Universidad Nacional Autónoma de México

Facultad de Ingeniería
Estructura De Datos y Algoritmos II

Proyecto 2:
Programación paralela

Profesor:
Edgar Tista García

Equipo: 8

Integrantes:
Barrios Aguilar Dulce Michelle
Chong Hernández Samuel
Mendoza Hernández Carlos Emiliano

Semestre 2023-1

9 de enero de 2023

Índice

1. Objetivos

1.1. Objetivo general

Que el alumno ponga en práctica los conceptos de la programación paralela a través de la implementación de un algoritmo paralelo, así mismo desarrolle su capacidad para responder preguntas acerca de un concepto analizado a profundidad.

1.2. Objetivo del equipo

Ratificar los conocimientos adquiridos sobre procesos paralelos mediante la exposición del diseño, análisis e implementación de un algoritmo en su versión paralela versus su versión secuencial. Hacer énfasis en la teoría del paralelismo de uno o varios procesos mediante definiciones y ejemplos para una mejor comprensión, logrando de esta manera distinguir si existe una mejora o no al paralelizar el algoritmo en cuestión.

De igual manera, llevar a cabo satisfactoriamente el trabajo colaborativo para la realización de este proyecto.

2. Introducción

Para darle solución a un problema debemos analizarlo extrayendo toda la información posible para ajustar las necesidades del método resolutivo y generar a partir de dichas necesidades las diferentes alternativas para su solución, diseñando de esta manera la que mejor se ajusta y comenzar a implementarla, siempre procurando cumplir con dos de los aspectos más importantes en nuestro campo: eficacia y eficiencia del programa. Sin darnos cuenta, la mayoría de las veces terminamos dando una solución al problema de programación realizando un programa secuencial en donde las instrucciones se realizan una tras otra.

El paralelismo es una manera novedosa de solucionar problemas de programación en los que se requieren tiempo elevado de cómputo. En un programa paralelo se tienen acciones que se pueden realizar de una forma simultánea, pero ejecutándose de forma independiente por diferentes unidades de procesamiento mediante una red de comunicación por lo que se requiere de una computadora que tenga 2 o más unidades de procesamiento.

La razón de existir de la programación paralela es encontrar una mejora respecto a la programación secuencial aumentando de esta manera la velocidad en ejecución reduciendo en tamaño en cuanto al producto informático y costos. Algunos algoritmos paralelizables representan una mejora considerable en la ejecución de un programa paralelo respecto a su contraparte secuencial.

En el presente documento profundizaremos la paralelización del algoritmo Transformada rápida de Fourier que, se considera, uno de los descubrimientos más importantes de la segunda mitad del siglo XX, y su estudio e implementación ha permitido que se utilice permanentemente por casi todo dispositivo tecnológico que maneja cualquier tipo de información, tanto para transmitirla como para manipularla.

3. Antecedentes

3.1. Algoritmo

Una vez realizado el análisis de un problema computacional, se debe de identificar el conjunto de entrada y salida para proceder al diseño de la solución, hablamos de la generación del algoritmo.

Un algoritmo es una lista bien definida ordenada y finita de instrucciones que dado un estado inicial y una entrada a través de pasos sucesivos y bien definidos se llega a un estado final obteniendo una solución del problema computacional.

Características más importantes de un algoritmo

- **Preciso**: pasos ncecesarios y sin renudar procesos inncesarios.
- **Finito**: determinado número de instrucciones
- **Definido**: no debe ser ambiguo (dobles interpretaciones).
- **Efectivo**: debe de resolver el problema.
- **Correcto**: debe cumplir con el objetivo esperado.
- **Eficiente**: debe seralizarse en el menor tiempo posible.

Ya que hablamos de caracteristicas que nos conciernen como que sea finito y eficiente, es necesario hablar de la **complejidad**.

3.2. Complejidad

Cuando hablamos de complejidad, sabemos que su principal propósito es analizar el comportamiento de los algoritmos que permiten resolver un problema y determinar la **eficiencia** basándose en 2 objetivos fundamentales:

- Medir o estimar la cantidad de recursos necesarios para resolver problemas computacionales.
- Clasificar algoritmos de acuerdo a su dificultad computacional.

El tiempo empleado por el algoritmo se mide en instrucciones, independientemente del software, compilador o lenguaje de programación que influya en el análisis. De esta manera, el coste depende del tamaño de los datos. En la evaluación del coste, se toman en consideración tres posibles casos:

- Coste promedio
- Coste mejor
- Coste peor

El tiempo requerido por un algoritmo es función del tamaño de los datos, por esta razón, la complejidad se expresa de la siguiente manera:

$$T(n)$$

Sin embargo , la notación más común es la Big-O:

$$f(n) = O(g(n))$$

A grandes rasgos, podemos identificar 4 tipos de comportamientos:

- Lineal: muestran un comportamiento predecible, si se duplica el tamaño de la entrada, se requerirá el doble de tiempo para resolverlos.
- Logarítmico: permiten atacar problemas muy grandes, ya que una entrada de doble de tamaño solo requiere un poco más de tiempo.

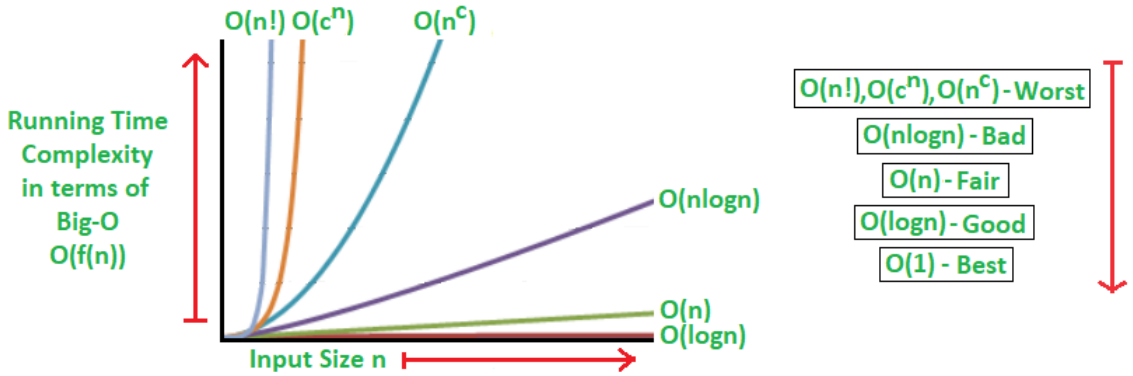
- Polinomial: representan un punto intermedio entre lo posible y no posible , la mayor parte de los casos se podrán resopver satisfactoriamente para entradas suficientemente grandes
- Exponencial: solamente funciona para entradas muy pequeñas

De manera creciente, la clasificación de la categoría de complejidad O-grande es la siguiente:

$O(1)$	Constante
$O(logn)$	Logarítmico
$O(n)$	Lineal
$O(n\ log\ n)$	Lineal-logarítmico
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial
$O(n!)$	Factorial

Es importante determinar los valores frontera de un algoritmo

- **Peor Caso:** el que exige más recursos para ser resuelto
- **Caso Promedio:** tiempo aproximado para un ejemplar típico. Se calcula la media del tiempo para todos los posibles ejemplares de tamaño n asumiendo distribución uniforme
- **Mejor Caso:** aquella instancia que requiere menos recursos



3.3. Paralelismo

Esta técnica se basa en el principio de que ciertas tareas se pueden dividir en partes más pequeñas que se pueden resolver simultáneamente. La computación paralela se ha convertido en el paradigma dominante en la fabricación de procesadores, por lo que es fundamental conocer no solo las aplicaciones actuales de esta forma de computación sino también su relevancia en el futuro.

La computación paralela es el uso de múltiples recursos informáticos para resolver un problema. Se diferencia de la computación secuencial en que pueden ocurrir múltiples operaciones al mismo tiempo.

