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

Laboratorio 1: OpenMP - Ecuación de Schroedinger

Integrantes: César Salazar
Jorge Sandoval

Curso: HPC

Modalidad: Magister

Profesor: Fernando Rannou

10 de Octubre de 2022

Tabla de contenidos

1. Introducción

Por *High Performance Computing*, comprendemos la practica de añadir potencia a soluciones informáticas con tal de ser utilizadas para gestionar/procesar grandes volúmenes de datos a través de diferentes técnicas, estándares o recursos. Es por esto que, para comprender la importancia de la computación de alto rendimiento, sus conceptos y principios, se realizaran una serie de laboratorios para generar un acercamiento empírico a esta área, los que quedaran registrados en este documento.

Dando lugar a este laboratorio, es que se abordo *OpenMP* (Board, 2018), un estándar de programación paralela dentro de sistemas de memoria compartida, no es un lenguaje, sino un estándar con el que se facilita desarrollar aplicaciones paralelas eficientes. Para este primer laboratorio se abordara la ecuación de *Schroedinger* con tal de simular como se difunde una ola en un medio, el cual fue abordado con técnicas tanto secuencial como paralela para posteriormente contrastar sus resultados.

1.1. Objetivo

1. Implementar un simulador paralelo de la difusión de una onda según la ecuación de *Schroedinger*, usando *OpenMP* como tecnología de paralelización.

2. Descripción del problema

Se define un medio acuoso en reposo (valor 0 inicialmente), el que se perturba con un pulso inicial definido por la proporción y generando el siguiente comportamiento:

$$H^0_{i,j} = 20 \quad \forall\, 0.4N < i < 0.6N, \;\; 0.4N < j < 0.6N$$

Figura 1: Sector de la grilla que afecta el primer pulso.

Es durante el primer paso de tiempo que su estado cambia a :

$$H^t_{i,j} = H^{t-1}_{i,j} + \frac{c^2}{2}\left(\frac{dt}{dd}\right)^2\left(H^{t-1}_{i+1,j} + H^{t-1}_{i-1,j} + H^{t-1}_{i,j-1} + H^{t-1}_{i,j+1} - 4H^{t-1}_{i,j}\right)$$

Figura 2: Comportamiento en el primer paso de tiempo.

Posteriormente los estados durante cada paso de tiempo t están definido por:

$$H^t_{i,j} = 2H^{t-1}_{i,j} - H^{t-2}_{i,j} + c^2\left(\frac{dt}{dd}\right)^2\left(H^{t-1}_{i+1,j} + H^{t-1}_{i-1,j} + H^{t-1}_{i,j-1} + H^{t-1}_{i,j+1} - 4H^{t-1}_{i,j}\right)$$

Figura 3: Comportamiento en un paso de tiempo t.

Tal que el medio acuoso tiene el siguiente comportamiento en cada instante de tiempo T:

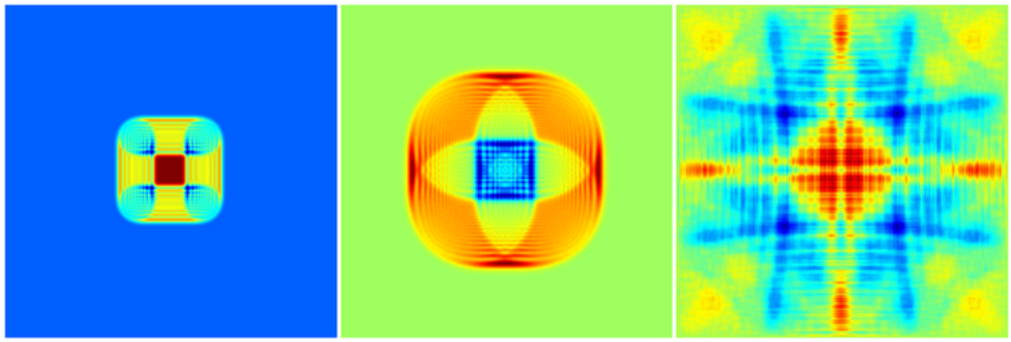

Figura 4: Resultados esperados para los pasos 300, 1000 y 10000 (para imagen de 256).

