar las matrices: 12349116.253000
Son iguales
```

205 minutos
3.4 horas

Se ve un 100% extra porque tardaba tanto que tuve que poner el porcentaje para saber por donde iba.

Mostramos en una gráfica la mejora que hemos conseguido para cada uno de los tamaños con respecto a la computación sin paralelizar:



Podemos ver que cuando la matriz es pequeña, o hasta 2000x2000, el tiempo que tarda el programa cuando usamos threads a lo loco es mucho mayor, porque tarda en crear y gestionar todos esos procesos.

Sin embargo, cuando la matriz ya es muy grande, 10000x10000, el tiempo relativo que tarda en gestionar esos procesos se vuelve pequeño y conseguimos una mejora notable:

De 205 minutos de forma secuencial a 57, lo que es un SpeedUp real de 359%.

Método “división estática”

100x100

1 procesador:

```
Tiempo en leer la matriz 1: 0.577000
Tiempo en leer la matriz 2: 0.588000
Tiempo en multiplicar las matrices: 0.950000
Son iguales
```

2 procesadores:

```
Tiempo en leer la matriz 1: 0.631000
Tiempo en leer la matriz 2: 0.580000
Tiempo en multiplicar las matrices: 0.979000
Son iguales
```

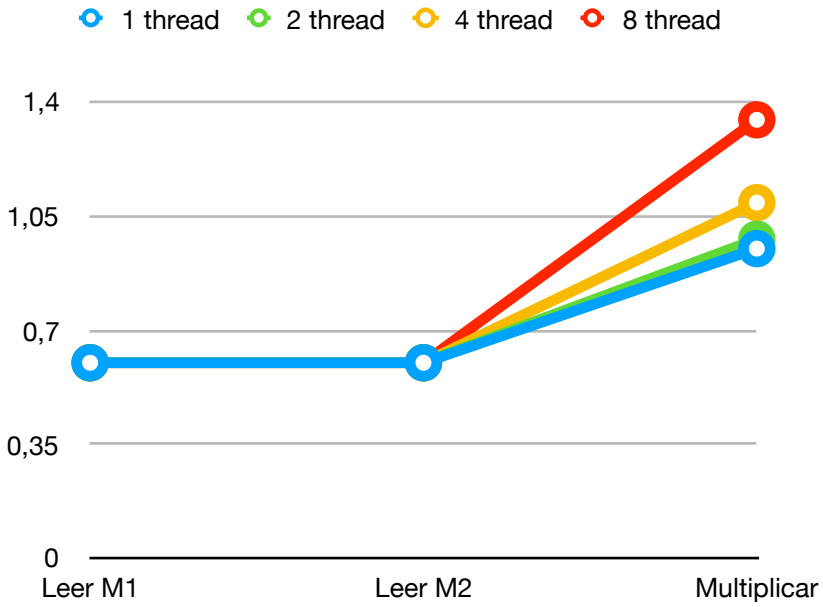
4 procesadores:

```
Tiempo en leer la matriz 1: 0.649000
Tiempo en leer la matriz 2: 0.613000
Tiempo en multiplicar las matrices: 1.091000
Son iguales
```

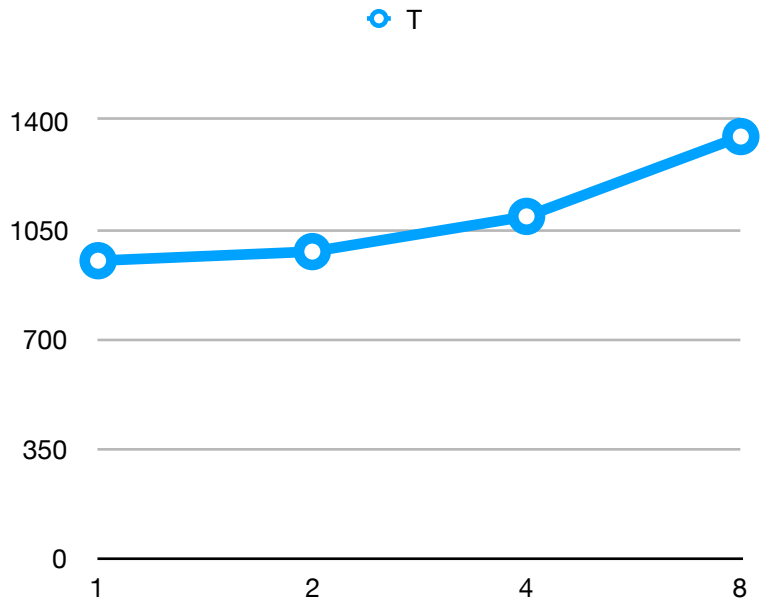
8 procesadores:

