



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ESTRUCTURAS DE DATOS Y ALGORITMOS II

VALENZUELA ASCENCIO GUSTAVO

# PROYECTO 2

Fecha de entrega: 4 de Diciembre de 2023

## Versión en español

## Objetivo

Encontrar el punto de equilibrio PE de un algoritmo con complejidad mayor o igual a $O(n* \log(n))$.

## Descripción

Implementar un algoritmo serial alg-ser en lenguaje C cuya complejidad sea mayor o igual a $O(n * \log(n))$. Después, implementar dos versiones paralelas del alg-ser en lenguaje C usando OpenMP:

• Primera versión utilizando todos los constructores, funciones y cláusulas vistas en clase, excepto el constructor task alg-par-no-task
• Segunda versión utilizando el constructor task alg-par-task Una vez implementados correctamente los tres algoritmos (alg-ser, alg-par-no-task y alg-par-task), se debe contabilizar el tiempo de ejecución para el mayor número de instancias posibles (según el algoritmo implementado).

El tiempo de ejecución para cada instancia se debe guardar en un archivo. A partir del archivo que tiene el número de instancias y el tiempo de ejecución para cada algoritmo (alg-ser, alg-par-no-task y alg-par-task), se debe implementar un programa en Python que permita graficar esos tiempos en una gráfica comparativa entre los 3 algoritmos. Al final, se debe mostrar, tanto en la gráfica como en los tiempos guardados, el punto de equilibrio PE. El PE está dado por el punto aproximado de intersección entre la curva serial y las curvas paralelas, de tal manera que se pueda concluir a partir de qué instancias de entrada IE es viable paralelizar (IE > P E) y a hasta qué instancias es mejor usar el algoritmo serial (IE < P E). Así mismo, se debe concluir qué manera de paralelizar fue más eficiente entre las versiones paralelas alg-par-no-task y alg-par-task.

## Instrucciones.

El proyecto está divido en dos partes: 1. Proponer un algoritmo. Cada persona debe proponer un algoritmo a implementar. 2. Implementar el algoritmo. Crear la versión serial, las

versiones paralelas y la gráfica comparativa tal como se menciona en la descripción de este documento. El proyecto se debe subir a la plataforma del SiCCAAD en un archivo comprimido. En el comprimido se debe entregar un documento PDF y el código fuente. El documento PDF debe contener una carátula, la descripción del algoritmo seleccionado (qué hace y cómo funciona), el análisis de complejidad del algoritmo, la gráfica comparativa de los tiempos de ejecución y las conclusiones del trabajo. El código fuente debe incluir todos los archivos requeridos para cumplir con la descripción de este documento.

## Restricciones.

• El algoritmo propuesto debe tener una complejidad mayor o igual a $O(n * \log(n))$
• El algoritmo propuesto no debe ser alguno algoritmo que se haya paralelizado en clase
• El algoritmo propuesto debe ser único en el grupo, es decir, no debe ser igual a alguno otro propuesto por alguien más del grupo
• El proyecto es individual
• La fecha de entrega es, a más tardar, el lunes 4 de diciembre a las 23 horas

### Algoritmo propuesto (Transformada rápida de Fourier de Cooley-Tukey).

La transformada rápida de Fourier es un algoritmo que permite computar la transformada de Fourier discreta (tiempo $O(n^2)$) en tiempo O(nlogn), usando la estrategia “divide y vencerás”. La TRF encuentra aplicaciones en el tratamiento digital de señales, la solución de ecuaciones diferenciales en derivadas parciales, algoritmos de multiplicación rápida de grandes enteros y multiplicación rápida de polinomios.

El algoritmo de la TRF se limita la señal al tamaño de una potencia de dos, debido a que si se maneja un tamaño distinto a este, puede generar imprecisiones debido a las divisiones de la señal que maneja el algoritmo, sin embargo, al ser en su mayoría polinomios, podemos seguir manejando un tamaño de potencia de dos dejando los términos no deseados con un coeficiente igual a cero.

### Definición formal.

Sea $x_n$ una señal aperiódica discreta definida en el dominio del tiempo. Su transformada discreta de Fourier se define de la siguiente manera.

$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}$ cuando k es un entero positivo

En el cual $X_k$ es un conjunto de números complejos. Como se mencionó anteriormente, la definición formal de esta fórmula toma $O(n^2)$ operaciones, sin embargo con el algoritmo de la TRF se obtiene el resultado con O(n log n) operaciones.

