UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Laboratorio N 3

Ecuación de ola de Schroedinger con programación paralela

Autores: Esteban Alarcón, Juan De Pablo.

Curso: Sistemas Operativos

Profesor: Fernando Rannou

Ayudante: Marcela Rivera

Fecha de entrega: 9 de noviembre

10 de Noviembre de 2017

Tabla de contenidos

Índice de figuras

1. Introducción

1.1. Resumen

En el presente informe se explica el desarrollo del laboratorio 3 de Sistemas Operativos, detallando la estrategia utilizada de paralelización usada para implementar la difusión de una ola según la ecuación de Schroedinger. Para esto se ocupa la herramienta de paralelización provista por el lenguaje C: *Pthreads*.

Posteriormente, se analiza el rendimiento en un computador utilizando distintas cantidades de hebras y distintos parámetros del programa.

1.2. Objetivos

1.2.1. Objetivo general

1. Aplicar la utilización de hebras de ejecución y la sincronización de éstas aplicada a la ecuación de ola de Schroedinger.

1.2.2. Objetivos específicos

1. Implementar la ecuación ola de Schroedinger en el lenguaje de programación C.
2. Aplicar programación paralela utilizando distintas cantidades de hebras de forma dinámica al algoritmo anterior.
3. Analizar el comportamiento y rendimiento del programa según la cantidad de hebras y otros parámetros de la ecuación.

1.3. Estructura del informe

En el capítulo 2 se explica la estrategia de solución con la cual se sincronizan las distintas hebras en la ejecución del programa, junto con los mecanismos utilizados.

En el capítulo 3 se muestran los resultados obtenidos a través de las distintas pruebas realizadas, modificando el número de hebras y otros parámetros del programa, como el tamaño de la grilla y el número de pasos o iteraciones.

Finalmente, en el capítulo 4 se analizan los objetivos planteados inicialmente, verificando sus dificultades, grado de cumplimiento y posibles mejoras.

2. Estrategia de solución

Hay que tener presente las distintas variables existentes, siendo las más esenciales el número de hebras, dimensión de la matriz y el número de iteraciones a realizar. El algoritmo fue implementado de forma iterativa; de esta forma se asegura que no colapsará el *stack* al acumular demasiadas iteraciones y casos pendientes, cómo podría ocurrir con un algoritmo recursivo.

Debido a que el número de hebras puede ser cualquier valor entero, este no necesariamente divide de forma equitativa o por cuadrantes a la matriz. Para solucionar este conflicto, a cada hebra se le asigna una cantidad de posiciones de forma equitativa, siendo esta desde derecha a izquierda, y por fila en orden creciente, sin considerar las posiciones de borde, las cuales son todas de valor 0 y no se modifican. Para entenderlo mejor se muestra un ejemplo.

Suponer que se tiene una matriz de 6 x 6. Las posiciones totales a repartir serán n x n - 4(n-1), es decir, el total menos las posiciones de borde. Entonces para n = 6, se tienen en total 16 posiciones.

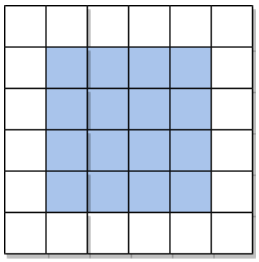

Figura 1: Cantidad de posiciones a repartir.

Suponer la cantidad de hebras a ejecutar son cinco; la repartición sería: 4, 3, 3, 3, 3. La matriz repartida queda gráficamente así:

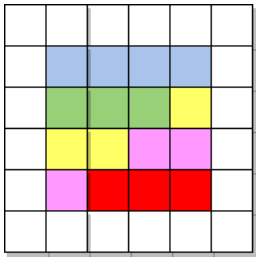

Figura 2: Matriz repartida en cinco hebras.

Debido a que cada hebra sólo escribe en una porción única de la matriz, no es necesario proteger las posiciones con semáforos u otras herramientas sincronizar. Pero resulta necesario asegurarse de que todas las hebras terminan una iteración, y ninguna puede pasar al otro instante t si otra sigue en el instante actual. En otras palabras, se debe asegurar que todas las hebras paren en ciertos puntos del programa, en este caso, la ejecución de una iteración t. Para sincronizar las hebras se implementan barreras, las cuales nos permiten acumular a las hebras en un punto específico del programa.