3. Estrategia de paralelización

Si bien en un inicio se pensó en segmentar los diferentes casos a resolver, para procesarlos y ejecutarlos por separado dando lugar a una estrategia SIMD (*Single Instruction Multiple Data*). Concluimos que seria curioso revisar el comportamiento de múltiples instrucciones en diferentes datos (MIMD), por lo que, decidimos integrar dentro del for de pragma que ejecuta la función *Schroedinger* bifurcaciones en las que los datos tuvieran diferentes comportamientos, forzando la integración de los casos bordes tanto como en pasos como en posiciones (Bordes tienen valores específicos). Así mismo implementamos dos pequeños ejemplos del estilo SIMD al gestionar los datos de las matrices como copiar o modificar una de ellas (Ver figuras 5 y 6) para revisar su implementación.

```
/*  FUNCION: copiar_matriz.
    ENTRADA: void
    PROCESAMIENTO: Copia una matriz.
    SALIDA: void
*/
void copiar_matriz_paralelo(float *matriz_origen,float *matriz_destino, int N, int numHebras){
    #pragma omp parallel shared(matriz_origen, matriz_destino) num_threads(numHebras)
    {
        //collapse: se utiliza para convertir un bucle anidado en un solo bucle que será paralelizado.
        #pragma omp for collapse(2)
        for(int i=0; i<N; i++){
            for(int j=0; j<N; j++){
                matriz_destino[i*N+j] = matriz_origen[i*N+j];
            }
        }
    }
}
```

Figura 5: SIMD: Copiar Matrices

```
/*  FUNCION: Matriz_ceros.
    ENTRADA: Matriz inicializada
    PROCESAMIENTO: Rellena toda la matriz con ceros.
    SALIDA: Matriz de ceros.
*/
int matriz_ceros_paralelo(float *matriz, int N, int numHebras){
    #pragma omp parallel shared(matriz) num_threads(numHebras)
    {
        //collapse: se utiliza para convertir un bucle anidado en un solo bucle que será paralelizado.
        #pragma omp for
        for(int i=0; i<N; i++){
            for(int j=0; j<N; j++){
                matriz[i*N+j] = 0.0;
            }
        }
    }
    return 0;
}
```

Figura 6: SIMD: Llenar Matriz con ceros

4. Rendimiento computacional

Uno de los aspectos que generaron mas curiosidad durante la elaboración de pruebas era la diferencia en la ejecución entre la versión secuencial y la versión paralela. En particular las cuestiones de tiempo con tal de justificar el uso respectivo de hebras con ejemplos concretos. Bien fue el caso entregado con un N igual a 256 y 12 hebras, pero con el que se fue variando su T, desde 10000, 100000, 1000000 y 10000000 pasos. Dejando claramente una diferencia sustancial entre ambas versiones (Ver figura 7).

Sin embargo, se realizo un registro intensivo (Ver figura 8 y 9) de ejecuciones con un N igual a 256, 512 y 1024 en diez mil y mil iteraciones. Con tal de analizar algún patrón en particular en el aumento de hebras a diferente nivel de carga en este problema en particular. Si bien se puede observar claramente una disminución del tiempo al aumento de hebras, este

tiene un comportamiento extraño cuando sube a la hebra 16-17. Sin embargo, queda claro que al aumentar las hebras se reparte la carga a procesar y por ende mejora en términos de tiempo.

```
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ make
gcc -Wall -fopenmp   -c -o wave.o wave.c
gcc -Wall -fopenmp   -c -o funciones.o funciones.c
gcc -Wall -fopenmp -o wave wave.o funciones.o -lm
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 10000 -H 12 -f archivoSalida.raw

Ejecucion secuencial: 7.038603 (segundos)

Ejecucion paralela: 0.688842 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 100000 -H 12 -f archivoSalida.raw

Ejecucion secuencial: 69.423960 (segundos)

Ejecucion paralela: 7.019957 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 1000000 -H 12 -f archivoSalida.ra

Ejecucion secuencial: 697.591456 (segundos)

Ejecucion paralela: 70.608298 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 10000000 -H 12 -f archivoSalida.r

Ejecucion secuencial: 6963.881099 (segundos)

Ejecucion paralela: 696.486219 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$
```

Figura 7: Tiempos de ejecución.