**Análisis con modelo RAM (Archivo: serialFFTDemo.c).**

```
// Function to perform in-place bit reversal of a number
unsigned int bit_reverse(unsigned int num, int bits) {
    unsigned int reversed_num = 0; // 3 || 1
    for (int i = 0; i < bits; i++) { // 5(Bits + 1) || 1
        reversed_num = (reversed_num << 1) | (num & 1); // 5(Bits) || 1
        num >>= 1; // 3 || 1
    }
    return reversed_num; // 1 || 1
}

// bit reverse TIME complexity: 10 (Bits) + 12 = O(n)
// bit reverse SPACE complexity: 5  = O(1)

// Cooley-Tukey FFT algorithm
void fft(complex double *a, int n) {
    // Bit-reversal permutation
    for (int i = 0; i < n; i++) { // 5 (n+1) || 1
         int j = bit_reverse(i, log2(n)); // 5(n * O(Bits) + 1) || 1 //
As the input is limited by log2n O(Bits) = log n
        if(j > i){ // 4(n) || 1
            complex double temp = a[i]; // 6(n) || 1
            a[i] = a[j]; // 4(n) || 1
            a[j] = temp; // 4(n) || 1
        }
    }

    // Iterative FFT
     for (int m = 2; m <= n; m *= 2) { //Constantly multiplying variable
m by 2  4(log(n) - 1) || 1
        complex double wm = cexp(2.0 * I * M_PI / m); // 6(log n) || 1
             for (int k = 0; k < n; k += m) { // As m increases
logarithmicly 4(n * log(n) + 1) || 1
```

```
            complex double w = 1.0; // 4(n * log n) || 1
               for (int j = 0; j < m / 2; j++) { // m = log(n) so...
4(log(n) * (n*log(n))+1) || 1
                // log^2(n) = 2 log(n)
                   complex double t = w * a[k + j + m / 2]; // 18 (n *
log(n)) || 1
                complex double u = a[k + j];// 12 (n * log(n)) || 1
                a[k + j] = u + t; // 12 (n * log(n)) || 1
                a[k + j + m / 2] = u - t; // 12 (n * log(n)) || 1
                w *= wm; // 6(n * log(n)) || 1
            }
        }
    }
}

// FFT TIME complexity: 81 (n*log(n)) + 17(n) + 6 log(n) + 5 =
O(n*log(n))
// FFT SPACE complexity: 16 = O(1)
```

**Implementación del algoritmo serial (Archivo: project2.c).**

```
void serialfft(complex double *a, int n, FILE *S, int inst) {
    double start_time = omp_get_wtime();
    // Bit-reversal permutation
    for (int i = 0; i < n; i++) {
        int j = bit_reverse(i, log2(n));
        if (j > i) {
            complex double temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    // Iterative FFT
    for (int m = 2; m <= n; m *= 2) {
        complex double wm = cexp(2.0 * I * M_PI / m);
        for (int k = 0; k < n; k += m) {
            complex double w = 1.0;
            for (int j = 0; j < m / 2; j++) {
                complex double t = w * a[k + j + m / 2];
                complex double u = a[k + j];
                a[k + j] = u + t;
```

```
                a[k + j + m / 2] = u - t;
                w *= wm;
            }
        }
    }
    double end_time = omp_get_wtime() - start_time;
    fprintf(S, "%d, %f\n", inst, end_time);

}
```

**Implementación del algoritmo paralelo sin task (Archivo: project2.c).**

```
void parallelfft(complex double *a, int n, FILE *P,int inst) {
    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        int j = bit_reverse(i, log2(n));
        if (j > i) {
            complex double temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    for (int m = 2; m <= n; m *= 2) {
        complex double wm = cexp(2.0 * I * M_PI / m);
        #pragma omp parallel for
        for (int k = 0; k < n; k += m) {
            complex double w = 1.0;
            for (int j = 0; j < m / 2; j++) {
                complex double t = w * a[k + j + m / 2];
                complex double u = a[k + j];
                a[k + j] = u + t;
                a[k + j + m / 2] = u - t;
                w *= wm;
            }
        }
    }
    double end_time = omp_get_wtime() - start_time;
    fprintf(P, "%d, %f\n", inst, end_time);

}
```

**Descripción del proceso paralelizado:** Dado que el primer ciclo para las permutaciones de bits toma tiempo lineal, nos vamos por el camino fácil y usamos un parallel for.

