Universidad Nacional Autónoma de México
Facultad de Ingeniería

Materia:
Estructuras de datos y algoritmos II

Grupo:03

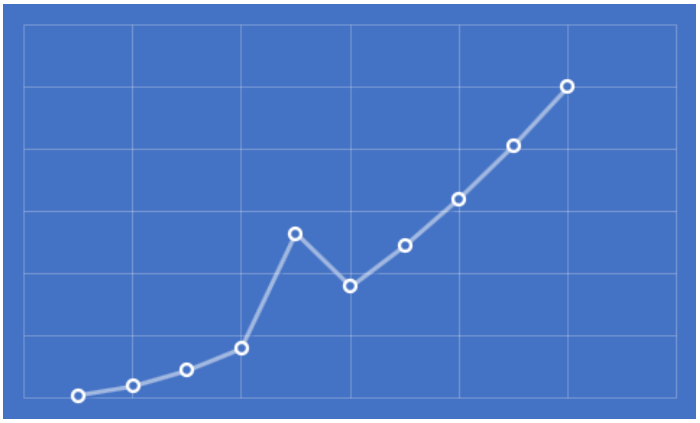
Semestre
2023-1

Nombre:
De la rosa Lara Gustavo

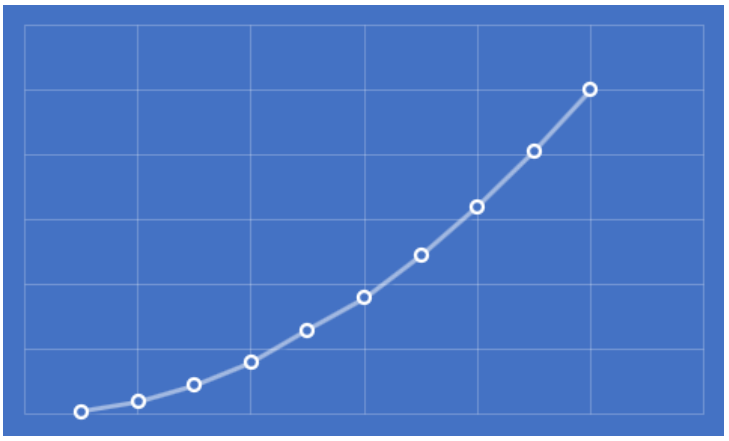
Número de cuenta:
318215961

Proyecto 2: Algoritmos paralelos

Programa a implementar:
El programa implementado tiene el objetivo de limpiar datos de un conjunto. La lógica consiste en que, al tener un rango de datos de la siguiente manera:



Se pueda obtener una gráfica más uniforme de la siguiente manera:



Este procedimiento se realiza de la siguiente manera: en primer lugar identificamos los lugares de la gráfica en los cuales los valores están descuadrados, posteriormente se calcula la media entre los valores inmediatos, el predecesor y el sucesor para cambiar el valor dañando lo menos posible su magnitud pero pudiendo regresar una uniformidad a la gráfica.

Simulación de datos:
Para poder obtener los datos con las características necesarias para emplear esta estrategia de limpieza, simulé los datos de la función x^2 como primer paso; posteriormente al 10% de los valores generados y en posiciones aleatorias sume un valor que haga desvariar la entrada.

x	f(x) = x^2
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Después del proceso ->

x	f(x) = x^2
1	1
2	4
3	9
4	16
5	33
6	36
7	49
8	64
9	81
10	100

Al obtener estos arreglos con el 10% de sus elementos descuadrados podemos pasar al objetivo de este proyecto, la limpieza de los datos.