Cabe destacar que en el diseño de solución se crea una matriz con tres capas: instante t, t y *t-2*; cada vez que una iteración es realizada, deben copiarse cada capa en la capa inferior, para ir guardando los estados anteriores.

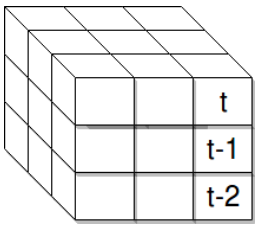

Figura 3: Matriz vista desde capas de instantes.

Por lo tanto, también dede asegurarse que las capas de la matriz sean copiadas completamente antes de comenzar la siguiente iteración, así que este es otro punto donde las hebras deben esperarse entre sí; también se usan barreras para sincronizarlas.

3. Resultados

El programa fue ejecutado en un computador con procesador i7 6700, 3.4 GHz y la cantidad de hilos paralelos es de ocho. EL tiempo fue medido a través de la consola con el comando *time*, donde se obtuvo el *real time*, el cual se define como el tiempo desde que una tarea comienza hasta que finaliza. Para calcular el *speedup (S(n))* y la *eficiencia (E(n))* se utilizaron las siguientes formulas:

$$\begin{aligned} S(n) &= \frac{T(1)}{T(n)} \\ E(n) &= \frac{T(1)}{n * T(n)} \end{aligned} \tag{1}$$

A continuacion se muestran los resultados obtenidos con las dimensiones e iteraciones solicitadas.

3.1. Ejecucion del programa con 2000 iteraciones

3.1.1. Grilla de 128x128

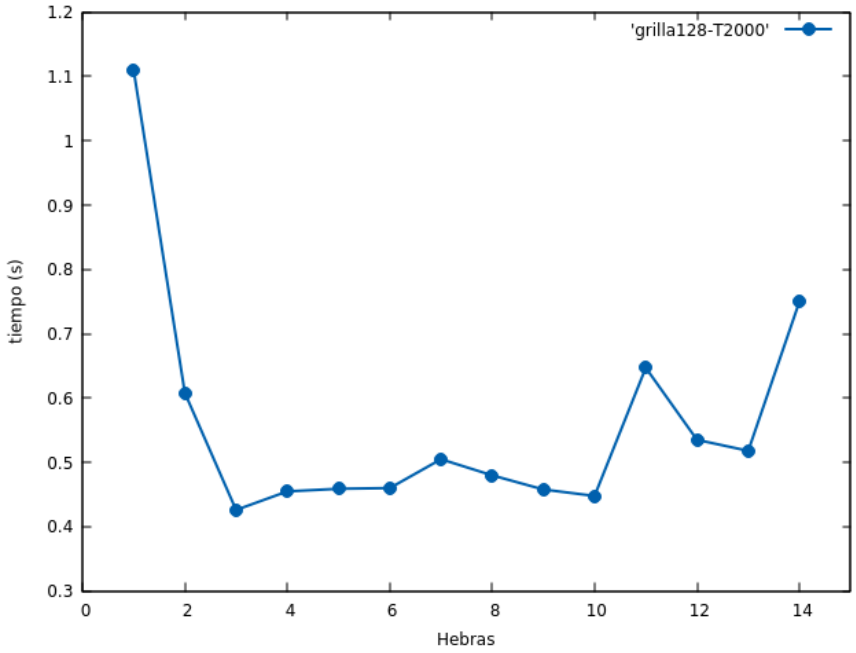

Figura 4: Hebras v/s tiempo de ejecucion, Grilla 128x128 y 2000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 3 hebras para la ejecucion del programa.

Grilla 128x128 y 2000 iteraciones.		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.809	0.912
3	2.582	0.868
4	2,417	0.609
5	2.396	0.483
6	2.413	0.402
7	2.198	0.314
8	2.312	0.289
9	2.423	0.269
10	2.477	0.247
11	1.715	0.155
12	2.074	0.172
13	2.142	0.164
14	1.480	0.105

Respecto al speedup, el mejor es alcanzado por la ejecucion del programa con 3 hebras. Mientras que Analizando la eficiencia el procesador con mejor Speedup posee una eficiencia mayor a 0,5.