Ahora nos movemos al proceso principal, únicamente se paraleliza el ciclo que le genera más carga al algoritmo, se intentó paralelizar los tres ciclos, sin embargo el rendimiento disminuye respecto a si solo paralelizamos el segundo ciclo alrededor de un 50%.

**Estrategia utilizada:** Paralelización de ciclos.

### Implementación del algoritmo paralelo con task (Archivo: project2.c).

```
void taskfft(complex double *a, int n, FILE *T, int inst) {
    double start_time = omp_get_wtime();

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop
    for (int i = 0; i < n; i++) {
        int j = bit_reverse(i, log2(n));
        if (j > i) {
            complex double temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    for (int m = 2; m <= n; m *= 2) {
        complex double wm = cexp(2.0 * I * M_PI / m);
        #pragma omp parallel
        #pragma omp single
        #pragma omp taskloop
        for (int k = 0; k < n; k += m) {
            complex double w = 1.0;
            for (int j = 0; j < m / 2; j++) {
                complex double t = w * a[k + j + m / 2];
                complex double u = a[k + j];
                a[k + j] = u + t;
                a[k + j + m / 2] = u - t;
                w *= wm;
            }
```

```
        }
    }
    double end_time = omp_get_wtime() - start_time;
    fprintf(T, "%d, %f\n", inst, end_time);
}
```

**Descripción del proceso paralelizado:** Se probó a usar solo el pragma task para el algoritmo, sin embargo, a pesar de intentar ponerlo en varios lugares del algoritmo, este tardaba como mucho igual que la ejecución serial, entonces decidí que los lugares en los que usaba un parallel for, usaría un taskloop y un single, paralelizando solo para un hilo. Si quitamos este single, nos genera inconsistencia de datos. Esto nos puede llevar a que el algoritmo corra más lento que el paralelo sin task.

## Resultados.

**Hipótesis:** Al principio consideré cambiar el algoritmo por la transformada discreta simple ya que los rendimientos eran muy similares durante los intervalos delimitados por cada potencia de dos, sin embargo, esto podría ser una ventaja ya que podríamos encontrar mucho más fácil el punto de equilibrio y el momento en el que el algoritmo paralelo se vuelve mejor. A su vez ahora podríamos hablar sobre la existencia de un intervalo de equilibrio, pero esto lo veremos más adelante.

Se ejecutaron dos casos de prueba. Ambos se plantean de la misma forma, sin embargo los valores de los coeficientes cambian, agregando casos con coeficientes iguales a cero para simular un uso real del algoritmo.

Caso 1: Se generan números desde 1 hasta $2^{16}$ que sería el tamaño máximo de nuestra señal, generando los polinomios por los tramos que marcan las potencias de dos. Ejemplo

n = 2
1 0
1 2
n = 4
1 2 3 0
1 2 3 4

n = 8

1 2 3 4 5 0 0 0

1 2 3 4 5 6 0 0

1 2 3 4 5 6 7 0

1 2 3 4 5 6 7 8

n = 16

…

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Caso 2: Se generan números desde 1 hasta el 10 de manera uniforme cada 10 términos, hasta llegar al tamaño máximo de nuestra señal $2^{16}$. Es muy parecida al caso anterior. Ejemplo:

n = 2

1 0

1 2

n = 4

1 2 3 0

1 2 3 4

n = 8

1 2 3 4 5 0 0 0

1 2 3 4 5 6 0 0

1 2 3 4 5 6 7 0

1 2 3 4 5 6 7 8

n = 16

1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0

1 2 3 4 5 6 7 8 9 1 0 0 0 0 0 0

…

1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7

**Prueba de consistencia.**

La prueba de consistencia es simple, se guardan los resultados de la transformada aplicada a la señal más grande para cada versión del algoritmo. Estos se guardan en un

archivo. Al ser números complejos, estos archivos poseen parte real e imaginaria, entonces cada archivo tiene dos columnas.