Ventajas del paralelismo

- Resuelve problemas que no se podrían realizar en una sola CPU
- Resuelve problemas que no se pueden resolver en un tiempo razonable
- Permite ejecutar problemas de un orden y complejidad mayor
- Permite ejecutar código de manera más rápida

- Permite la ejecución de varias instrucciones en simultáneo
- Ofrece mejor balance entre rendimiento y costo que la computación secuencial
- Gran expansión y escabilidad

Desvetajas del paralelismo

- Altos costos por producción y mantenimiento
- Número de componentes usados es directamente proporcional a los fallos potenciales
- Retardos ocasionados por comunicación ente tareas
- Dificultad para lograr una buena sincronización y comunicación entre las tareas
- Mayor dificultad en las modificaciones del programa
- Condiciones de carrera
    - Múltiples procesos se encuentran en condición de carrera si el resultado de los mismos depende del orden de su llegada
    - Si los procesos que están en condición de carrera no son correctamente sincronizados, puede producirse una corrupción de datos

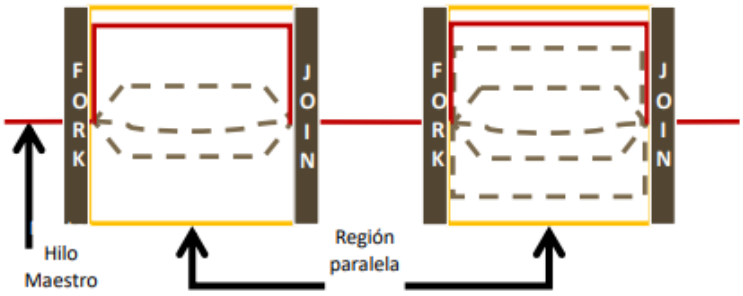
3.4. OpenMP

Es un interfaz de programación de aplicaciones (API) multiproceso portable, para computadoras paralelas que tiene una arquitectura de memoria compartida formado por:

- Conjunto de directivas del compilador (ordenes abreviadas que instruyen al compilador para insertar ordenes en el código fuente y realizar una acción en particular)
- Biblioteca de funciones
- Variables de entorno

Para paralizar el programa hay que hacerlo de forma explícita, es decir el programador debe analizar y determinar qué problemas o partes del programa se pueden ejecutar simultáneamente para que se pueda usar un grupo de subprocesos para resolver el problema.

OpenMp utiliza una arquitectura llamada fork-join, en la que se generan muchos subprocesos a partir del proceso o subproceso principal, que se utilizarán para soluciones paralelas denominadas regiones paralelas, que luego se unen solo al subproceso o proceso principal nuevamente. El programador especifica qué partes del programa requieren el número de subprocesos. Por lo tanto, se dice que OpenMP combina códigos seriales y paralelos.



3.5. Transformada de Fourier

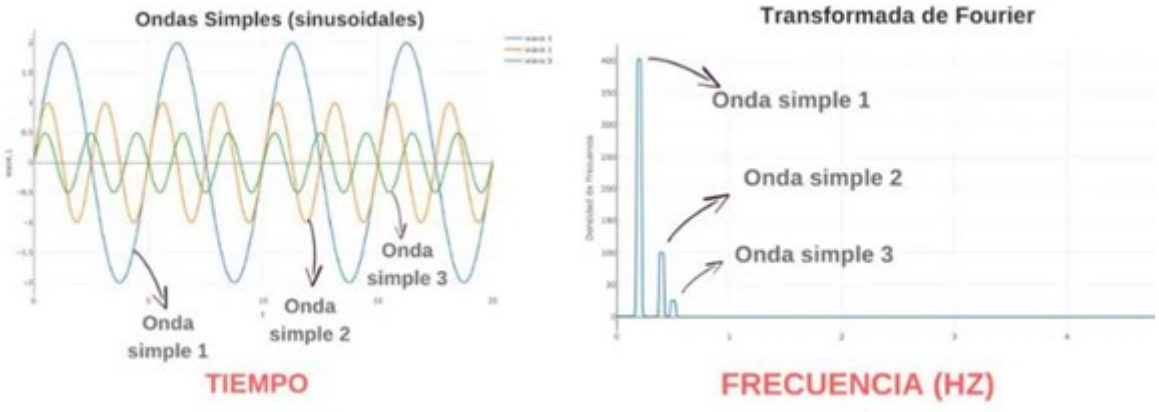
La Transformada de Fourier permite analizar las funciones no periódicas. Complementa de esta manera a la Serie de Fourier, que permite analizar sistemas donde están involucradas las funciones periódicas y que podemos representarla en términos de sus componentes sinusoidales, cada componente con una frecuencia en particular.Conforme el período se incrementa, la frecuencia fundamental disminuye y las componentes relacionadas armónicamente se hacen más cercanas a la frecuencia.

A medida que el periodo se hace infinito, las componentes de frecuencia forman un continuo y la suma de la serie de Fourier se convierte en una integral.

$$\mathfrak{F}(x_t) = \int_{-\infty}^{\infty} x_t e^{-j2\pi st} dt$$

donde **j** representa la unidad imaginaria.

Cuando hablamos de la descomposoción de señales, la transformada de Fourier ayuda a identificar los componentes simples, las ondas puras de la señal, ya que estas en conjunto forman la onda compleja.



Visualmente, podemos imaginarlo como enrrollar una gráfica alrededor de un círculo con una frecuencia **F**, registrando en esta gráfica su centro de masa.
La transformada de Fourier de una fución de intensidad versus tiempo es una función que no tiene como argumento al tiempo, sin embargo, toma una frecuencia, donde el valor que está función retorna es un número complejo en el plano complejo correspondiente a la fuerza de la frecuencia en la señal original.

3.5.1. Transformada Discreta de Fourier

Toda señal puede ser representada por la suma de series de Fourier. Con un análisis adecuado es posible obtener una representación de Fourier para señales de duración finita. Esta representación es la que se conoce como la Transformada de Fourier Discreta (TFD). La TFD se puede representar como:

$$x[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}$$

donde $W_N = e^{-j\frac{2\pi}{N}}$

Se puede observar a simple vista que su resolución directa implica N multiplicaciones complejas y N-1 adiciones complejas por cada k. Por lo tanto, el cálculo directo de una TFD es de orden $O(N^2)$

Para valores pequeños de N la resolución no conlleva mucho tiempo o gastos de recursos, pero para valores N lo suficientemente grandes el cálculo directo se torna poco eficiente, no sólo por el gran

tiempo que consume sino también por el acaparamiento de los recursos necesarios.

Se puede ver, entonces, que el orden del cálculo directo impone un límite en aquellas aplicaciones que hacen uso de la TFD, especialmente las de tiempo real, dado que para valores mayores a cierto N el cálculo podrá resultar demasiado lento y los recursos disponibles podrán ser insuficientes.
Es así que aparece la Transformada Rápida de Fourier (en inglés Fast Fourier Transform, FFT), un algoritmo para el cálculo eficiente de la TFD. Su importancia radica en el hecho que elimina una gran parte de los cálculos repetitivos a los que se ve sometida la TFD, por lo que se logra un cálculo más rápido a menor costo.

El algoritmo de la FFT fue originalmente inventado por Carl Friedrich Gauss en 1805. Diferentes versiones del algoritmo fueron descubiertas a lo largo de los años, pero la FFT no se hizo popular sino hasta 1965, con la publicación de James Cooley y John Tukey, quienes reinventaron el algoritmo al describir como ejecutarlo de forma eficiente en una computadora

3.5.2. Tranformada Rápida de Fourier

La transformada rápida de Fourier es un algoritmo para el cálculo de la transformada discreta de Fourier que reduce el tiempo de ejecución de un programa en gran medida.

Conociendo el concepto y función de la transformada de Fourier Discreta pasamos a un terreno computacional, donde el cálculo de esta puede llegar a ser tardado y costoso para grandes instancias debido a su fórmula:

$$x[k] = \sum_{n=0}^{N-1} x[n] r^{\frac{-j2\pi kn}{N}}$$

Si queremos determinar la DTF de una señal $x[n]$ donde N es el tamaño de términos a calcular, debemos multiplicar cada uno de sus valores por la constante **"e"** elevada a una función de **n**. Esto nos haría realizar **n** (multiplicaciones) multiplicado por**n** (adiciones), dándonos así una complejidad cuadrática $0n^2$. Computacionalmente hablando, con poca información no podría presentar un gran problema, sin embargo, sabemos que una complejidad cuadrática es problemática en los casos que se trabaja con grandes cantidades de información.