Código empleado para la simulación:

```
void simulacion(int array[],int numero_entradas){
    int i;
    for (i=0;i<numero_entradas;i++){
        array[i] = generar_numero(i);
    }
    for (i=0;i<numero_entradas/10;i++){ //Al 10% de los datos
        int indice_random = rand()%numero_entradas;
        array[indice_random] += 10; // Le suma para descuadrarlo
    }
}
```

```
int generar_numero(int x){
    return x*x;
}
```

Reparación de datos

Una vez generado el arreglo correspondiente en la simulación, se comienza a tratar los datos, en primer lugar se busca cuáles de los valores en el eje Y no corresponden a su entrada en X para que después de conocer su ubicación se pueda obtener el valor que tomará su lugar, el cual será la media entre su antecesor y su sucesor. Esto último solo tiene una excepción, si el valor que no corresponde a su lugar es el que tiene la posición 0, únicamente se sustituye por 0 sin hacer el proceso de obtener la media.

Código programa serial:

```
void reparar_datos(int array[],int tamano_array){
    int media,i;
    for (i=0;i<tamano_array;i++){
        if (array[i] != generar_numero(i)){
            if (i == 0){
                media = 0;
                array[i] = media;
            }else{
                media = (array[i-1] + array[i+1]) / 2;
                array[i] = media;
            }
        }
    }
}
```

Código programa paralelo:

Para el programa paralelo se divide el trabajo entre todos los hilos disponibles antes de empezar a trabajar; considerando un residuo para tamaños de instancia no divisibles de manera entera entre el número de hilos. Este residuo se le da como trabajo extra al último hilo.

Para comenzar a reparar los datos, ya teniendo cuánto trabajo hará cada hilo, el cual está delimitado por sus variables privadas inició_intervalo y fin_intervalo, el programa hace el mismo proceso que el código serial, identifica cual es el valor que debería tener y si está descuadrado lo sustituye por la media de su antecesor y su sucesor.

```
void reparar_datos(int array[],int tamano_array){
    int tamano_intervalo=0,residuo=0;
    tamano_intervalo = tamano_array/omp_get_num_threads();
    residuo =  tamano_array%omp_get_num_threads();
    #pragma omp parallel
    {
        int inicio_intervalo=0,fin_intervalo=0;
        inicio_intervalo = tamano_intervalo * omp_get_thread_num();
        fin_intervalo =  tamano_intervalo * omp_get_thread_num() + tamano_intervalo;

        if (omp_get_thread_num() == omp_get_num_threads() - 1){
            fin_intervalo += residuo;
        }

        int i,media;
        for(i=inicio_intervalo;i<=fin_intervalo;i++){
            if (array[i] != generar_numero(i)){
                if (i==0){
                    media = 0;
                    array[i] = media;
                }else{
                    media = (array[i-1] + array[i+1]) / 2;
                    array[i] = media;
                }
            }
        }

    }
}
```

Generación de datos

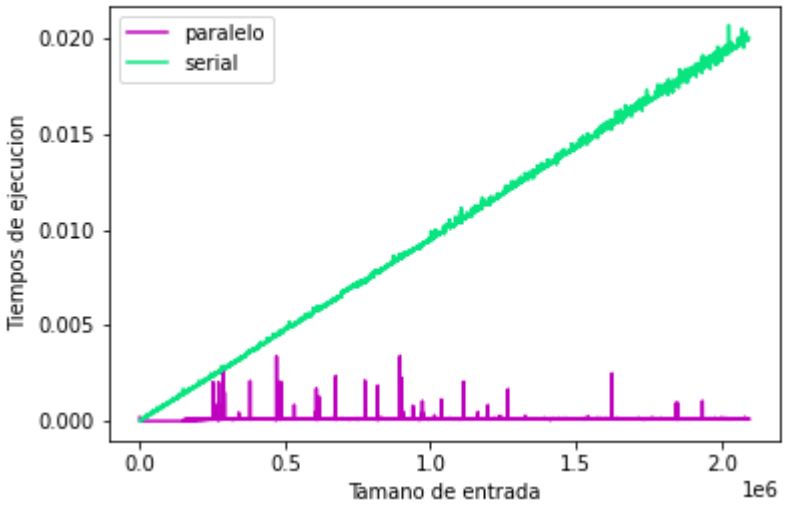
Para medir cuánto tiempo se tarda en ejecutar cada proceso y registrarlo se tiene el siguiente código

```
int main(){
    srand(time(NULL));
    arch = fopen("datos_paralelo.csv","a+");
    int i;
    double tiempo,t_i,t_f;
    for (i=0;i<=2095000;i+=1){
        int array[i];
        simulacion(array,i);
        t_i = omp_get_wtime();
        reparar_datos(array,i);
        t_f = omp_get_wtime();
        tiempo = t_f-t_i;
        escribir_archivo(i,tiempo);
    }

    fclose(arch);
}
```