En python, leemos los resultados y los pasamos por esta prueba:

```
def read_file(file_path):
    real_part = []
    imag_part = []

    with open(file_path, 'r') as file:
        # Skip the first line
        next(file)
        for line in file:
            # Split the line into real and imaginary parts
            real_str, imag_str = line.strip().split(',')

            # Convert strings to float
            real_part.append(float(real_str))
            imag_part.append(float(imag_str))

    return real_part, imag_part


file_path = 'resultsSerial.txt'
serialr,seriali = read_file(file_path)
file_path = 'resultsParallel.txt'
parallelr,paralleli = read_file(file_path)
file_path = 'resultsTask.txt'
taskr, taski = read_file(file_path)


for i in range (len(serialr)):
       if  serialr[i]  !=  parallelr[i]  or  serialr[i]  !=  taskr[i]  or
parallelr[i] != taskr[i]:
        print("Data inconsisency at term "+str(i))
        print(seriali[i])
        print(paralleli[i])
        print(taski[i])

for i in range (len(seriali)):
       if  seriali[i]  !=  paralleli[i]  or  seriali[i]  !=  taski[i]  or
paralleli[i] != taski[i]:
```

```
        print("Data inconsisency at term "+str(i))
        print(seriali[i])
        print(paralleli[i])
        print(taski[i])
```

Para el caso 1 la prueba arrojó este único mensaje, el cual no alarma mucho ya que es solo uno y una diferencia de milésimas entre un término, podría ser incluso un cálculo ajeno a la paralelización.

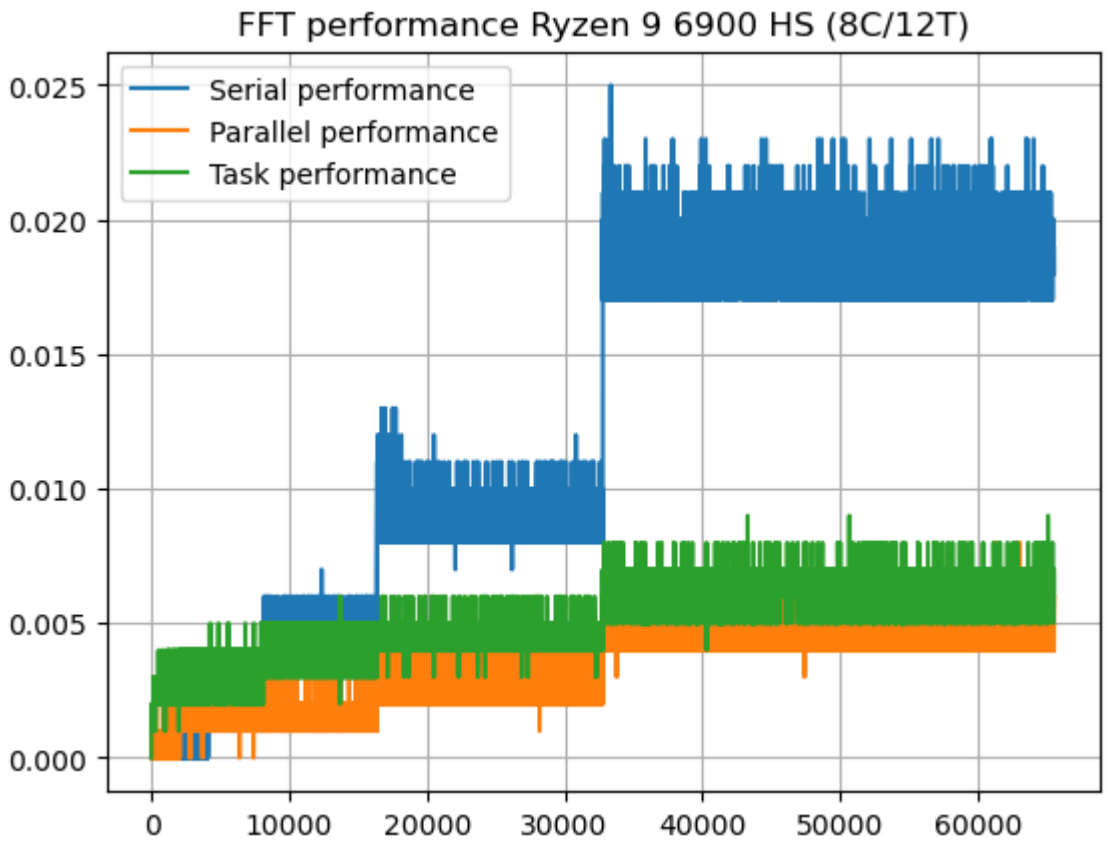
```
Data inconsisency at term 6135
-108189.6283
-108189.62829
-108189.6283
```

Para el caso 2 la prueba no arrojó ningún mensaje y se completó de manera exitosa.