Aquí entra la Transformara Rápida de Fourier, un algoritmo eficiente para calcular la DFT eliminando a todos aquellos procesos repetitivos que puede presentar, logrando un cálculo más rápido a menor costo. Su idea principal se basa en la descomposición iterativa en Transformadas de Fourier Discretas más simples. Cabe mencionar que la FFT asume como datos de entrada valores equivalentes a una potencia de 2, por lo tanto, para la fórmula anterior se asume que N es potencia de 2. La FFT hace uso de dos teoremas importantes de la DFT:

$$\begin{array}{cc} \textit{Simetría Conjugada Compleja:} & W_{N)}^{k(N-n} = W_{N)}^{-kn} = (W_N{}^{kn})^* \\ \textit{Periodicidad en n,k:} & W_N^{kn} = W_{N)}^{k(N+n)} = (W_N{}^{(k+N)n}) \end{array}$$

Sin entrar en muchos detalles, gracias estas propiedades se es posible calcular la Transformada de Fourier Discreta por medio de su división, obteniendo la siguiente fórmula:

$$X[k] = \sum_{r=0}^{N/2-1} x[2r](W_{N/2)})^{rk} + W_N^K \sum_{r=0}^{N/2-1} x[2r+1](W_{N/2)})^{rk}$$

Con esta fórmula obtenemos la DFT a partir de pares de DFT con $\frac{n}{4}$ muestras hasta obtener transformadas de muestras singulares, cuyo cálculo es sencillo. Si inicialmente se tenían N muestras, ahora se podrán realizar $log_2 N$ divisiones, así con el tamaño de entrada se tiene una complejidad de $O(Nlog_2 N)$, complejidad logarítmica mucho mejor que la exponencial cuando se habla de grandes cantidades de información. Con estos datos, haciendo una pequeña comparación entre ambas versiones, podemos obtener la siguiente tabla:

N	N° de operaciones usando cálculo directo (N^2)	N° de operaciones usando FFT ($N.log_2N$)
4	8	4
8	64	12
16	256	32
32	1.024	80
64	4.096	192
28	16.384	448
256	65.536	1.024
512	262.144	2.304
1.024	1.048.576	5.120
2^{30}	2^{60}	30 x 2^{30}

Donde notamos que para pocos elementos la diferencia no es mucha, sin embargo la diferencia se ve reflejada en cantidades de información más grandes. Que, mirando de cerca, para una entrada equivalente a 2^{30}, utilizando la DFT tardaría aproximadamente 13,343 días en calcular, sin embargo, con la FFT tomaría únicamente 32 segundos aproximadamente.

Ahora, si la FFT es un algoritmo que optimiza el proceso de cálculo de la Transformada de Fourier Discreta, ¿Podemos optimizar aún más este proceso a través de la paralelización obteniendo menores tiempos? El siguiente análisis de este algoritmo nos dirá si su versión paralela es eficiente y recomendable a comparación de su versión secuencial.

4. Descripción del algoritmo

4.1. Algoritmo Cooley-Tukey FFT

El algoritmo más común para calcular la FFT es el algoritmo Cooley-Tukey. Este algoritmo reexpresa la transformda discreta de Fourier (DFT) como un arreglo de tamaño $N = N_1 N_2$, en términos de N_1 DFTs más pequeñas de tamaño N_2, recursivamente. Esto reduce la complejidad del algoritmo a $O(N \log N)$ para un arreglo de tamaño N. Algunas características del algoritmo, grosso modo, son las siguientes:

- **Entrada**: Un arreglo $X[N]$ de números complejos, donde $X[i]$, $i = 0, 1, 2, ..., N-1$ puede ser interpretado como el coeficiente de x^i de un polinomio de grado $N-1$.
- **Salida**: Otro arreglo $Y[N]$ de números complejos, resultado de aplicar la transformada rápida de Fourier a $X[N]$.
- El objetivo del algoritmo es evaluar el polinomio representado por $X[N]$, en las raíces N-ésimas de la unidad.
- Se utiliza recursividad, cuyo caso base es $N = 1$.
- Existen implementaciones donde N es una potencia de algún número mayor que 2, pero tienden a ser mucho más complicadas.

Considerando la entrada del algoritmo $X[N]$ como la representación de coeficientes de un polinomio de grado $N-1$, el algoritmo de la FFT se describe de la siguiente manera:

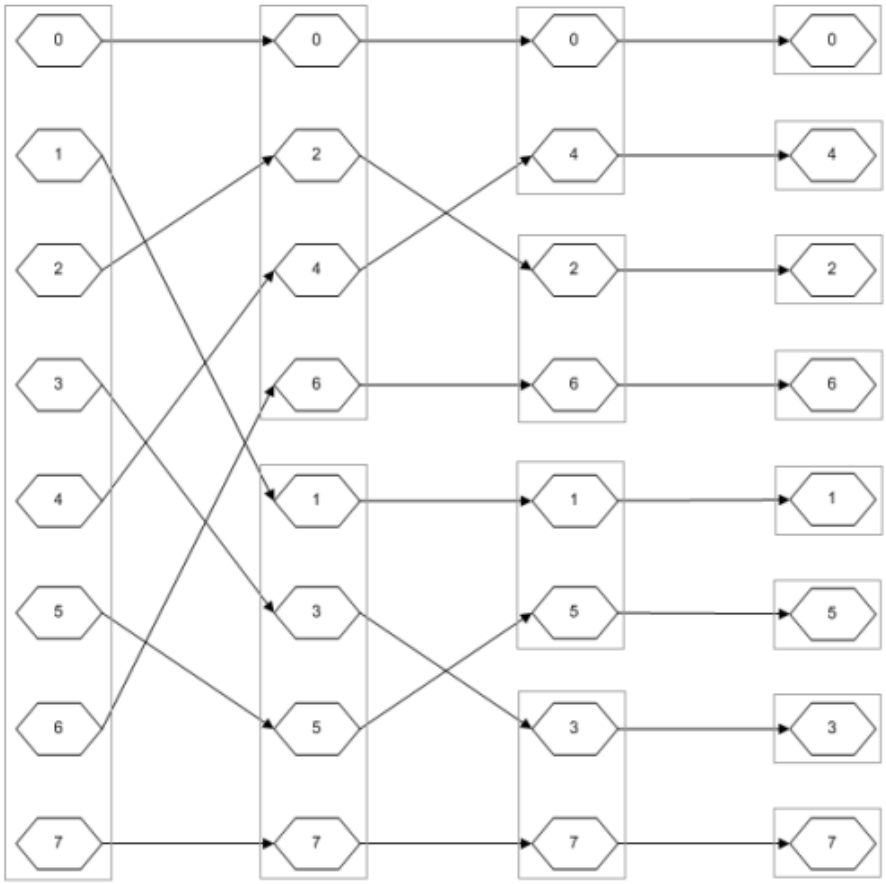
1. N es igual al tamaño del arreglo. Asumimos que N es una potencia de 2.
2. Se establece el caso base de la recursividad cuando $N = 1$. Si se entra en este caso sucede lo siguiente:
   - Este caso es equivalente a evaluar un polinomio de grado 0 (valor constante). Simplemente se regresa el arreglo con este valor.
3. Se define $\omega = e^{\frac{2\pi i}{N}}$.
4. Para la primera parte de los llamados recursivos, se dividen los índices pares de $X[N]$ tal que $X_{pares} = [x_0, x_2, x_4, ..., x_{N-2}]$.
5. De la misma manera se dividen los índices impares de $X[N]$ tal que $X_{impares} = [x_1, x_3, x_5, ..., x_{N-1}]$.
6. Se hacen dos llamadas recursivas a la FFT: una con X_{pares}, que devolverá el arreglo Y_{pares}; y otra con $X_{impares}$, que se asigna al arreglo $Y_{impares}$. Ahora los polinomios que llaman a la función son de grado $N/2$.
7. Se inicializa el arreglo $Y[N]$.
8. Para j desde 0 hasta $N/2$:
   - $Y[j] = Y_{pares}[j] + \omega^j Y_{impares}[j]$
   - $Y[j + n/2] = Y_{pares}[j] - \omega^j Y_{impares}[j]$
9. Regresar la lista Y

4.1.1. Pseudocódigo Cooley-Tukey

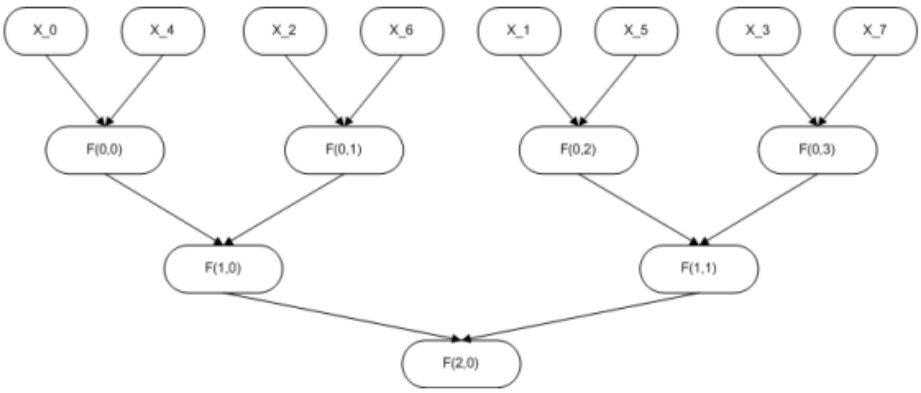
```
def FFT(P):
    # P - [p_0, p_1, ..., p_{n-1}] coeff representation
    n = len(P) # n is a power of 2
    if n == 1:
        return P
    ω = e^{2πi/n}
    P_e, P_o = [p_0, p_2, ..., p_{n-2}], [p_1, p_3, ..., p_{n-1}]
    y_e, y_o = FFT(P_e), FFT(P_o)
    y = [0] * n
    for j in range(n/2):
        y[j] = y_e[j] + ω^j y_o[j]
        y[j + n/2] = y_e[j] - ω^j y_o[j]
    return y
```