```
Tiempo en leer la matriz 1: 0.596000
Tiempo en leer la matriz 2: 0.583000
Tiempo en multiplicar las matrices: 1.345000
Son iguales
```

Para calcular los tiempos de ejecución, como las diferencias en la lectura de los datos son casuales, ya que esa parte del programa no varía en ningún momento, vamos a establecer una media en 0.6 milisegundos.



La matriz es tan pequeña que al tener que gestionar los procesos perdemos mas tiempo del que ganamos.



Resulta que en este caso, el SpeedUp real es negativo, ya que se tarda mucho tiempo en crear un thread.
Tenemos un SpeedUp relativo de 1 a 8 procesadores de -29%.
Si calculamos el SpeedUp con respecto al programa secuencial:
Secuencial: 0.0009s
Paralelizado: 1.34s
Tenemos un SpeedUp real de 99.99%…, muy alejado del SpeedUp teórico que habíamos calculado (página 5) +29.2%.
Esto significa que casi toda la carga del programa esta en la paralelización.

1000x1000

1 procesador:

```
Tiempo en leer la matriz 1: 61.649000
Tiempo en leer la matriz 2: 75.359000
Tiempo en multiplicar las matrices: 937.777000
Son iguales
```

2 procesadores:

```
Tiempo en leer la matriz 1: 61.058000
Tiempo en leer la matriz 2: 79.632000
Tiempo en multiplicar las matrices: 951.871000
Son iguales
```

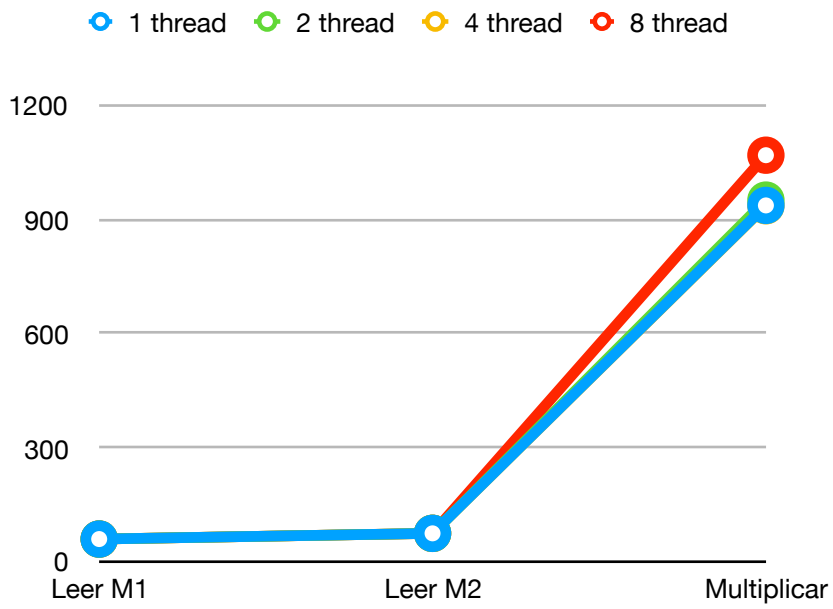
4 procesadores:

```
Tiempo en leer la matriz 1: 57.943000
Tiempo en leer la matriz 2: 77.531000
Tiempo en multiplicar las matrices: 936.545000
Son iguales
```

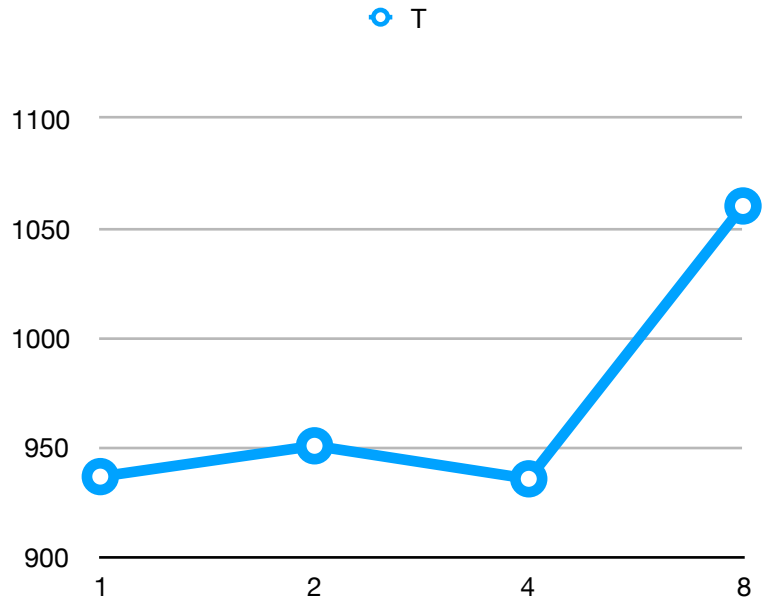
8 procesadores:

```
Tiempo en leer la matriz 1: 61.348000
Tiempo en leer la matriz 2: 69.940000
Tiempo en multiplicar las matrices: 1069.890000
Son iguales
```

Vamos a establecer la media de carga de la primera matriz en 60, y la segunda en 75.
Tarda un poco mas en leer la segunda matriz porque la lee traspuesta.



Parece que seguimos sin obtener resultados, cuantos más procesadores utilizamos, más tarda el proceso de multiplicación.
Esperaba que con una matriz de 1000x1000 ya se notase bastante la diferencia.



De nuevo tenemos un SpeedUp con respecto a 1 procesador negativo -13%
Si calculamos el SpeedUp con respecto al programa secuencial
Secuencial: 1s
Paralelizado: 1.06s
Lo que nos deja un -5% de SpeedUp real.

Empiezo a pensar que aunque estoy llamando a la función std::thread.join() tantas veces como numero de procesos que tenemos, el pc solo esta usando un procesador, y llama a estos procesos secuencialmente.

He estado buscando en stackoverflow y resulta que std::thread.join() espera a que termine el proceso para saltar a la siguiente linea, por lo que aunque estaba llamando a los threads correctamente, se estaban ejecutando uno a uno, y no todos a la vez.
Sin embargo he encontrado la función std::thread::detach(), que deja un deamon en vez de esperar a que termine el proceso.
Estoy leyendo que en realidad es lo mismo, sin embargo std::thread::detach() me da resultado y stf::thread::join() no.

Vamos a repetir las pruebas de 100 y 1000, esta vez usando std::thread::detach:

100x100

1 procesador:

```
Tiempo en leer la matriz 1: 0.722000
Tiempo en leer la matriz 2: 0.753000
Tiempo en multiplicar las matrices: 1.922000
Son iguales
```

2 procesadores:

```
Tiempo en leer la matriz 1: 0.580000
Tiempo en leer la matriz 2: 0.637000
Tiempo en multiplicar las matrices: 1.421000
Son iguales
```

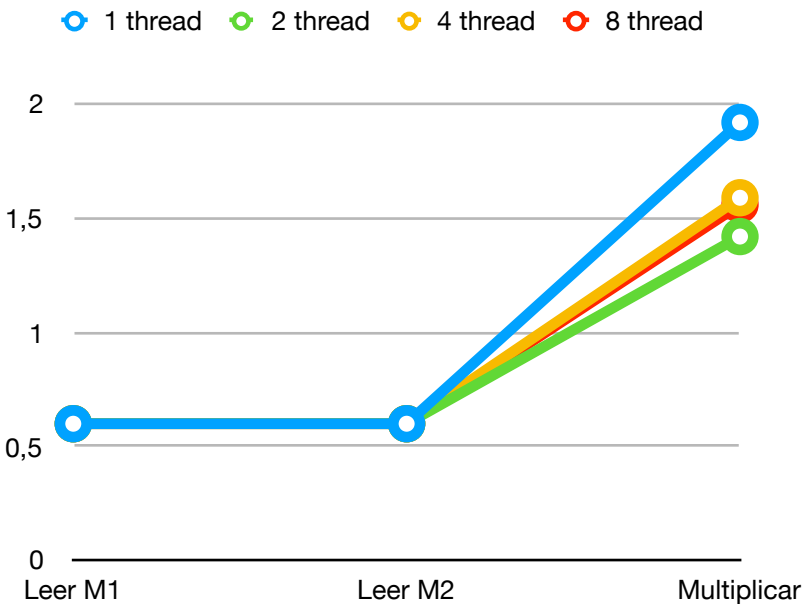
4 procesadores:

```
Tiempo en leer la matriz 1: 1.068000
Tiempo en leer la matriz 2: 0.642000
Tiempo en multiplicar las matrices: 1.592000
Son iguales
```

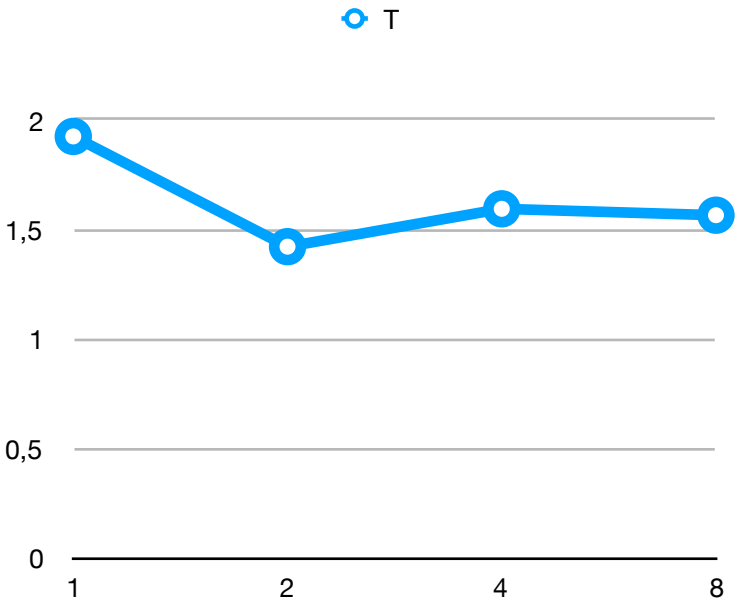
8 procesadores:

```
Tiempo en leer la matriz 1: 0.752000
Tiempo en leer la matriz 2: 0.653000
Tiempo en multiplicar las matrices: 1.564000
Son iguales
```

Vamos a establecer la media de lectura de las matrices en 0.6 milisegundos como antes.



Parece que esta vez SI estamos obteniendo una mejora, estamos gastando tiempo en saber cuándo han terminado todos los procesos, pero aún así conseguimos mejora.



Calculamos el SpeedUp secuencial vs paralizado 2 procesadores (el más rápido)
Secuencial: 0.27s
Paralelizado: 1.42s

Lo que nos deja un -81% de SpeedUp real.
La carga computacional de los procesos es MUCHO mayor que la de la multiplicación de la matriz, y esto hace que todo el proceso se ralentice.

1000x1000

1 procesador:

```
Tiempo en leer la matriz 1: 61.825000
Tiempo en leer la matriz 2: 72.647000
Tiempo en multiplicar las matrices: 1758.871000
Son iguales
```

2 procesadores:

```
Tiempo en leer la matriz 1: 59.483000
Tiempo en leer la matriz 2: 73.356000
Tiempo en multiplicar las matrices: 1351.145000
Son iguales
```

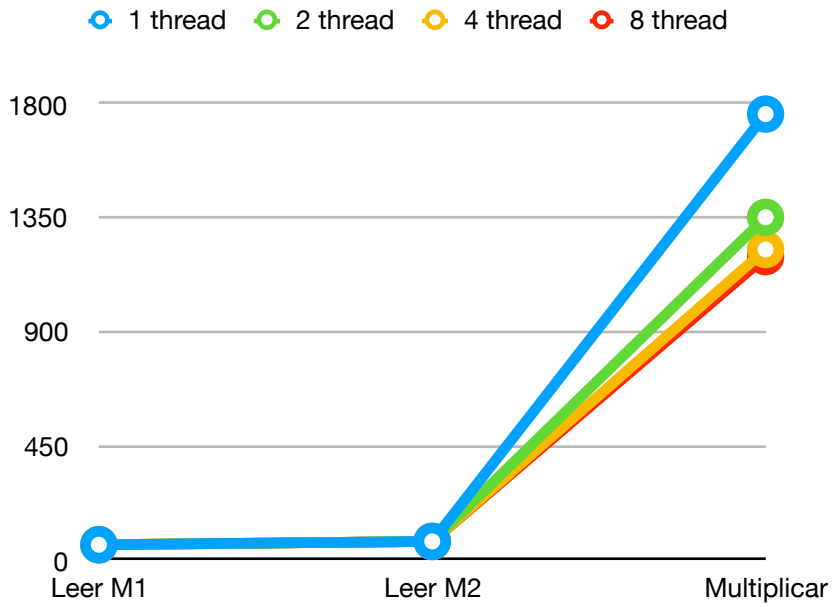
4 procesadores:

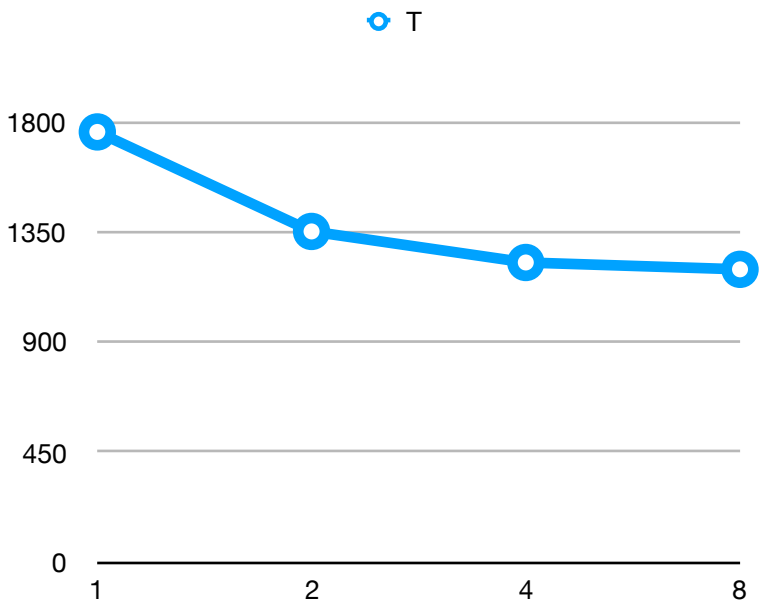
```
Tiempo en leer la matriz 1: 60.603000
Tiempo en leer la matriz 2: 73.331000
Tiempo en multiplicar las matrices: 1224.938000
Son iguales
```

8 procesadores:

```
Tiempo en leer la matriz 1: 60.400000
Tiempo en leer la matriz 2: 71.088000
Tiempo en multiplicar las matrices: 1196.133000
Son iguales
```

Vamos a establecer el tiempo de lectura de M1 en 60ms y de M2 en 73.





Ahora SI que estamos teniendo una mejora considerable aumentando el numero de procesadores, aunque en la segunda gráfica podemos ver como, según agregamos procesadores, la velocidad cada vez aumenta menos.

Calculamos el SpeedUp secuencial vs paralizado 8 procesadores
Secuencial: 1s
Paralelizado: 1.19s

Lo que nos deja un -16% de SpeedUp real.
Todavía no hemos alcanzado el tamaño de la matriz suficiente para que la carga de gestionar los procesos merezca la pena.

10000x10000

1 procesador:

```
Tiempo en leer la matriz 1: 5244.116000
Tiempo en leer la matriz 2: 6383.617000
100.0%
Tiempo en multiplicar las matrices: 1922580.947000
Son iguales
```

32 min

2 procesadores:

```
Tiempo en leer la matriz 1: 5166.846000
Tiempo en leer la matriz 2: 6397.656000
100.0%50.0%

Tiempo en multiplicar las matrices: 1494481.463000
Son iguales
```

24.9 min

4 procesadores:

```
Tiempo en leer la matriz 1: 5586.202000
Tiempo en leer la matriz 2: 7122.511000
75.0%%00.0%5.0%49.9%
100.0%25.0%
100.0%
50.0%
Tiempo en multiplicar las matrices: 1426865.290000
Son iguales
```

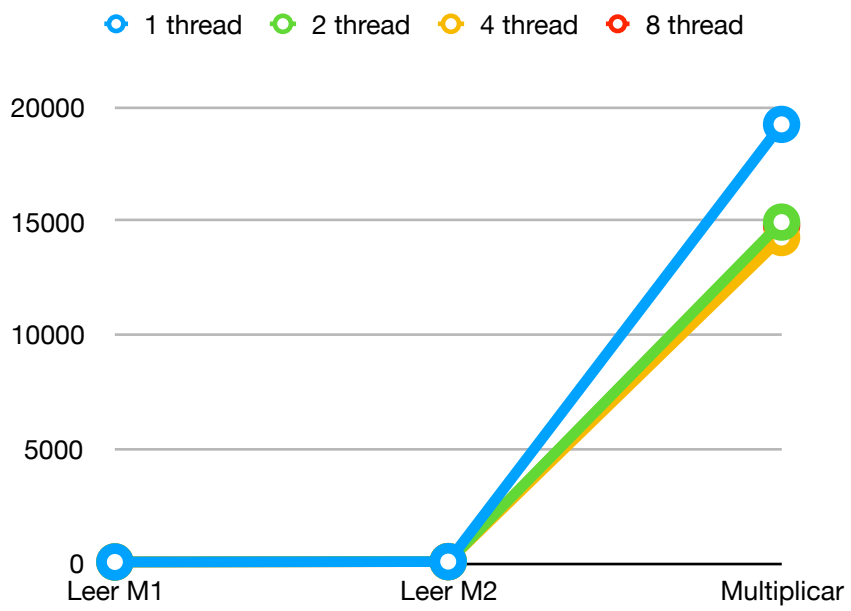
23.7 min

8 procesadores:

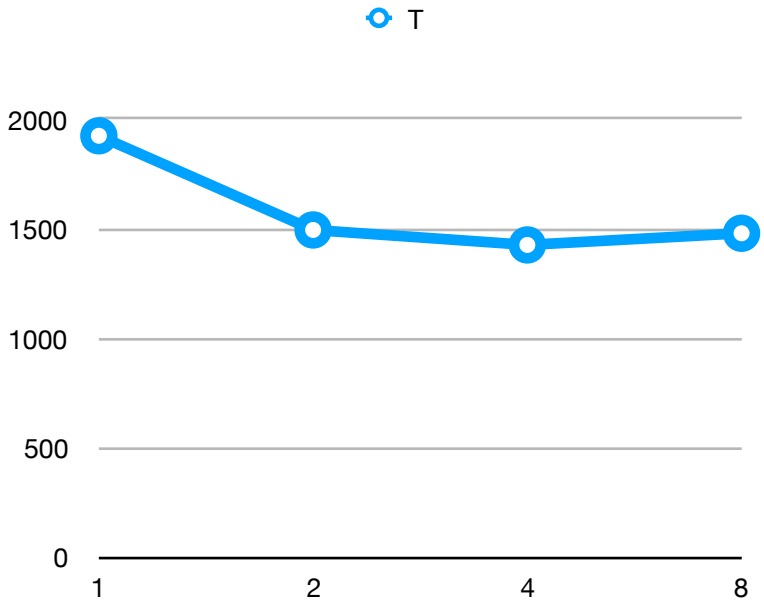
```
Tiempo en leer la matriz 1: 5757.747000
Tiempo en leer la matriz 2: 7109.647000
87.5%%9.9%49.5%57.7%
25.0%%
75.0%
50.0%
100.0%
37.5%
12.5%
62.5%
Tiempo en multiplicar las matrices: 1479545.591000
Son iguales
```

24.6 min

Vamos a establecer el tiempo de lectura de M1 en 55 segundos y de M2 en 70.



Esta vez alcanzamos una gran mejora; de 4 a 8 procesadores no hay casi cambio porque el ordenador que estoy usando solamente tiene 4 procesadores físicos, los otros 4 funcionan de manera lógica.



Calculamos el SpeedUp secuencial vs paralizado 4 procesadores
Secuencial: 12349s
Paralelizado: 1479s

Lo que nos deja un 834.9% de SpeedUp real.