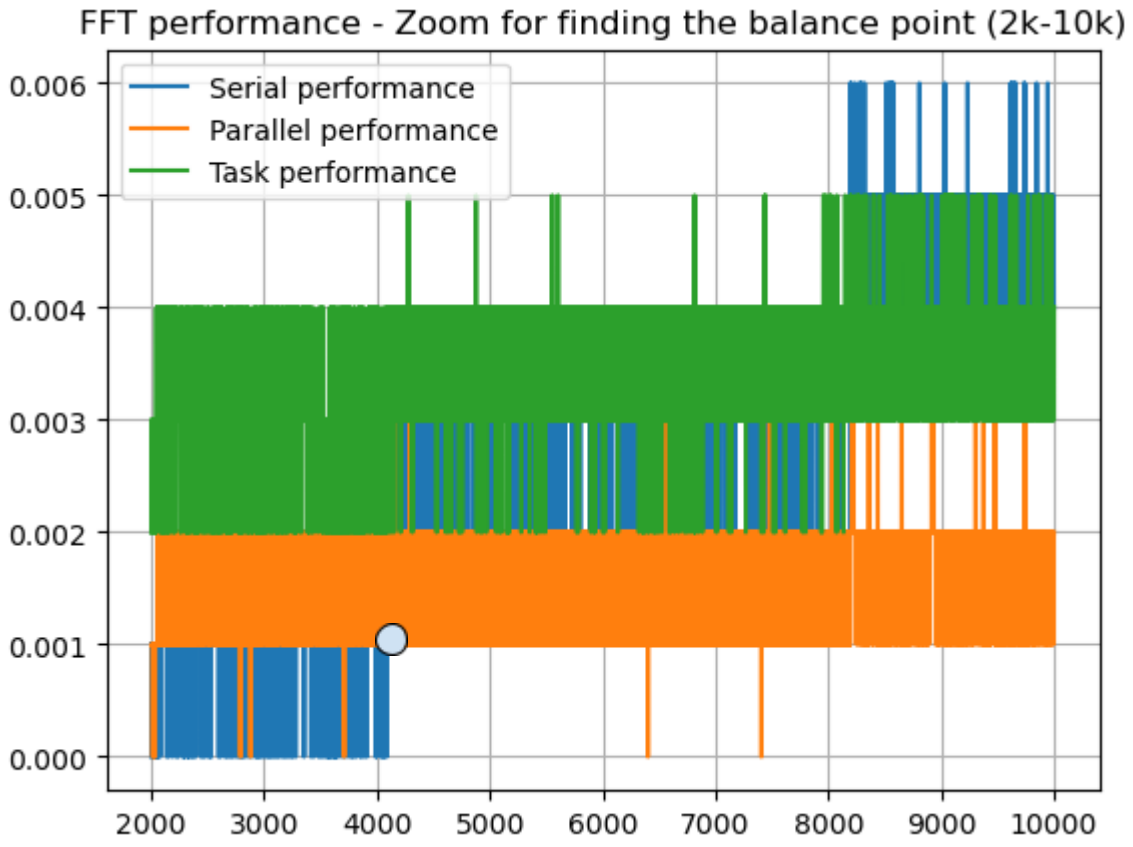
## Gráficas.

Se generaron tres gráficas, una general y dos con intervalos de interés.

**Gráfica general:**




**Zoom en el intervalo [2000,10000] para encontrar el punto de equilibrio**



Mediante observar la gráfica, nos damos cuenta que el punto de equilibrio entre el algoritmo serial y el algoritmo paralelo se encuentra en las 4096 instancias, a partir de ahí las curvas empiezan a tomar caminos distintos.

