

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



HPC Laboratorio 1: SIMD- OpenMP

Nombres: Ekaterina Cornejo Contreras

Felipe Villalobos Padilla

Rut: 20.187.903-5

20.139.310-8

Profesor: Fernando Rannou Fuentes

9 de Octubre de 2022

Tabla de contenidos

1. Introducción	1
2. Desarrollo	2
2.1. Contexto	2
2.2. Programación	3
2.2.1. Secuencial	3
2.2.2. Paralelización	3
3. Equipo	5
4. Análisis	5
5. Conclusión	6
Referencias	8

1. Introducción

El proceso de paralelización consiste en transformar un programa secuencial en una nueva versión concurrente semánticamente equivalente. Este proceso puede ser realizado de forma automática o manualmente programando el algoritmo en un lenguaje de programación paralelo que permita especificar tareas u operaciones a ejecutar de forma concurrente, de manera que la finalidad es disminuir el tiempo de procesamiento mediante la distribución de tareas entre los procesadores disponibles. En el siguiente laboratorio se presentará el proceso de paralelización sobre la simulación de la difusión de una ola en un medio a partir de un estado inicial, empleando la ecuación de Schroendinger.

Objetivo

El objetivo del siguiente informe es presentar el desarrollo y resultados generados al implementar un simulador paralelo de la difusión de una onda según la ecuación de Schroendinger, usando OpenMP como tecnología de paralelización .

2. Desarrollo

2.1. Contexto

Se solicita simular cómo se difunde una ola en un medio, por a partir de un estado inicial, donde para iniciar se discretizará el dominio de difusión en una grilla de $N \times N$ celdas. En particular, la matriz H contendrá la altura de la onda en cada posición de la grilla. Además, como el proceso de difusión es dependiente del tiempo, H_{ij}^t es la altura, ya sea positiva o negativa, de la onda en la celda (i, j) , en tiempo t . La ecuación de Schroendinger dice que:

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

Figura 1: Ecuación de Schroendinger

Donde "c" es la velocidad de la onda en el medio, "dt" es el intervalo de tiempo con que avanza la simulación y "dd" es el cambio en la superficie. Para este laboratorio, usaremos $c = 1.0$, $dt = 0.1$, y $dd = 2.0$. Además de esto, se deberán considerar las condiciones de borde:

La condición de borde para este problema es cero, es decir, la altura en las celdas de los bordes del dominio permanece siempre con valor cero ($H_{0,j}^t = H_{i,0}^t = H_{N-1,j}^t = H_{i,N-1}^t = 0$ Para todo i,j,t).

También, la iteración para $t = 1$, es la siguiente ecuación:

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

Figura 2: Ecuación para primera iteración

2.2. Programación

2.2.1. Secuencial

Para el desarrollo de la simulación, se inició generando el código con las condiciones antes mencionadas de manera secuencial (previo a cualquier tipo de paralelización).

Para esto inicialmente se generó el ingreso de los parámetros que serán requeridos por consola (N = tamaño de la grilla, T = número de pasos, f = archivo de salida, H = número de hebras a emplear, $t=$).

Tras esto se procede al llenado inicial de la grilla (matriz) de $N \times N$ con valores 0 en los casos borde y 20 en el resto de los puntos (posiciones) de la grilla. Una vez hecho esto se procede a realizar el cálculo de la ecuación para cada una de las posiciones de la grilla considerando el caso de la primera iteración, cuyo cálculo poseía características específicas (tal como se mencionó anteriormente) y tras esto se continúa hasta que se realicen el número de pasos solicitados por consola. Además de esto, cada "t" número de iteraciones, se generará un mensaje por consola.

Finalmente, se generará un archivo (.raw) con el resultado obtenido tras T iteraciones realizadas llamado "ejemplo.raw".