3.1.2. Grilla de 256x256

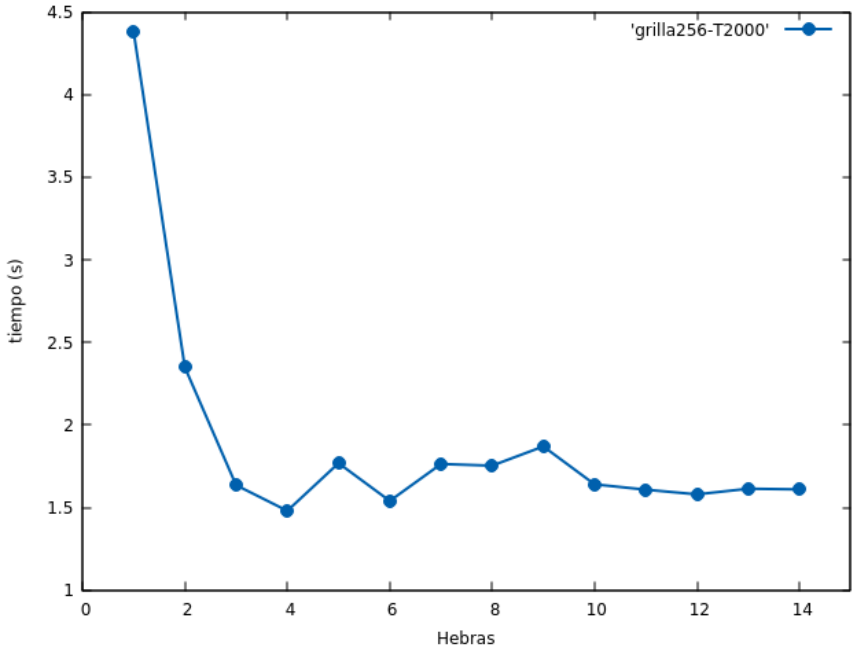

Figura 5: Hebras v/s tiempo de ejecucion, Grilla 256x256 y 2000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 4 hebras para la ejecucion del programa.

Grilla 256x256 y 2000 iteraciones		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.860	0.932
3	2.677	0.892
4	2.959	0.739
5	2.475	0.495
6	2.840	0.474
7	2.482	0.357
8	2.497	0.312
9	2.342	0.260
10	2.669	0.266
11	2.723	0.247
12	2.772	0.231
13	2.713	0.208
14	2.720	0.194

En este caso el mejor speedup es alcanzado por la ejecucion del programa con 4 hebras. Mientras que Analizando la eficiencia el procesador con mejor Speedup posee una eficiencia mayor a 0,5.

3.2. Ejecucion del programa con 4000 iteraciones

3.2.1. Grilla de 128x128

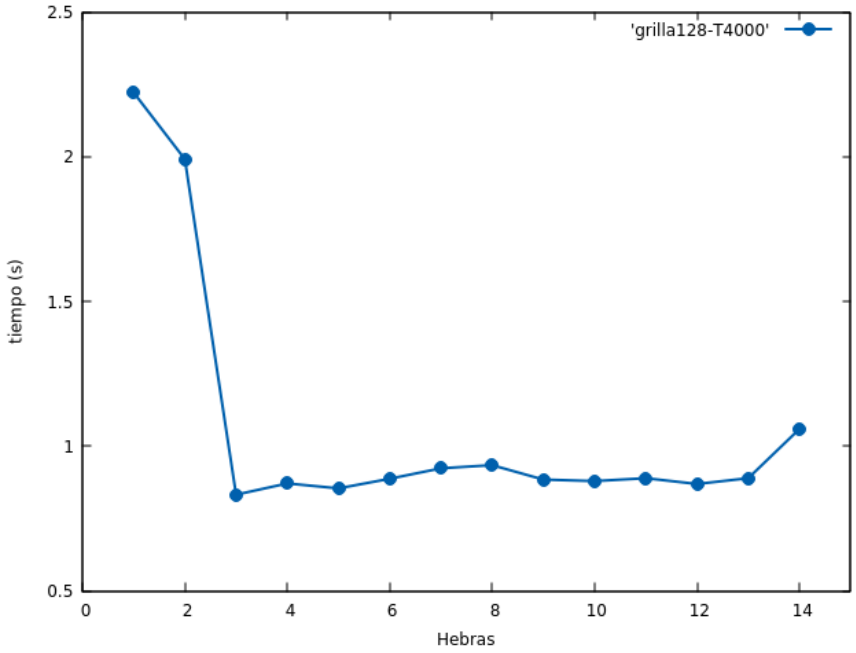

Figura 6: Hebras v/s tiempo de ejecucion, Grilla 128x128 y 4000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 3 hebras para la ejecucion del programa.