4.2. Versión iterativa de la FFT

En lugar de un algoritmo recursivo top-down, la FFT puede ser calculada de manera iterativa bottom-up. Sin embargo, la implementación iterativa no es tan sencilla (ni elegante) como la versión recursiva. En la implementación recursiva, los datos son divididos en dos arreglos: el primero contiene los índices pares y el segundo los índices impares, respectivamente. Si este procedimiento se vuelce a aplicar, se obtienen 4 subarreglos, y así sucesivamente. La siguiente figura ilustra el proceso de división de un arreglo de 8 elementos, efectuada 3 veces:



Esto sugiere que podemos reordenar los datos de manera que la DFT se calcule como en la siguiente imagen, donde los elementos se combinan de la misma manera que en la imagen anterior. $F(2,0)$ es la DFT de $x_0, ..., x_7$:



De esta manera, se pueden definir los siguientes pasos para la versión iterativa:

1. Reordenar los índices del arreglo de entrada.
2. Calcular la DFT tomando los elementos en pares. Así, se tendría que el arreglo contiene $N/2$ DFTs de pares de elementos.
3. Tomar las $N/2$ DFTs en pares y calcular nuevamente su DFT. Así, se tendrían DFTs de arreglos de 4 elementos, cada una reemplazando una DFT de 2 elementos. Es decir, se tienen ahora $N/4$ DFTs de subarreglos de 4 elementos.
4. Se continua hasta que el vector tenga 2 DFTs de $N/2$ elementos, que se combinan para tener la DFT final de N elementos.