2.2.2. Paralelización

Para el proceso de paralelización en primer lugar se realizó la paralelización del primer llenado de la grilla (inicialización) empleando:

```
#pragma omp parallel shared(H1,H2) num_threads(H)
#pragma omp for schedule(static, H) collapse(2)
```

En primer lugar, se definen como variables compartidas $H1$ y $H2$ quienes contienen los valores de la grilla en dos momentos de tiempo diferentes, y se define el número de hebras según el valor entregado por consola " H ". Tras esto, previo a los dos "for" anidados, se emplea la segunda línea antes mostrada (omp for), incluyendo un "schedule static" de modo que se planifica que (de manera estática) cada una de

las hebras recibirá porciones del "for" anidado de tamaño "H", además de esto, el uso de "collapse(2)" permitirá para especificar que se deben contraer los dos "for" anidados en uno solo. Por otro lado, se paralelizará el proceso de cálculo de la grilla comenzando con un:

```
#pragma omp parallel shared(H1,HAUX,H2,contador,T)...  
...private(jMas1,jMenos1,iMenos1,iMas1) num_threads(H)
```

Esto es aplicado previo al bucle "while" donde se encierra el cálculo de la grilla. De esta forma se definen como variables compartidas los cálculos de las grillas en "H1" y "H2", junto con "HAUX" que funciona como un auxiliar para almacenar la grilla en un estado particular, un contador que indica el número de iteraciones realizadas y "T" con el valor total de iteraciones posibles. Por otro lado, se definen las variables privadas que solo serán vistas por cada hebra, de manera que no existan inconsistencias conteniendo los valores temporales de algunas casillas dentro de la grilla y finalmente el número de hebras que actuarán "H". Luego de esto se emplea sobre cada uno de los ciclos for dobles:

```
#pragma omp for schedule(static) collapse(2)
```

En este caso se solicitará que la paralelización se programe de manera estática; sin embargo, el número de iteraciones entregadas a cada hebra no se define, de modo que sea el mismo sistema quién decida la mejor repartición del trabajo (se realizaron pruebas con algunos ejemplos comprobando que la velocidad aumentaba levemente con este cambio) y nuevamente se condensarán dos "for" en uno con el uso de "collapse". Junto con esto se aplicará el uso de:

```
#pragma omp single
```

Con este se podrá apartar las zonas críticas que implican el cambio de valores de variables generales y aumento del contador de iteraciones. Para esto se aplicó "single" debido a que solo se necesita que una hebra realice esta acción al finalizar los cálculos de los ciclos

```
143     #pragma omp single
144     {
145         swap(HAUX,H2,N);
146         contador++;
147     }
148
```

Figura 3: Uso de "pragma omp single"

"for" y antes de estos, por lo que emplear "critical" generaría errores de inconsistencia.

3. Equipo

El equipo utilizado para las pruebas del programa corresponde a un computador portátil, con una máquina virtual con el sistema operativo de Ubuntu 18.04 con las siguientes especificaciones técnicas:

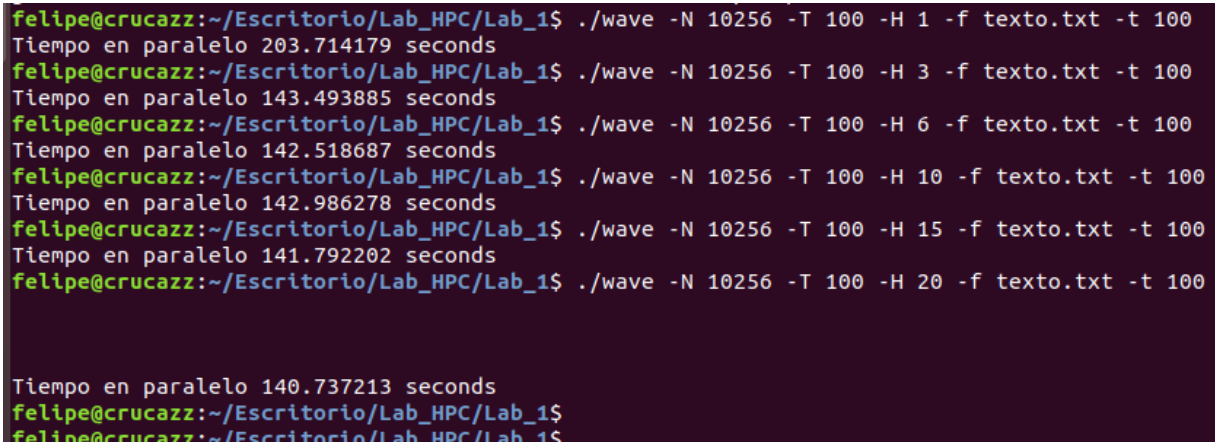
- Modelo de Procesador: Intel(R) Core(TM) i5-1135G7 @2.40GHz
- Cantidad de núcleos: 3
- Memoria RAM: 8 Gb

4. Análisis

Durante el proceso de pruebas se obtuvieron diversos resultados. En primero lugar, al iniciar con algunos ejemplos se pudo observar que no había grandes diferencias entre los resultados obtenidos entre el uso de un número de hebras y otro, luego se descubrió que esto se debía al número de procesadores que se había dispuesto a la máquina virtual empleada y que, por lo tanto, no se estaba explotando el paralelismo; sin embargo, luego de agregarle más, se pudo ver el ahorro de tiempo que se generaba al paralelizar.