Grilla 128x128 y 4000 iteraciones		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.115	0.557
3	2.667	0.889
4	2.548	0.768
5	2.598	0.519
6	2.502	0.417
7	2.404	0.343
8	2.376	0.297
9	2.510	0.278
10	2.525	0.252
11	2.496	0.226
12	2.554	0.212
13	2.496	0.192
14	2.096	0.149

En este caso el mejor speedup es alcanzado por la ejecucion del programa con 3 hebras. Mientras que Analizando la eficiencia.

3.2.2. Grilla de 256x256

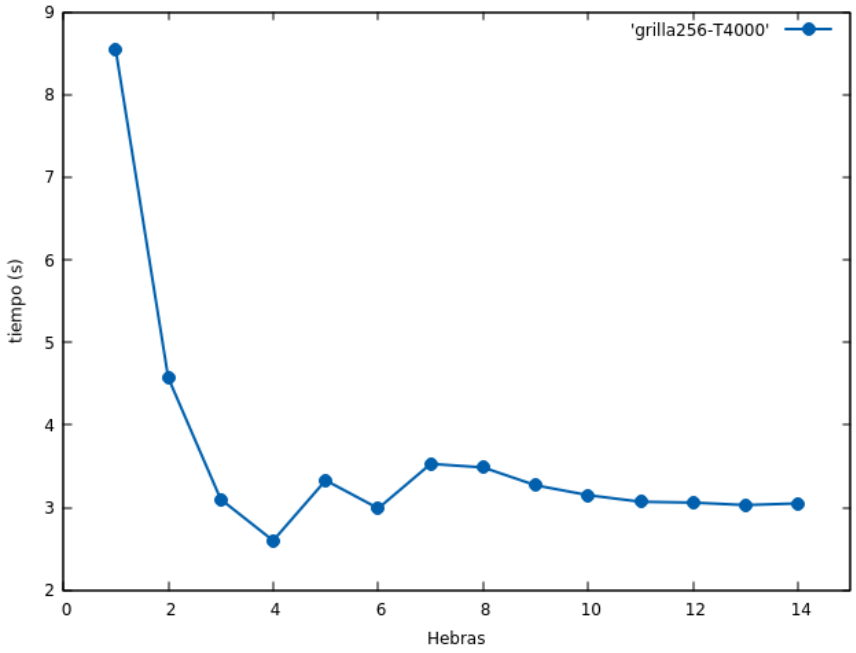

Figura 7: Hebras v/s Tiempo de ejecucion, Grilla 256x256 y 4000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 4 hebras para la ejecucion del programa.

Grilla 256x256 y 4000 iteraciones		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.870	0.935
3	2.756	0.918
4	3.288	0.822
5	2.567	0.513
6	2.859	0.476
7	2.421	0.346
8	2.452	0.306
9	2.614	0.290
10	2.710	0.281
11	2.784	0.271
12	2.772	0.253
13	2.743	0.218
14	1.821	0.200

En este caso el mejor speedup es alcanzado por la ejecucion del programa con 4 hebras. Mientras que Analizando la eficiencia, la ejecucion con 4 hebras posee una eficiencia