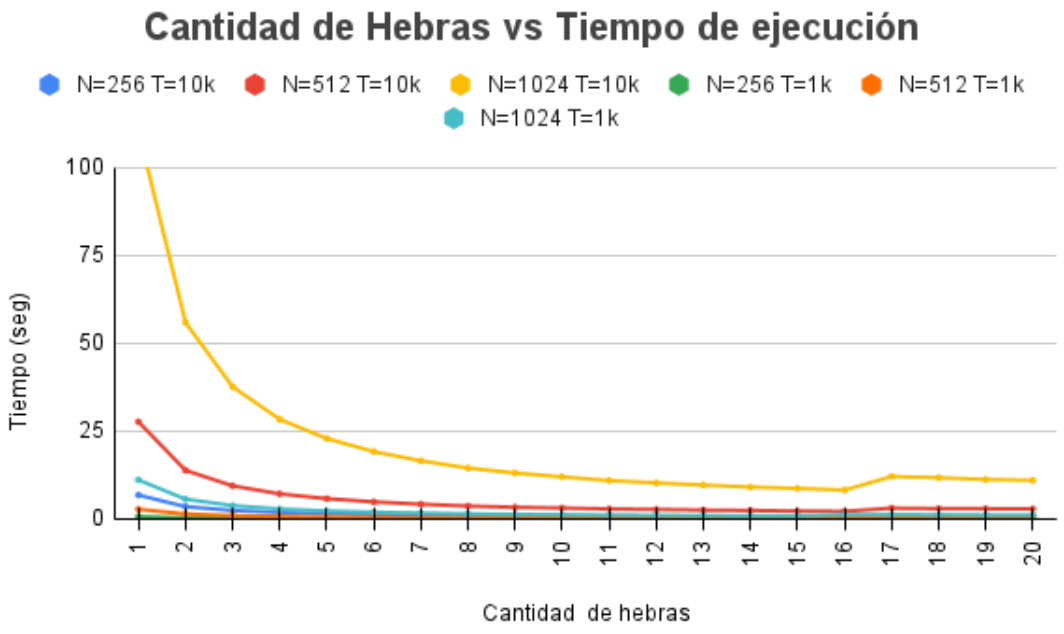

Figura 8: Cantidad de Hebras vs Tiempo de ejecución

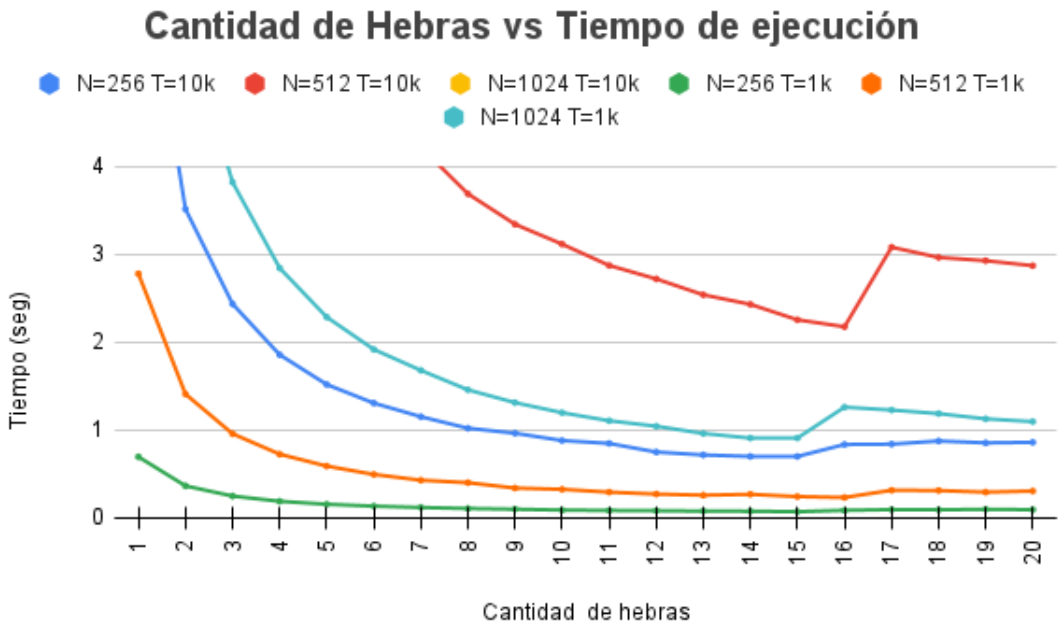

Figura 9: Cantidad de Hebras vs Tiempo de ejecución (Zoom)