A su vez podemos confirmarlo mediante el registro de sus tiempos

**Serial**

4082, 0.001000
4083, 0.001000
4084, 0.001000
4085, 0.001000
4086, 0.001000
4087, 0.000000
4088, 0.001000
4089, 0.001000
4090, 0.001000
4091, 0.001000
4092, 0.001000
4093, 0.001000
4094, 0.001000
4095, 0.001000
4096, 0.001000
4097, 0.002000
4098, 0.002000
4099, 0.002000
4100, 0.002000
4101, 0.002000
4102, 0.002000
4103, 0.002000
4104, 0.003000

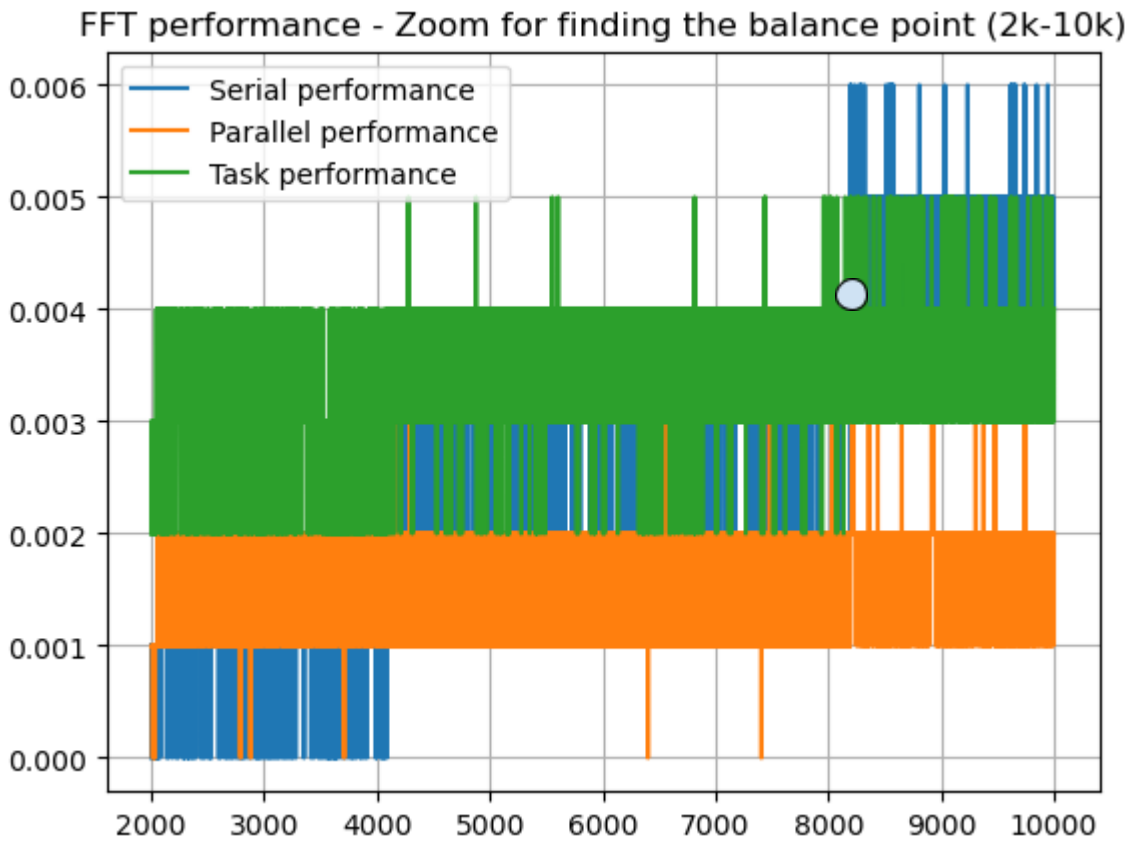
**Parallel**

4082, 0.001000
4083, 0.001000

4084, 0.001000
4085, 0.001000
4086, 0.001000
4087, 0.001000
4088, 0.001000
4089, 0.001000
4090, 0.001000
4091, 0.001000
4092, 0.002000
4093, 0.001000
4094, 0.001000
4095, 0.001000
4096, 0.001000
4097, 0.001000
4098, 0.001000
4099, 0.001000
4100, 0.002000
4101, 0.001000
4102, 0.001000
4103, 0.001000
4104, 0.001000