4.2.1. Pseudocódigo de la versión iterativa de la FFT

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n ← length[a]            ▷ n is a power of 2.
 3  for s ← 1 to lg n
 4       do m ← 2^s
 5          ω_m ← e^(2πi/m)
 6          ω ← 1
 7          for j ← 0 to m/2 − 1
 8               do for k ← j to n − 1 by m
 9                    do t ← ω A[k + m/2]
10                       u ← A[k]
11                       A[k] ← u + t
12                       A[k + m/2] ← u − t
13                  ω ← ω ω_m
14  return A
```

Nota: Obsérvese que *bit-reverse-copy* es una función que invierte los bits de la representación binaria de un número. Esto es útil para hacer el reordenamiento de los elementos del arreglo, donde cada número se empareja con su número con bits invertidos.

4.3. Estructura paralela seleccionada

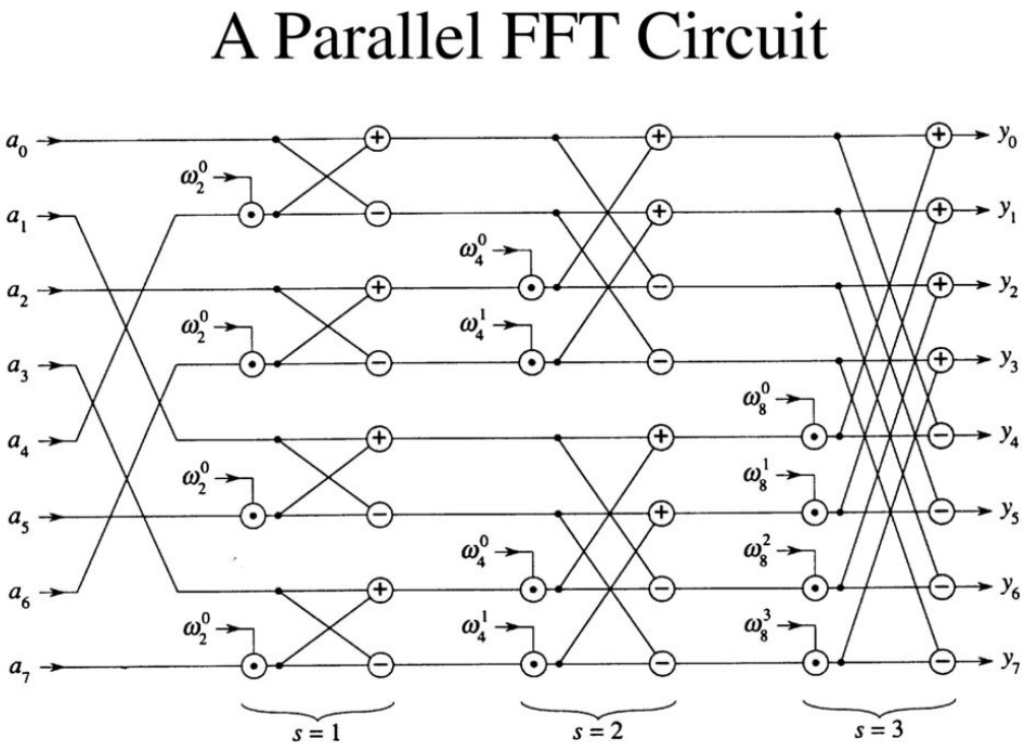
La implementación seleccionada para este algoritmo se ejecuta en varios procesadores, usando el lenguaje C y directivas del compilador (OpenMP) para acceder a las variables de memoria compartida. Más adelante se explicará por qué se implementa el paralelismo de datos, pero por ahora, podemos decir que el programa tiene una estructura paralela.

5. Paralelización del algoritmo

Para paralelizar el algoritmo de la FFT, tenemos que considerar qué versión del algoritmo es más ideal. Existen dos razones principales para elegir la versión iterativa sobre la recursiva: primero, la versión iterativa puede funcionar realizando menos operaciones para calcular los índices; segundo, es más fácil de paralelizar la versión iterativa si se aprovechan las iteraciones de los bucles.

El algoritmo paralelo se desarrolla en dos etapas. Primero, se reordenan los elementos del arreglo de entrada (puede usarse inversión de bits). En la segunda etapa, se llevan a cabo $\log N$ fases, donde cada fase tiene $N/2$ mariposas (operaciones de multiplicación, suma y resta). Las operacines mariposa de cada fase son paralelizadas.

Dadas las condiciones del algoritmo, es observable que se puede paralelizar dividiendo las iteraciones de los ciclos for. En dichos ciclos es importante cuidar los recursos de uso compartido y las variables privadas. La siguiente imagen muestra el proceso para el algoritmo paralelo:



5.1. Tipo de paralelismo

Usando la versión iterativa del algoritmo, podemos identificar que se cumple con las siguientes características:

1. El dominio del problema (un arreglo de números complejos) es subdividido y cada procesador se vuelve "propietario"de una porción del arreglo. Así, cada propietario realiza las instrucciones para sus propios elementos.
2. Para subdividir el arreglo, se reparten las iteraciones del ciclo que lo recorre.
3. Algunas variables de las iteraciones son compartidas y otras son privadas. Es decir, se hace uso combinado de memoria compartida y memoria distribuida.
4. Después de cada fase, cada par de hilos se combina (join) para la siguiente fase hasta obtener el arreglo de salida.

Este tipo de paralelismo es conocido como paralelismo de datos o descomposición del dominio, lo que significa que varios hilos pueden trabajar con el mismo algoritmo o instrucciones que se repetirán, pero sobre diferentes datos y no hay dependencias con las demás iteraciones de cada fase.

5.2. Métricas de desempeño

El fin de paralelizar un algoritmo es buscar una mejora a comparación de su versión secuencial, por ejemplo, que el tiempo de ejecución se vea disminuido ya que si es así significaría que el desempeño de nuestro algoritmo es eficiente a comparación de su versión secuencial. Buscando mirar este desempeño, existen métricas que nos permiten observar si es realmente conveniente paralelizar nuestro algoritmo o dejarlo en su versión serial, métricas que se verán a continuación.

Tenemos cuatro métricas, las cuales son el tiempo de procesamiento y ejecución, speedup, eficiencia y fracción serial. De estas se mirarán a profundidad las tres últimas, dejando el tiempo de procesamiento y ejecución al apartado de las pruebas realizadas para realizar un mejor análisis.

5.2.1. SpeedUp

Brevemente, el SpeedUp nos ofrece la relación que existe entre el tiempo de ejecución de un programa sobre un procesador sobre el tiempo que toma en ejecutarse sobre n procesadores o hilos. Con la formula a continuación:

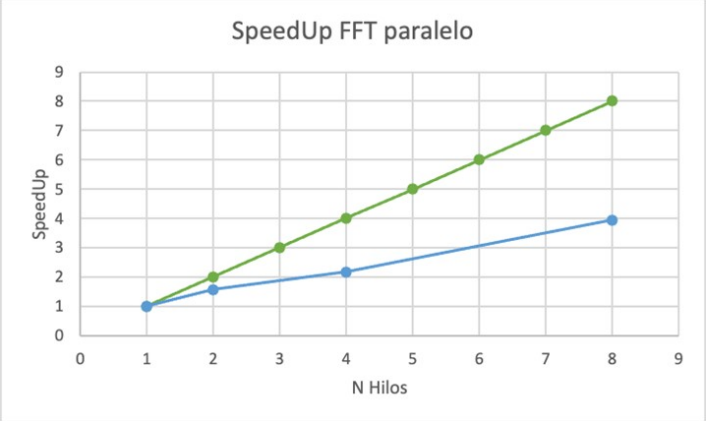
$$S(n) = \frac{T(1)}{T(n)}$$

Al calcular esta métrica se pretende el caso ideal en que sea lineal, ya que de esta manera si se trabajan sobre n procesadores, se debería tener una mejora de factor n. Sin embargo, la realidad es que es habitual tener un SpeedUp sublineal (o menor al ideal), ya que no siempre se garantiza que al utilizar n procesadores se obtendrá una mejora de factor n, esto no es malo ya que la mayoría de los algoritmos paralelizables pertenecen a esta categoría. Cabe mencionar que puede haber casos donde el SpeedUp del algoritmo pueda superar al ideal.

Para el momento en que se realiza este reporte ya se han realizado las pruebas sobre el tiempo de ejecución para una entrada de 2^{20} elementos sobre un equipo con un procesador de 4 núcleos y 8 hilos, obteniendo los siguientes valores de SpeedUp:

N hilos	SpeedUp
1	1
2	1.568
4	2.172
8	3.934

Con estos resultados podemos obtener la siguiente gráfica:



Tras mirar la gráfica nos podemos dar cuenta que nuestro algoritmo para calcular la FFT en su versión paralela cuenta con un SpeedUp sublineal, lo cual es un factor favorable debido a que nos indica que hay una mejora en los tiempos de ejecución cuando se trabaja con más hilos a comparación de cuando se trabaja con solo uno.

5.2.2. Eficiencia

Esta métrica hace referencia a la proporción de recursos computacionales utilizados de manera simultánea, corresponde a un valor normalizado del SpeedUp (entre 0 y 1) y se asocia a la idea de que n procesadores deben realizar una fracción $\frac{1}{n}$ del tiempo que le lleva a un solo procesador. Mientras el valor se acerque más a uno nos indicará una situación ideal en el rendimiento y aprovechamiento.

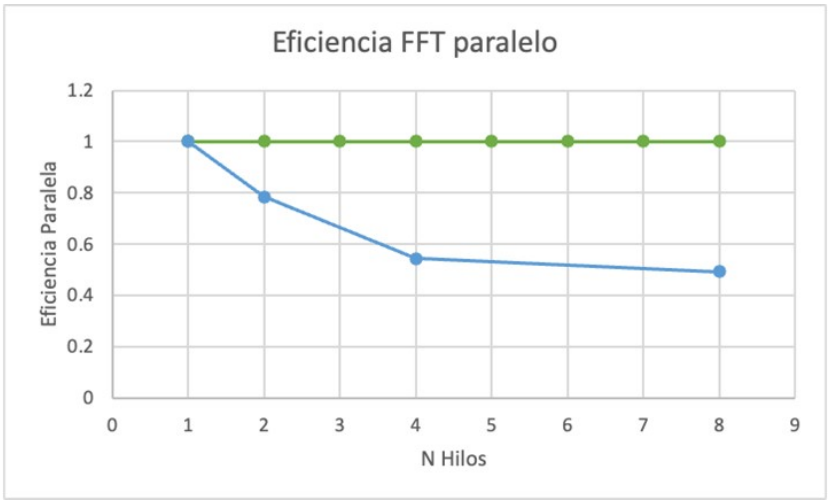
Para obtener el valor de la eficiencia según el número de procesadores se tiene la siguiente fórmula:

$$E(n) = \frac{T(1)}{n \cdot T(n)} = \frac{S(n)}{n}$$

Que relaciona al SpeedUp entre el número de procesadores. Con los valores obtenidos anteriormente, se llegó a los valores contenidos en la siguiente tabla:

N hilos	Eficiencia
1	1