3.3. Ejecucion del programa con 8000 iteraciones

3.3.1. Grilla de 128x128

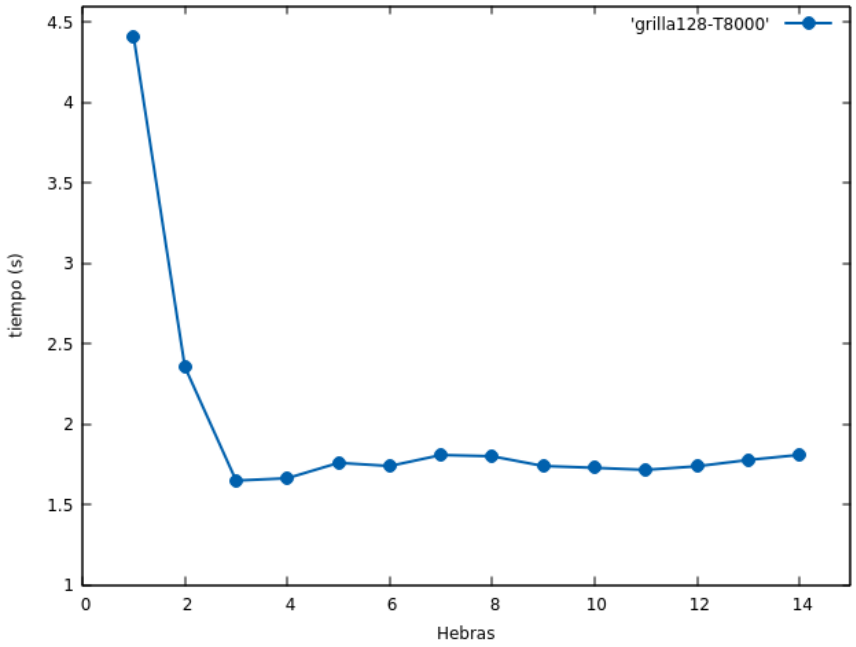

Figura 8: Hebras v/s tiempo de ejecucion, Grilla 128x128 y 8000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 3 hebras para la ejecucion del programa, por una diferencia minima si se compara con 4 hebras.

Grilla 128x128 y 8000 iteraciones		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.871	0.935
3	2.672	0.890
4	2.650	0.622
5	2.505	0.501
6	2.534	0.422
7	2.424	0.348
8	2.448	0.306
9	2.534	0.281
10	2.550	0.255
11	2.569	0.233
12	2.535	0.211
13	2.480	0.190
14	2.436	0.174

En este caso el mejor speedup es alcanzado por la ejecucion del programa con 3 hebras. Mientras que Analizando la eficiencia, la ejecucion con 3 hebras posee una eficiencia mayor a 0,5, lo cual y una eficiencia mucho mayor que utilizando 4 hebras.

3.3.2. Grilla de 256x256

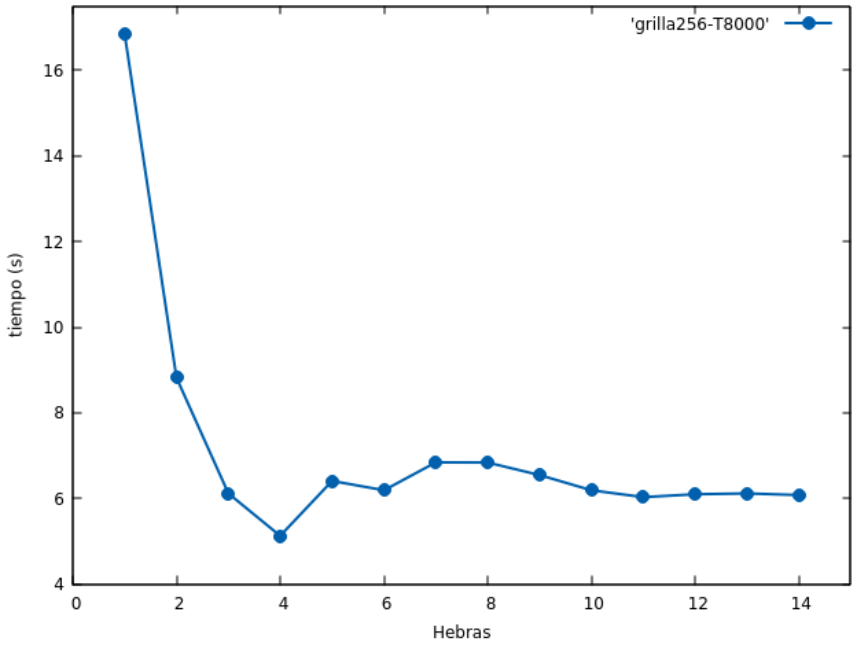

Figura 9: Hebras v/s tiempo de ejecucion, Grilla 256x256 y 8000 iteraciones.

Se visualiza que el menor tiempo es alcanzado cuando se utilizan 4 hebras para la ejecucion del programa.