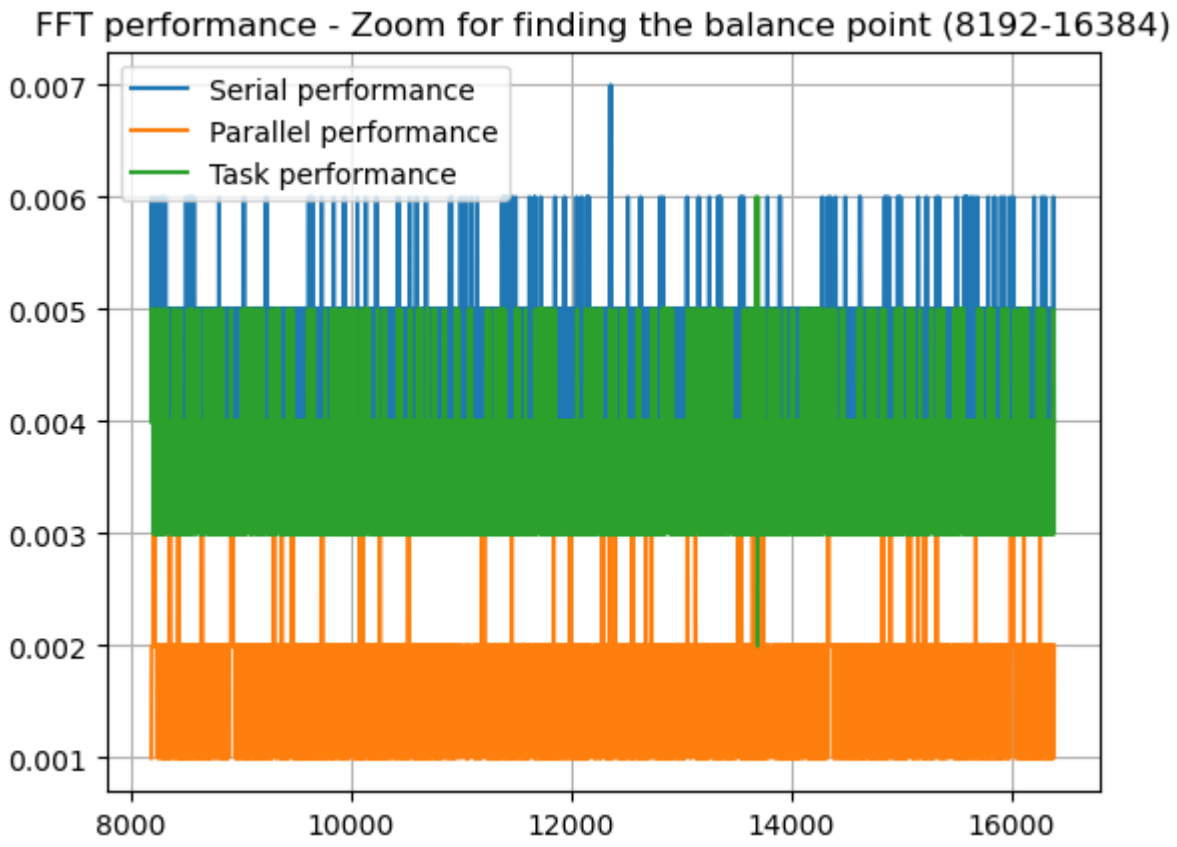
Ahora debemos considerar el punto de equilibrio con task. Este al parecer se encuentra en 8192.

**Zoom en el intervalo [2000,10000] para encontrar el punto de equilibrio**




Haciendo zoom en el intervalo de 8192 a 16384, nos damos cuenta que ahí se encuentra el punto de equilibrio con task y a su vez podemos apreciar un intervalo de equilibrio ya que los resultados son similares durante ese intervalo.