Posteriormente, se realizó pruebas con grillas de distinto tamaño para observar la diferencia entre el uso de una hebra y números mayores, notando que él en caso de grillas pequeñas (iguales o menores a $N = 1000$) el cambio en el tiempo era extremadamente pequeño

e incluso nulo, mientras que en grillas de mayor tamaño y que, por lo tanto, requerían una gran cantidad de ciclos para dividir entre las hebras, la diferencia era mucho más notoria.



```
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 1 -f texto.txt -t 100
Tiempo en paralelo 203.714179 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 3 -f texto.txt -t 100
Tiempo en paralelo 143.493885 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 6 -f texto.txt -t 100
Tiempo en paralelo 142.518687 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 10 -f texto.txt -t 100
Tiempo en paralelo 142.986278 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 15 -f texto.txt -t 100
Tiempo en paralelo 141.792202 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$ ./wave -N 10256 -T 100 -H 20 -f texto.txt -t 100
Tiempo en paralelo 140.737213 seconds
felipe@crucazz:~/Escritorio/Lab_HPC/Lab_1$
```

Figura 4: Resultados para grilla $N = 10.256$

De esta forma se pudo comprobar:

- OMP es más útil en aplicaciones que manipulen arreglos y matrices grandes (dado que para los casos de pequeños arreglos no hubo gran cambio en el tiempo de compilación, sin importar el número de hebras)
- Los compiladores explotan la arquitectura propia (Dado que al realizar pruebas y aumentar el número de núcleos integrados a la máquina virtual utilizada, se pudieron observar reales cambios en el tiempo de procesamiento)
- Si bien la diferencia entre una hebra y otra es notable, a medida que aumenta el número de hebras, el ahorro de tiempo comienza a disminuir, lo que indicaría que el número óptimo de hebra, dependerá de cada caso particular y en ocasiones hay que decidir hasta qué punto es realmente conveniente usar cierta cantidad en base al tiempo ahorrado y el trabajo del sistema.

5. Conclusión

Finalmente, en base a los resultados obtenidos, se puede concluir, por un lado, que efectivamente el paralelizar los procesos realizados puede ayudar a optimizar el tiem-

po empleado, especialmente cuando se trata de elementos a gran escala, pero para esto es importante considerar el contexto de la situación y evaluar hasta qué punto vale la pena aplicar paralelización y cuando esta termina generando un mayor costo que en el caso de no emplearla. Por otro lado, cuando se paraleliza se debe tener en cuenta que existen muchas medidas distintas que pueden ser aplicadas, tanto en especificaciones, como definiciones entregadas al momento de guiar, como serán organizadas las hebras y tratados los datos que serán entregados o manejados por estas. No todos los métodos son tan efectivos en todos los casos y no siempre los valores serán igualmente efectivos al aplicar ciertas especificaciones. El número de hebras, si bien es un factor que influye en la optimización del tiempo, también lo hará el cómo son distribuidas las tareas entre estas, de modo que probar fue de gran utilidad en esta experiencia. De igual forma se reconoce que existen muchas otras formas de paralelizar el código generado y que podrían ser aún más efectivas, dado que, como se mencionó anteriormente, hay muchas variantes que afectan el resultado y aun muchos más comandos y acciones que pudieron ser aplicadas y que de conocerse, significarán una gran herramienta a aplicar en cualquier área de trabajo computacional.

Referencias

Rannou, F. (2022a). High Performance Computing, Departamento de Ingeniería en Informática LAB1 : SIMD- OpenMP.

Rannou, F. (2022b). High Performance Computing SIMD con hebras. OpenMP.

University of Colorado Boulder (2022). Fundamentals of parallel programming — Research Computing University of Colorado Boulder documentation. [Online] <https://curc.readthedocs.io/en/latest/programming/parallel-programming-fundamentals.html>.