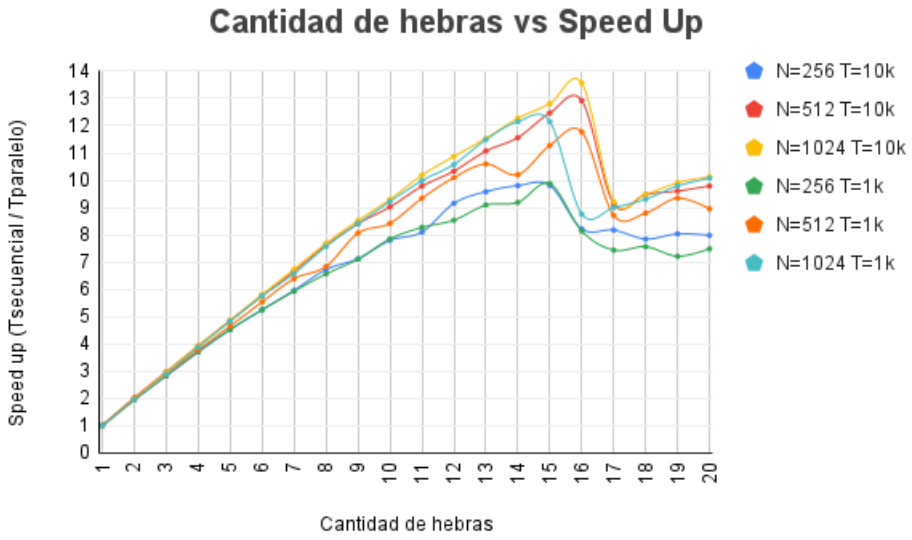

Figura 10: *Speedup* vs Cantidad de hebras

Como se puede ver en la figura 10, se presenta el gráfico donde se realiza la comparación entre el *Speedup* (versión simple), que es la división entre el tiempo secuencial por el tiempo paralelo, con el fin de comparar ambos tiempos. Tal como en los tiempos de

ejecución, quienes vieron una notable mejoría, el *Speedup* aumenta igualmente, aunque se hace presente la caída de este valor para cuando alcanza los valores 15-17. Examinando con mayor detalle el equipo que ejecutaba las pruebas, se detecto que presentaba 16 núcleos / 32 hilos en su procesador, por lo que es debido a ello la baja del *Speedup* en ese umbral.

Las pruebas se ejecutaron en un procesador Ryzen 9 5950x con sistema operativo Ubuntu 22LTS. Otros aspectos de hardware importantes son 240 GB WDC WDS240G2G0A-00JH30 (SSD) y 32GB (2 x 16GB — DIMM DDR4-3200 RAM).

5. Conclusiones

Durante el desarrollo de este laboratorio, se puede evidenciar que se realizó correctamente la simulación de la difusión de una onda según la ecuación de Schroendinger utilizando la tecnología de *OpenMP* en el lenguaje de programación C.

Se puede evidenciar que frente a un procesamiento secuencial, el uso de *OpenMP* facilita la creación, gestión y control de hebras, lo que permite disminuir considerablemente los tiempos de procesamiento. El uso de este recurso para grandes cálculos repetitivos (como los que ocurren al operar en una matriz) permite utilizar de mejor manera los recursos del hardware disponible para su procesamiento. Dependiendo del procesador donde se ejecute el código se podrá obtener un procesamiento concurrente o paralelo, que siempre será mejor a una ejecución secuencial.

Durante la experiencia se encontró la limitante en cuanto a la gestión de recursos paralelos con el procesador siendo este la cantidad de núcleos disponibles para crear hebras, evidenciándose tanto en el *Speedup* como en los conteos de tiempo referente a la cantidad de hebras destinadas para su ejecución. Tanto así como sobrepasar la cantidad física de núcleos, se nota la presencia de concurrencia en los núcleos, ya que disminuye el *Speedup* aumentando la carga en cada núcleo.

Otro factor que surgió durante esta experiencia, fue la dificultad de recordar C y su gestión de variables y asignación de memoria debido al tiempo sin programar. Pero que tras un par de ensayos, se pudo recuperar conocimiento que esperamos aplicar en próximas experiencias.

Bibliografía

Board, O. A. R. (2018). OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. November 2018.