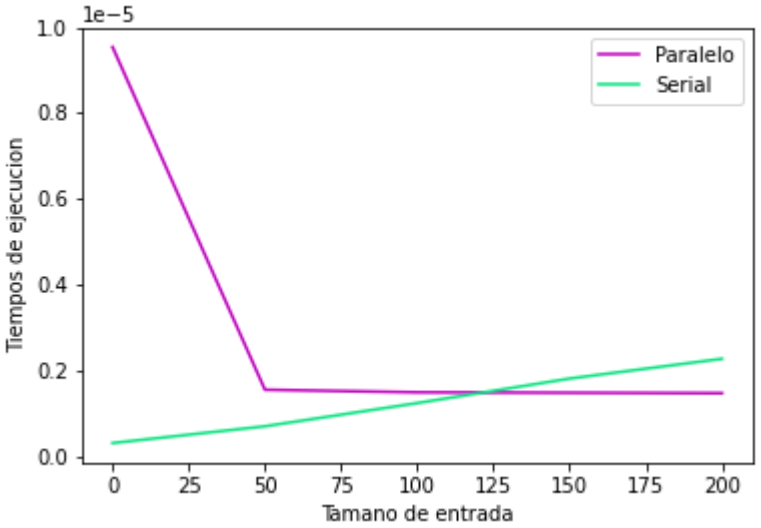
```
void escribir_archivo(int entradas,double tiempo){
    fprintf(arch,"%d,%.8f\n",entradas,tiempo);
}
```

Gráficas:
Una vez terminado el proceso de registro de datos y con ayuda del lenguaje Python podemos graficar los resultados



En esta gráfica podemos observar el comportamiento de los dos programas,donde claramente el programa serial es muy inferior al programa paralelo en la comparación de tiempo de ejecución. En esta gráfica se usaron datos entre el rango [1,2095000] haciendo saltos de 100 en 100.

Punto de equilibrio
En algunos casos el programa serial es igual o incluso más rápido que el programa paralelo, pero esto únicamente se ve en instancias muy pequeñas, además de que el programa serial es superado por el paralelo antes de las 12 instancias de entrada.
Lo anterior lo podemos ver en la siguiente figura, en la que únicamente se ve la comparación de los algoritmos para las primeras 200 entradas.



Podemos observar que el punto de equilibrio entre ambos algoritmos se encuentra en las 124 entradas, en ese punto los algoritmos tienen el mismo tiempo de ejecución.

Conclusión de la eficiencia:
Para un número de instancias menor a 120 el algoritmo serial es más rápido que el paralelo, entre las 120 y las 130 entradas prácticamente no hay diferencia entre implementar uno u otro, pero para un número de instancias más grande que 130 ya es más conveniente implementar el algoritmo paralelo.

Conclusión:
Conocer en qué punto puede ser más eficiente plantear una solución de forma serial o paralela permite manejar grandes cantidades de procesamiento de una manera en la que se aprovechen las cualidades del procesador al máximo, esto permite a los desarrolladores desempeñar sus funciones tomando en cuenta que existe otra forma de hacer las cosas que puede ayudar a los programas que requieren mucho procesamiento a disminuir su tiempo de ejecución.