Grilla 256x256 y 8000 iteraciones		
Cantidad de hebras	Speedup	Eficiencia
1	1.000	1.000
2	1.860	0.954
3	2.752	0.917
4	3.283	0.820
5	2.627	0.525
6	2.719	0.453
7	2.450	0.351
8	2.462	0.307
9	2.571	0.286
10	2.719	0.272
11	2.791	0.253
12	2.760	0.230
13	2.751	0.211
14	2.679	0.213

En este caso el mejor speedup es alcanzado por la ejecucion del programa con 4 hebras. Mientras que Analizando la eficiencia esta ejecucion es similar a la ejecucion del programa con 3 hebras.

3.4. Analisis de resultados

Respecto al tiempo de ejecución de las hebras, se puede apreciar una tendencia que entre menor sea el tamaño de la grilla, mejor resultado se obtiene para 3 hebras para el hardware utilizado, pero entre mayor sea el tamaño de la grilla se obtiene un menor tiempo de ejecución para 4 hebras. Por lo tanto, el menor tiempo de ejecución para este programa varía entre 3 a 4 hebras. En los gráficos y tablas anteriores se puede visualizar que si se utilizan más de 4 hebras, el tiempo de ejecución del programa comienza a crecer. Analizando los Speedup de las ejecuciones se reafirma lo mencionado, ya que se puede apreciar que las ejecuciones con 3 y 4 hebras poseen los Speedups más altos(o sea, que mejora la velocidad de ejecución en relación a la ejecución del programa con una hebra). Respecto a la eficiencia se aprecia que ninguna ejecución paralelizada posee una eficiencia exactamente igual a 1, y que a medida que aumenta la cantidad de hebras se llega a un punto que no disminuye el tiempo de ejecución, si no que aumenta. También mencionar que a medida que aumenta la cantidad de hebras en la ejecución, aumenta el overhead(muchas hebras quedan ociosas esperando que las que siguen en ejecución terminen).

4. Conclusiones

Retomando los objetivos específicos planteados inicialmente:

1. Implementar la ecuación ola de Schroedinger en el lenguaje de programación C.

   El algoritmo fue implementado a través de iteraciones; esto nos resguarda el uso de memoria, ya que no hay casos pendientes como en un algoritmo recursivo. Este punto es realmente importante debido a la cantidad de cálculos realizados y la optimización del uso de memoria.

2. Aplicar programación paralela utilizando distintas cantidades de hebras de forma dinámica al algoritmo anterior.

   Primeramente, se implementó el algoritmo de forma secuencial y posteriormente de forma paralela. Hubo cierta dificultad al analizar cómo paralelizarlo; una forma fue crear hebras por cada iteración, pero esto es realmente costoso al observar que podía iterar hasta 8000 veces. Luego, se analizó crear las hebras una sola vez y sincronizarlas dentro de la ejecución del programa.

   En el capítulo 2 se explica la estrategia realizada para lograr sincronizar las hebras, para así paralelizar el algoritmo. Se hicieron muchas pruebas y se verificó el correcto funcionamiento del programa usando distintas cantidades de hebras. Sin embargo, surgieron dificultades para lograr sincronizar las hebras en ciertos puntos del programa; para cumplir este objetivo se usaron barreras, la cual se aplicó correctamente.

3. Analizar el comportamiento y rendimiento del programa según la cantidad de hebras y otros parámetros de la ecuación.

   En el capítulo 3 se evidencian varios gráficos que se diferencian en los parámetros ocupados respecto al tamaño de la grilla y la cantidad de iteraciones.

   En general, se analiza que en una cierta cantidad de hebras se da el óptimo en tiempo de ejecución. Estos gráficos ayudan visualmente a comparar el potencial rendimiento que provee la programación paralela, aunque esto depende de la cantidad de hebras utilizadas y además (muy importante) es el procesador utilizado; la cantidad de hebras

óptimas depende de la cantidad de hebras que el procesador puede ejecutar paralelamente. En esta experiencia, se utilizó un procesador con 8 hebras, pero existen procesadores con menos o incluso una mayor cantidad.

Bibliografía

[1] Stallings, W., *Sistemas operativos.* 7ª ed. Madrid: Pearson Educación, 2005.

[2] Universidad de Valencia, *Organización de Computadores*, Capítulo 4. Introducción al paralelismo y al rendimiento.