2	0.784
4	0.543
8	0.491

Comparando con el caso ideal, obtendríamos la siguiente gráfica:



Podemos notar que conforme aumenta el número de hilos aumentará el reparto y uso de recursos entre cada uno de ellos de manera simultánea, haciendo que el valor de la eficiencia disminuya. Esto no indica una situación mala para nuestro algoritmo, ya que al final al paralelizar un algoritmo se dividirán los recursos entre el número de procesadores, núcleos o hilos con los que se trabaje, buscando aproximar lo mayor posible el valor de la eficiencia a uno.

5.2.3. Fracción serial

Esta métrica depende de la cantidad de N operaciones, si N aumenta la fracción serial disminuye. Esta relaciona al SpeedUp y a la eficiencia con el fin de tomar en cuenta a otros factores, como el tiempo serial o en el cálculo de la fracción paralela que contribuye al escalamiento de un código paralelo. Su calculo se obtiene a partir de la siguiente fórmula:

$$f = \frac{\frac{1}{S} - \frac{1}{n}}{1 - \frac{1}{n}}$$

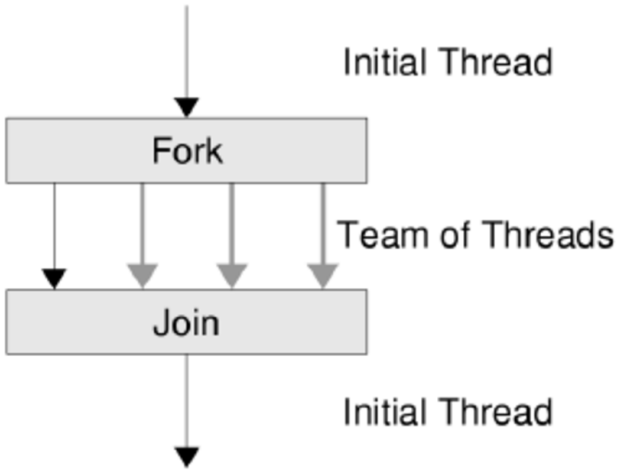
A partir de ella se obtuvieron los siguientes resultados para cada número de hilos.

N hilos	Eficiencia
1	0
2	0.2755
4	0.2805
8	0..1476

5.3. Formas de comunicación

Sabemos que OpenMP implementa la metodología Fork-Join. Este patrón de diseño permite dividir el programa en ramas paralelas para volver a unirlas en algún punto subsecuente del programa y continuar con la ejecución serial. Algunas secciones paralelas pueden tener de igual manera ramas anidadas, de manera que se puedan alcanzar otros tipos de granularidad.

El reordenamiento del arreglo de entrada de la primera etapa permite acomodar los elementos de manera que en la segunda etapa cada fase sea libre de comunicación (al hacer las mariposas), y solo sería requerida al reunir los hilos.



El programa utiliza algunas variables en memoria compartida y otras son de uso privado (directiva private). Es importante definir cuales de estas variables son de uso compartido y cuales no:

- Variables de uso compartido:
  1. Cantidad de elementos del arreglo.
  2. Arreglo de números complejos de entrada.
  3. Arrreglo de números complejos de salida.
- Variables privadas:
  1. Variable de control del ciclo for (por defecto, OpenMP suele establecer esta variable como privada).

2. $\omega_m^0, \omega_m^1, ..., \omega_m^{m/2-1}$, donde $m = 2^s$ y s es una fase paralela.

Los procesos comparten un espacio de memoria común y escriben y leen de manera asíncrona durante las fases de la segunda etapa (mariposas); por lo tanto, no es necesario especificar como se comunican los datos entre las tareas, puesto que cada hilo trabaja en su partición (y cada partición es independiente) y al final se unen de nuevo por pares (por esta razón, el número de hilos debe ser una potencia de 2) hasta unirse todos en el hilo principal del programa.

5.4. Granularidad

Para la segunda etapa de este programa, se tienen $\log N$ fases. Para cada fase s, desde 1 hasta $\log N$, hay $N/2^s$ conjuntos de mariposas con 2^{s-1} mariposas cada grupo. Para obtener una granularidad gruesa, cada hilo procesa subconjuntos de N/P elementos, asumiendo que P es una potencia de 2. En estas subrutinas, los procesadores hacen uso de referencias en memoria a recursos compartidos (los arreglos X y Y); sin embargo, para modificar los valores a lo largo del arreglo, cada hilo tiene asignada una porción de iteraciones contiguas que va a realizar.

Es decir, cada hilo tiene también asignada la porción de memoria compartida que modifica, y es encargado de calcular y escribir N/P elementos. La comunicación sucede cuando se combinan los resultados de los pares de hilos para calcular la siguiente fase hasta llegar al arreglo final.

5.5. Balance de carga

En este programa, la directiva for de OpenMP indica al compilador distribuir las iteraciones de los ciclos entre los hilos que se manejen en cada region paralela. Por lo tanto, se trata de un tipo de balance carga estático, puesto que se asignan las instrucciones en tiempo de compilación.

6. Implementación del algoritmo paralelo

Antes de comenzar con el análisis de la implementación del código se reconoce el gran trabajo por parte de Wesley Petersen y John Burkardt, encargados de implementar la Transformada Rápida de Fourier en el lenguaje de programación C y su derivada interfaz OpenMP para programación de multiprocesamiento. Además, se proporciona el enlace con el cual se accedió al programa, el cual es el siguiente: $https://people.sc.fsu.edu/\ jburkardt/c\_src/fft\_serial/fft\_serial.html$.

Comenzando con la implementación, el programa se encarga de calcular la transformada de Fourier Discreta de un vector de datos complejos de manera más eficiente computacionalmente hablando, como se mencionó anteriormente, reduciendo cálculos repetitivos. Dicho en otras palabras, generará la transformada de un arreglo de n elementos. Sin entrar en mucho detalle ya que un análisis a profundidad del código podría derivar en otro trabajo de investigación completo, se resumirá en las partes importantes del código. El programa se encarga de generar arreglos de un tamaño n igual a un valor equivalente a una potencia de dos, ya que como se mencionó en los antecedentes, la FFT espera entradas de este tipo.

Posteriormente de acuerdo al tamaño del arreglo generará números aleatorios entre 0 y 1 a los cuales se les generará un arreglo de senos y cosenos con los cuales se podrá aplicar la transformada. Teniendo estos elementos, el programa realizará la transformación al vector de datos aplicando los conceptos anteriormente donde la transformada se va dividiendo en la suma de otras transformadas más sencillas de resolver y así obtener la transformada final. Cabe mencionar que a lo largo de este proceso el programa va registrando el tiempo que tomo calcular por cada tamaño de arreglo.

Del proceso anterior, en las funciones principales se colocan las directivas de OpenMP para poder hacer uso de sus funciones multiproceso de forma cautelosa para . De estas destacan las directivas para decidir que variables son compartidas y que variables serán privadas. Además de la directiva for que divide el número de iteraciones entre los hilos que se tengan; así como la directiva nowait que permite a los hilos terminar su bloque de instrucciones sin tener que esperar a otros.

Lo anterior podemos verlo en la parte del código para asignar los valores aleatorios en el arreglo:

```
135  # pragma omp parallel \
136      shared ( n, x, z ) \
137      private ( i, z0, z1 )
138
139  # pragma omp for nowait
140
141          for ( i = 0; i < 2 * n; i = i + 2 )
142          {
143            z0 = 0.0;
144            z1 = 0.0;
145            x[i] = z0;
146            z[i] = z0;
147            x[i+1] = z1;
148            z[i+1] = z1;
149          }
150      }
```

En la parte para generar las tablas de senos y cosenos necesarias para el cálculo de la transformada:

```
06   # pragma omp parallel \
07       shared ( aw, n, w ) \
08       private ( arg, i )
09
10   # pragma omp for nowait
11
12     for ( i = 0; i < n2; i++ )
13     {
14       arg = aw * ( ( double ) i );
15       w[i*2+0] = cos ( arg );
16       w[i*2+1] = sin ( arg );
17     }
18     return;
19   }
```

Y por último en el proceso de la partición de la Transformada en varias transformadas de Fourier:

```
513  # pragma omp parallel \
514      shared ( a, b, c, d, lj, mj, mj2, sgn, w ) \
515      private ( ambr, ambu, j, ja, jb, jc, jd, jw, k, wjw )
516
517  # pragma omp for nowait
518
519    for ( j = 0; j < lj; j++ )
520    {
521      jw = j * mj;
522      ja  = jw;
523      jb  = ja;
524      jc  = j * mj2;
525      jd  = jc;
526
527      wjw[0] = w[jw*2+0];
528      wjw[1] = w[jw*2+1];
529
530      if ( sgn < 0.0 )
531      {
532        wjw[1] = - wjw[1];
533      }
534
535      for ( k = 0; k < mj; k++ )
536      {
537        c[(jc+k)*2+0] = a[(ja+k)*2+0] + b[(jb+k)*2+0];
538        c[(jc+k)*2+1] = a[(ja+k)*2+1] + b[(jb+k)*2+1];
```

El programa realiza el cálculo de la transformada de Fourier de arreglos de tamaños equivalentes hasta 2^{20}. Tras ejecutarlo el programa nos ofrece una salida en consola mostrando el tamaño del arreglo al que se le aplicó la transformada, el tiempo que tomó en calcular dicho arreglo y finalmente la cantidad de operaciones realizadas en la unidad de medida megaflops. La forma en que se logró esto fue por medio de funciones que contiene la librería $< omp.h >$ para obtener el tiempo de ejecución; en específico la función $omp\_get\_wtime()$

La salida es la siguiente:

N	Time	MFLOPS
2	2.771100e-03	7.217351
4	1.315670e-02	6.080552
8	2.118590e-02	11.328289
16	3.028660e-02	21.131457
32	2.028300e-03	78.883794
64	2.349800e-03	163.418163
128	9.551500e-03	93.807255
256	5.509800e-03	371.701332
512	5.494000e-04	838.733164
1024	1.117700e-03	916.167131
2048	2.836100e-03	794.330242
4096	1.111600e-02	442.173444
8192	7.438000e-04	1431.782732
16384	1.205200e-03	1903.219391
32768	2.095800e-03	2345.261950
65536	1.851080e-02	566.467144
131072	1.377010e-02	1618.161087
262144	2.496560e-02	1890.037492
524288	4.651940e-02	2141.358659
1048576	1.105911e-01	1896.311729

Notamos que se muestra el tiempo en segundos que tomó aplicar la transformada rápida de Fourier en cada arreglo de tamaño N, así como los megaflops que requirió. Como podemos notar, todos los tamaños de los arreglos son valores equivalentes a potencias de dos.

7. Pruebas realizadas

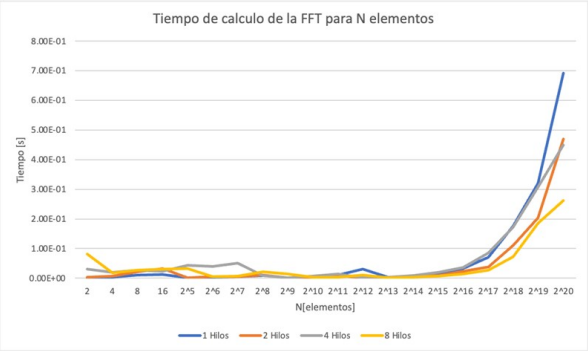
En este apartado se mostrará el resultado de diversas pruebas realizadas con el algoritmo en su forma secuencial, como en su forma paralela para diferentes tamaños de entrada, así como en diferentes equipos con diferentes procesadores. Además, se mirará la métrica de desempeño restante con la cual fue posible el cálculo de las demás: el tiempo de procesamiento y ejecución. Las pruebas que se realizaron tuvieron el fin de ver cómo se modificaba el balance de carga entre hilo, la comparación del tiempo de ejecución entre versión serial y paralela y por último la comparación entre el número de operaciones entre ambas versiones.

Las pruebas se dividieron en dos casos:

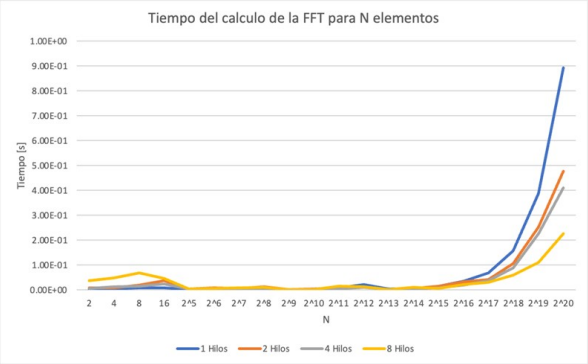
Para el caso primero se realizaron pruebas con un procesador AMD Ryzen 5 de núcleos y 8 hilos. Para el segundo caso se realizaron las pruebas con un procesador Intel i7-8650u con 4 núcleos y 8 hilos.

La primera prueba consistió en ver cómo se ve afectado el tiempo de cálculo de la transformada para cada arreglo dependiendo la repartición de tareas entre el número de hilos. Mirando como se ve afectado el balance de carga.
Tras recolectar la información se obtuvo las siguientes gráficas:

- Caso 1



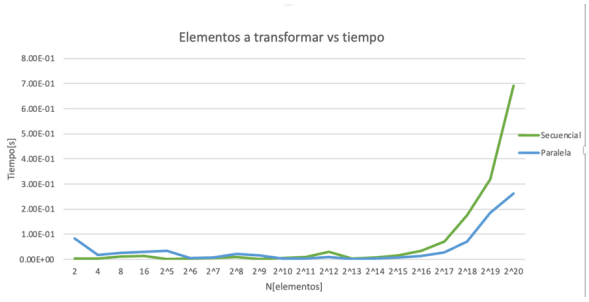
- Caso 2



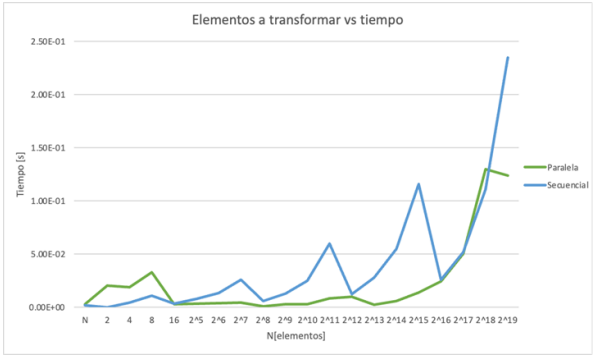
Podemos observar que conforme se utilizan más hilos, las tareas se reparten entre ellos, disminuyendo el tiempo de ejecución.

La segunda prueba consiste en mirar el tiempo de ejecución de acuerdo al tamaño del arreglo al que se le aplicará la transformada. Tras obtener datos se obtuvo la siguiente gráfica, donde se compara la versión secuencial y paralela del algoritmo.

- **Caso 1**



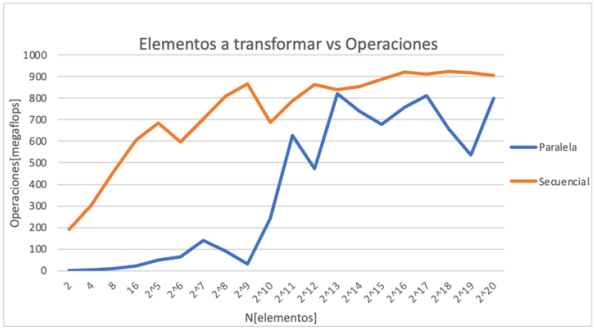
- **Caso 2**



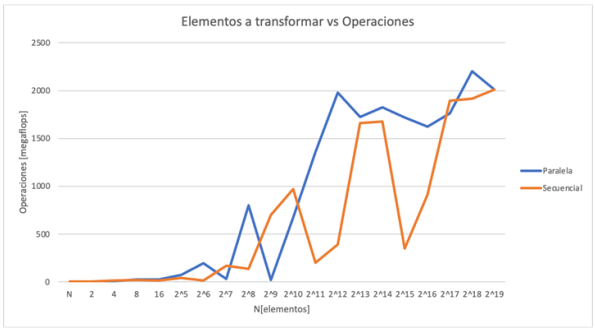
Analizando la gráfica podemos notar que como se mencionaba antes, en un inicio para cantidades no tan grandes de datos las versiones del algoritmo no presentan mucha diferencia, sin embargo, es en la cantidad grande de información donde se ve reflejada esta diferencia en el tiempo de ejecución. La versión paralela es eficiente para grandes cantidades de datos.

Por último, tenemos la prueba donde se busca mirar el número de operaciones realizadas de acuerdo a la cantidad de elementos con los que se trabaja. Para el número de operaciones se utilizaron los megaflops, una unidad de rendimiento computacional que representa la cantidad de operaciones flotantes realizadas por segundos.

- **Caso 1**



- **Caso 2**



Para el primer caso podemos notar un buen aprovechamiento de los recursos debido a la cantidad de operaciones realizadas comparando ambas versiones; la versión secuencial requirió muchas más operaciones a comparación de su versión paralela. Por otro lado, tenemos una idea diferente en el segundo caso, ya que podemos notar que la versión paralela llega a realizar muchas más operaciones que la versión paralela. Esto puede deberse al balance de carga entre hilos. Estas pruebas nos permiten concluir que en efecto el balance de carga entre hilos va modificando de acuerdo a los que utilicemos,

permitiendo una disminución en el tiempo de ejecución. Además, que la diferencia del tiempo de ejecución entre ambas versiones se ve mayormente reflejado cuando se trabaja con grandes cantidades de información; factor favorable debido a las múltiples implicaciones que tiene la Transformada de Fourier Discreta en sus diversas aplicaciones.

8. Aplicaciones del algoritmo

Como se analizó previamente, el objetivo del algoritmo de la transformada rápida de Fourier es evaluar el polinomio representado por $X[N]$ en las raíces N-ésimas de la unidad, demostrando la implementación de un vector de datos complejo, usando OpenMP para ejecución paralela.

La digitalización de una señal, como el sonido, en donde se requieren una gran cantidad de muestreos como puede ser N>1000 puede convertirse en un proceso tedioso por lo gran cantidad de operaciones que se deben realizar, haciendo que haya más demora en el procesamiento. Cada punto en la imagen transformada requiere de N^2multiplicaciones complejas y N(N-1) sumas, por lo que la implementación de este algoritmo de la FFT es ideal para este problema.

Actualmente, en todo proceso productivo se busca ser eficiente respeto a costes mínimos y aumento de volumen de la produccion para una mejor competitividad con productos de calidad. El mantenimiento de las maquinarias es sin duda para cumplir estaos objetivos, entre ellos, las vibraciones mecánicas es una de las herramientas más importantes que tiene el mantenimiento para conocer el estado en el que se encuentran los equipos.

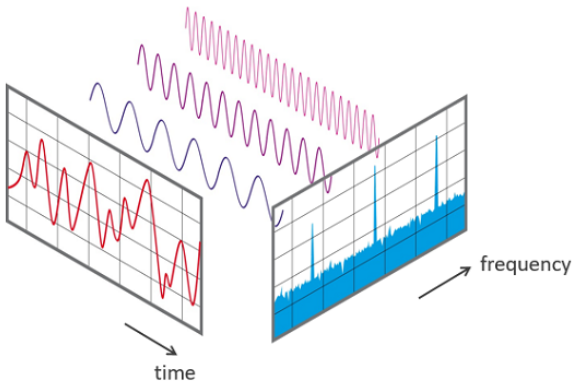
Medir las vibraciones de la maquinaria ayuda a monitorear las condiciones del equipo. La representación frecuencial captura las características espectrales de una vibración. Además de la frecuencia fundamental existen muchas frecuencias presentes en una forma de onda. Las componentes de frecuencias armónicas son enteros simples de la frecuencia fundamental, aunque no en todos los casos hay una fundamental clara.

Nuevamente tenemos un problema en donde nos ocupa realizar estos muestreos frecuenciales sin que se vuelva una tarea aburrida que conlleve mucho tiempo, y es por esto que, nuevamente, el algoritmo de la tranformada rapida de Fourier es excelente para resolver este problema.

La importancia del algoritmo radica en el hecho de que elimina una gran parte delos cálculos repetitivos a los que se ve sometida la TFD, por lo que se logra un cálculo más rápido a menor costo.