**Zoom en el intervalo [8192, 16384] para encontrar el punto de equilibrio**




**Serial**

8186, 0.002000

8187, 0.002000

8188, 0.002000

8189, 0.002000

8190, 0.002000

8191, 0.002000

8192, 0.002000

8193, 0.005000

8194, 0.004000

8195, 0.004000

8196, 0.005000

8197, 0.006000

8198, 0.005000

8199, 0.006000

8200, 0.005000

**Task**

8186, 0.003000

8187, 0.003000

8188, 0.003000

8189, 0.003000

8190, 0.004000

8191, 0.003000

8192, 0.003000

8193, 0.004000

8194, 0.004000

8195, 0.004000

8196, 0.005000

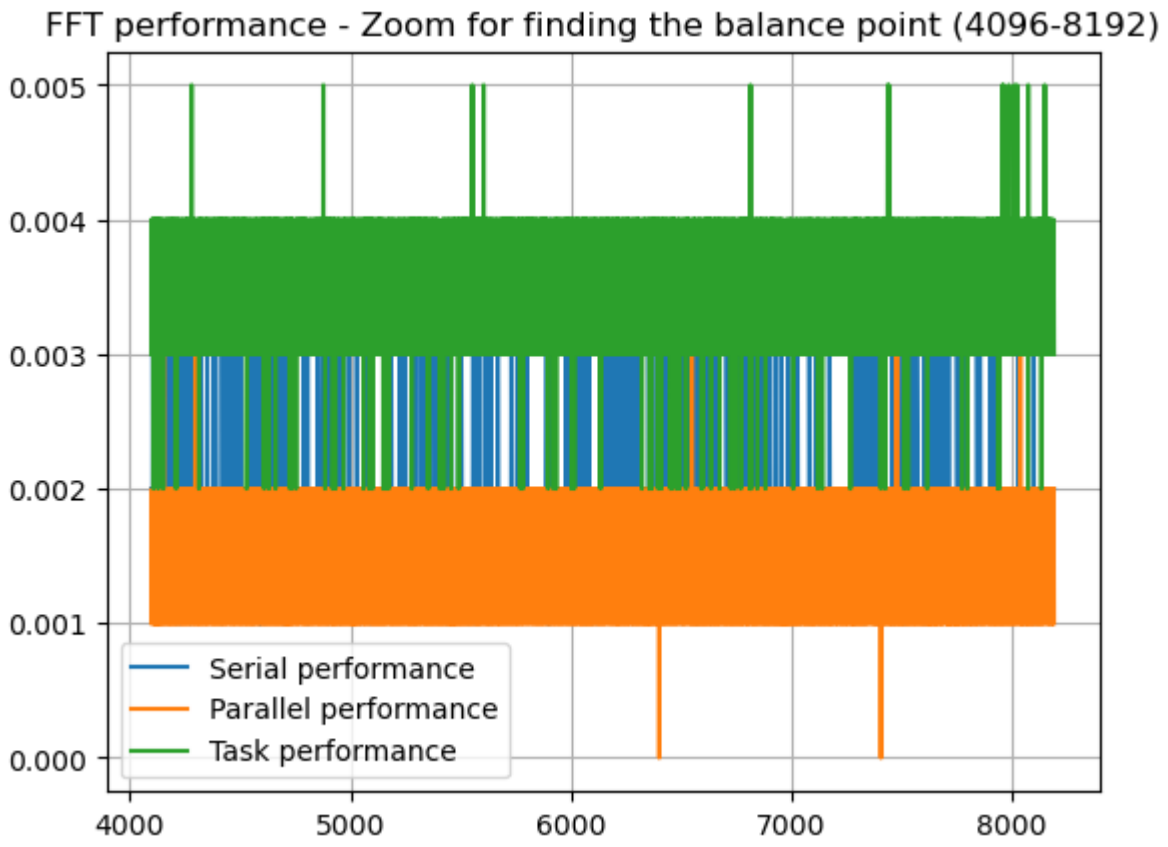
8197, 0.004000

8198, 0.004000

8199, 0.004000

8200, 0.004000

**Zoom en el intervalo [4096, 8192] para analizar un posible intervalo de equilibrio.**




Si bien este es el intervalo en el que más se parecen los tiempos, para nada es un intervalo de equilibrio, pero nos da una idea. A partir de 4096 instancias, conviene paralelizar, ya que se observa que la gráfica paralela empieza a colocarse abajo de la serial y de ahí se estabiliza haciendo más grande la diferencia a mayor número de instancias.

**Conclusiones:**

Los constructores task y taskloop son bastante útiles, ya que paralelizan tareas sin la necesidad de que tengan algún formato específico como un for, sin embargo para este caso no es la mejor elección ya que el task normal tarda lo mismo que el algoritmo serial y el taskloop tiene la restricción de solo poder usarlo con single para evitar la inconsistencia de datos. Sigue siendo una buena opción, pero la paralela mediante paralelismo de ciclos con pragma for es bastante mejor.

A su vez conviene paralelizar a partir de una instancia de grado mayor a 4096, ya que hasta antes de ese valor, el algoritmo serial es mejor.