Otros ejemplos de la aplicación del algoritmo de la FFT son el diseño de circuitos, espectroscopia, cristalografía, procesamiento de señales y comunicaciones, imágenes, etc.

Es extremadamente útil al reducir el número de cálculos a partir de algo del orden n^2 a $n\ log\ n$ lo que obviamente ofrece una enorme reducción en el tiempo de cálculo.



9. Conclusiones

En conjunto con todos los conceptos vistos hasta este momento en el presente documento, recalcamos la importancia de saber identificar cuando un algoritmo en su versión secuencial puede ser paralelizado, ya que no todos pueden serlo y verificar si este proceso es de ayuda o no respecto al tiempo de ejecución del algoritmo en su versión secuencial.

En nuestro caso, el algoritmo implementando nos permitió conocer un nuevo objeto de estudio: Las transformadas de Fourier. Conocer y comprender la teoría detrás del algoritmo nos fue de mucha relevancia, ya que pudimos ser capaces de relacionar conceptos y metodologías con temas vistos de las materias pertenecientes a la División de Ciencias básicas para una mejor comprensión de este tema, además, conocer las aplicaciones que estas transformaciones tienen y cómo es que hasta nuestros días siguen siendo de mucha utilidad y más aplicadas en nuestro campo, nos lleva a diversas ideas de como podríamos hacer uso del paralelismo con estas transformaciones para lograr mayor eficiencia en las aplicaciones y programas que contienen este algoritmo para poder resolver las problemáticas computacionales que se requieran.

Encontramos que existen dos maneras principales de implementar el algoritmo FFT: la versión recursiva y la versión iterativa. Dada la naturaleza de la versión iterativa, se determinó que la paralelización se podría lograr más fácil y eficientemente en esta versión si se aprovechaban las iteraciones para la paralelización. Se observó que reordenando los elementos de la entrada del algoritmo, se pueden hacer las operaciones mariposa de cada fase de forma paralela, combinando los resultados de los hilos en pares hasta llegar al arreglo final.

Se analizó y se determinaron los siguientes aspectos para la versión paralela:

- La implementación fue realizada en lenguaje C con directivas de OpenMP para el compilador.
- El tipo de paralelismo que se aplica es paralelismo de datos.
- Si se reordenan los elementos del arreglo de entrada, se pueden hacer las operaciones mariposa de cada fase de manera paralela e independiente; sin embargo, debe existir la comunicación al final de cada fase y reunir los pares de hilos.
- Las métricas de desempeño nos indican que efectivamente existe una mejoría notable en la versión paralela del algoritmo a comparación de su versión secuencial.
- Se usa memoria compartida, pero tambien se utilizan directivas para crear variables privadas.
- La granularidad puede ser gruesa si cada procesador toma N/P elementos para hacer las operaciones de mariposa, y al final se fusionan por pares después de cada fase para obtener el arreglo final de salida.
- El balance de carga es estático porque se distribuyen las operaciones en tiempo de compilación.
- Tras realizar pruebas con ambas versiones del algoritmo, se confirma que su versión paralela presenta gran mejoría en los tiempos de ejecución para grandes cantidades de información.

Vistas algunas de las aplicaciones del algoritmo FFT, podemos decir que es de suma importancia en nuestros días haciendo que en nuestro campo, la computación, el tiempo se vea reducido en gran medida dada la correcta implementación del algoritmo.

En conclusión podemos decir que la hipótesis formulada en un inicio se confirma, dado que los tiempos de ejecución para la FFT en su versión paralela se vieron disminuidos a comparación de su versión secuencial, convirtiendo a este algoritmo paralelo en algún tipo de optimización de la optimización. Por lo tanto podemos concluir que la versión paralela de la FFT es recomendable y eficiente para casos en los que se trabaje con grandes cantidades de información.

10. Autoevaluación general del equipo

Siendo este el segundo proyecto realizado por el equipo 8; conformado por Barrios Aguilar Dulce Michelle, Chong Herández Samuel y Mendoza Hernández Carlos Emiliano, consideramos que se presentó un desempeño regular para la forma de trabajo del presente proyecto a comparación del primer proyecto. Intuimos que esto se debió a que se presentaron algunas dificultades en la repartición del trabajo y en la forma de organización de este, así como los tiempos de trabajo para la creación del mismo.

Un factor que nos provocó retrasos en la elaboración de este proyecto fue el tiempo invertido por cada uno de nosotros para investigar, leer y comprender la teoría matemática necesaria, la cual pensamos que fue difícil de comprender y requirió de un buen tiempo de lectura y análisis, además de diversos materiales y recursos visuales. Por otra parte, consideramos que fue un reto condensar un concepto matemático tan poderoso y abstracto (además de avanzado) en una presentación breve que cubra todos los aspectos relevantes.

A pesar de esto, la forma en que se trabajó sobre presión resultó satisfactoria, cumpliendo con los requisitos requeridos para cumplir con el trabajo de investigación.

Como equipo tomamos esto como un aprendizaje que nos permite tener experiencia para futuros trabajos, teniendo como punto de partida la organización de prioridades no solo escolares sino en otros ámbitos para poder cumplir con cada una de ellas.

11. Referencias

1. Complejidad algorítmica. Departamento de Informática Universidad de Valladolid Campus de Segovia. Obtenido desde https://www2.infor.uva.es/ jvalvarez/docencia/tema5.pdf

2. Martínez A, N. (2017). Transformada rápida de Fourier Implementación y algunas aplicaciones[Tesis de fin de grado]. UNIVERSIDAD DE MURCIA. Facultad De Matemáticas.

3. Supervielle F. (Septiembre 2011). Digitalización del sonido. Universidad Nacional del Sur. Obtenido desde http://lcr.uns.edu.ar/fvc/NotasDeAplicacion/FVC-Gaston+Supervielle.pdf

4. Corral C. (Septiembre 2012). Vibraciones mecánicas. Universidad Nacional del Sur. Obtenido desde http://lcr.uns.edu.ar/fvc/NotasDeAplicacion/FVC-Carolina

5. Vista de Transformada Rápida de Fourier. (s. f.). Obtenido desde https://camjol.info/index.php/fisica/article/view/8276/8495.

6. Heath M.(s.f). Parallel Numerical Algorithms Chapter 13 – Fast Fourier Transform. Department of Computer Science University of Illinois at Urbana-Champaign. Obtenido desde *https* : *//courses.engr.illinois.edu/cs*554/*fa*2015/*notes*/13$_{f}ft_{8}up.pdf$

7. Sonzogni , V. Eficiencia de programas paralelos. Santa Fe Argentina: CIMEC. Obtenido de http://venus.santafe-conicet.gov.ar/cursos/moodledata/17/Transpeficiencia.pdf

8. Tinetti, F. Paralelización y Speedup Superlineal en Supercomputadoras Ejemplo con Multiplicación de Matrices. CORE. Retrieved from https://core.ac.uk/download/pdf/301043737.pdf

9. Speedup Ratio and Parallel Efficiency.TechWeb. Boston University. (n.f.). Obtenido desde https://www.bu.edu/tech/support/training-consulting/online-tutorials/matlab-pct/scalability/

10. Maklin, C. (2021, December 13). Fast Fourier Transform - Towards Data Science. Medium. https://towardsdatascience.com/fast-fourier-transform-937926e591cb

11. Prof. Edgar Tista. (2020, May 7). Introducción a la programación paralela p2 [Video]. YouTube. https://www.youtube.com/watch?v=vlM26S0XHFs

12. Schmidt, A. (2013). FFT: Transformada Rápida de Fourier. Universidad Nacional del Sur, Avda. Alem 1253, B8000CPB Bahía Blanca, Argentina. Obtenido de http://lcr.uns.edu.ar/fvc/NotasDeAplicacion/FVCSchmidt-Schmidt

13. Bernardo J. [3Blue1Brown] (2020). ¿Qué es la tranformada rápida de Fourier? Una introducción visual[Video]. Youtube. https://www.youtube.com/watch?v=h4PTucW3Rm0

14. Brian Douglas. (2013, January 11). Introduction to the Fourier Transform (Part 1) [Video]. YouTube. https://www.youtube.com/watch?v=1JnayXHhjlg

15. Reducible. (2020, November 14). The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever? [Video]. YouTube. https://www.youtube.com/watch?v=h7apO7q16V0kmara lo pusimos doble

16. Worner S. (s.f). Fast Fourier Transform. Swiss Federal Institute of Technology Zurich. Obtenido desde http://pages.di.unipi.it/gemignani/woerner.pdf

17. 3Blue1Brown Español. (2020, September 21). ¿Qué es la Transformada de Fourier? Una introducción visual [Video]. YouTube. https://www.youtube.com/watch?v=h4PTucW3Rm0

18. Camacho A.(2013, Noviembre 5). Transformada Discreta de Fourier[Video].Youtube. https://www.youtube.com/watch?v=ysjbvYvHZOY

19. (s.f) Parallel Fast Fourier Transform. Studies in Parallel and Distributed System. Obtenido de https://cs.wmich.edu/gupta/teaching/cs5260/5260Sp15web/studentProjects/tiba&hussein/03278999.pdf