Juan Esteban Navarro Camacho
Taller 2

1. ¿En qué consiste OpenMP?

OpenMP es un modelo de implementación para apoyar el paso de la implementación de algoritmos paralelos. Deja la responsabilidad de diseñar los algoritmos paralelos apropiados al programador (mediante directivas y otros indicadores dentro del código) y / u otras herramientas de desarrollo.

OpenMP aporta un conjunto de directivas del compilador que describen el paralelismo en el código fuente, junto con una biblioteca de soporte de subrutinas disponibles para las aplicaciones.

2. ¿Cómo se define la cantidad de hilos en OpenMP?

Definiendo y exportando una variable de entorno establecida como:

```
export OMP_NUM_THREADS=2
```

3. ¿Cómo se crea una región paralela en OpenMP?

Estableciendo en el código a paralelizar la directiva respectiva:

```
#pragma omp parallel /* define multi-thread section */
{
/** Código a ser ejecutado  */
}
```

4. ¿Cómo se compila un código fuente en c para utilizar OpenMP y qué encabezado debe incluirse?

Comando de Instalación: Ref

```
sudo apt-get install libomp-dev
```

Compilación en GCC:
Se asume que el archivo fuente se llama main.c, observar que se agrega el flag respectivo de openmp.

```
gcc -o main.out -fopenmp main.c
```

5. ¿Cuál función me permite conocer el número de procesadores disponibles para el programa?

Para el lenguaje de C/C++ existe la siguiente función:

```
int omp_get_num_threads(void);
```

Retorna el número de threads disponibles durante la ejecución de una región paralela.\

Juan Esteban Navarro Camacho
Taller 2

Realice un print con la función con los procesadores de su computadora.

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
#pragma omp parallel
      printf("Number of threads = %d\n", omp_get_num_threads());
}
```

Screenshot generado

```
junavarro@junavarro-Latitude-5480:/media/junavarro/work1/backup/projects/tec/arqui2/taller2$ ./main2.out
Number of threads = 8
Number of threads = 8
Number of threads = 8
Number of threads = 8
Number of threads = 8
Number of threads = 8
Number of threads = 8
Number of threads = 8
junavarro@junavarro-Latitude-5480:/media/junavarro/work1/backup/projects/tec/arqui2/taller2$
```

6. ¿Cómo se definen las variables privadas en OpenMP?¿Por qué son importantes?

Se utiliza una cláusula private en el código, este tipo de variables privadas permiten que cada subproceso que ejecuta instrucciones reciba una nueva variable temporal por subproceso del mismo tipo que la variable externa. Dentro del bloque paralelo, toda referencia a la variable externa será reemplazada por la variable temporal, permitiendo así la protección del dato que está siendo compartido. Referencia

Mediante el uso de la directiva `private(x)` es como se establece una variable privada dentro del contexto de OpenMP.

```c
 int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1;
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

7. ¿Cómo se definen las variables compartidas en OpenMP?¿Cómo se deben actualizar valores de variables compartidas?

Referencia. Mediante el modificador **`static`** se puede establecer una variable compartida.

De forma automática si no se define previamente como **`private`** en la directiva de OpenMp para la región paralela.

Los punteros obtenidos mediante la llamada ***malloc.***

Por último mediante el modificador **`shared`** en la construcción paralela, indicando que variables deben de ser compartidas.

Para actualizar las variables compartidas: Se utiliza las directivas: Referencia

```
#pragma omp atomic
```

que permite un acceso seguro a la variable compartida.

8. ¿Para qué sirve flush en OpenMP?

Referencias. Referencia2. Esta operación hace que la vista temporal de la memoria de un hilo sea consistente con la memoria y aplica un orden en las operaciones de memoria de las variables explícitamente especificadas o implícitas.

9. ¿Cuál es el propósito del **`pragma omp single`**? ¿En cuáles casos debe usarse?

La directiva omp single identifica una sección de código que debe ser ejecutada por un solo hilo disponible. **Referencia.**
Los otros subprocesos asociados a la región paralela, que no ejecutan el bloque, esperan en una barrera implícita al final de la construcción única a menos que se especifique una cláusula nowait Referencia.

Útil en escenario donde el código es Not thread safe.

10. ¿Cuáles son tres instrucciones para la sincronización? Realice una comparación entre ellas donde incluya en cuáles casos se utiliza cada una.

Referencia
Referencia 2
Referencia 3

Barrier	Mutual Execution	Locks
Define un punto en el código donde todos los subprocesos activos se detendrán hasta que todos los subprocesos hayan llegado a ese punto.	Se posee: ● **Critical**: Una sección crítica funciona adquiriendo una cerradura, que conlleva una	Garantiza que algunas instrucciones solo pueden ser realizadas por un proceso a la vez.

	sobrecarga sustancial. • **Atomic**: La sintaxis de las secciones atómicas se limita a la actualización de una única ubicación de memoria. •	
Utilidad		
Con esto, puede garantizar que ciertos cálculos estén terminados.	Permitir que solo un hilo ejecute una parte del código, aquel donde se entra en condición de carrera.	Se asegura de que algunos elementos de datos sólo puedan ser tocados por un proceso a la vez

11. ¿Cuál es el propósito de **reduction** y cómo se define?

Las cláusulas de reducción son cláusulas de atributos de intercambio de datos que se pueden utilizar para realizar algunas formas de cálculos de recurrencia en paralelo.

Referencia
La cláusula de reducción de OpenMP le permite especificar una o más variables privadas de subprocesos que están sujetas a una operación de reducción al final de la región paralela.

```
{
   int sum = 0;
   #pragma omp parallel for reduction(+ : sum)
      for (int i = 0; i < count; ++i)
      {
      sum += fibonacci(i);
      }
}
```

En el código anterior mediante la construcción de `reduction( + : sum)` se indica que el aporte realizado por cada thread será incluido en la variable sum la cual es compartida.

3.Análisis

Para este primer código debe realizar lo siguiente:

Juan Esteban Navarro Camacho
Taller 2

1. Identifique cuáles secciones se pueden paralelizar, así como cuáles variables pueden ser privadas o compartidas. Justifique.

Atributo	Identificadores	Justificación
Sección paralelizable	Loop for	Basado en la propiedad conmutativa y asociativa de la suma , es posible descomponer una iteración de 100x10-6 steps en N jobs de (100/N) x10-6 steps por job y sumar los resultados parciales al final.
Variables privadas	sum	pues cada thread actualizará el propio para luego hacer la respectiva suma al final.
Variables compartidas	el valor del step	pues es solo de lectura

2. ¿Qué realiza la función omp get wtime()?

Referencia
Devuelve un valor igual al tiempo transcurrido del reloj de pared en segundos desde algún tiempo en el pasado. El tiempo real en el pasado es arbitrario, pero se garantiza que no cambiará durante la ejecución del programa de aplicación. El tiempo devuelto es un tiempo por subproceso, por lo que no es necesario que sea coherente globalmente en todos los subprocesos que participan en una aplicación.

3. Compile haciendo uso de OpenMP y ejecute el código modificando el parámetro de número de steps.

Con el fin de proporcionar una entrada parametrizada de la cantidad de steps se agrega al código la asignación mediante **argv.**

```
int main(int argc, char *argv[])
{
   if (argc == 2)
   {
        num_steps = atoi(argv[1]);
   }
```

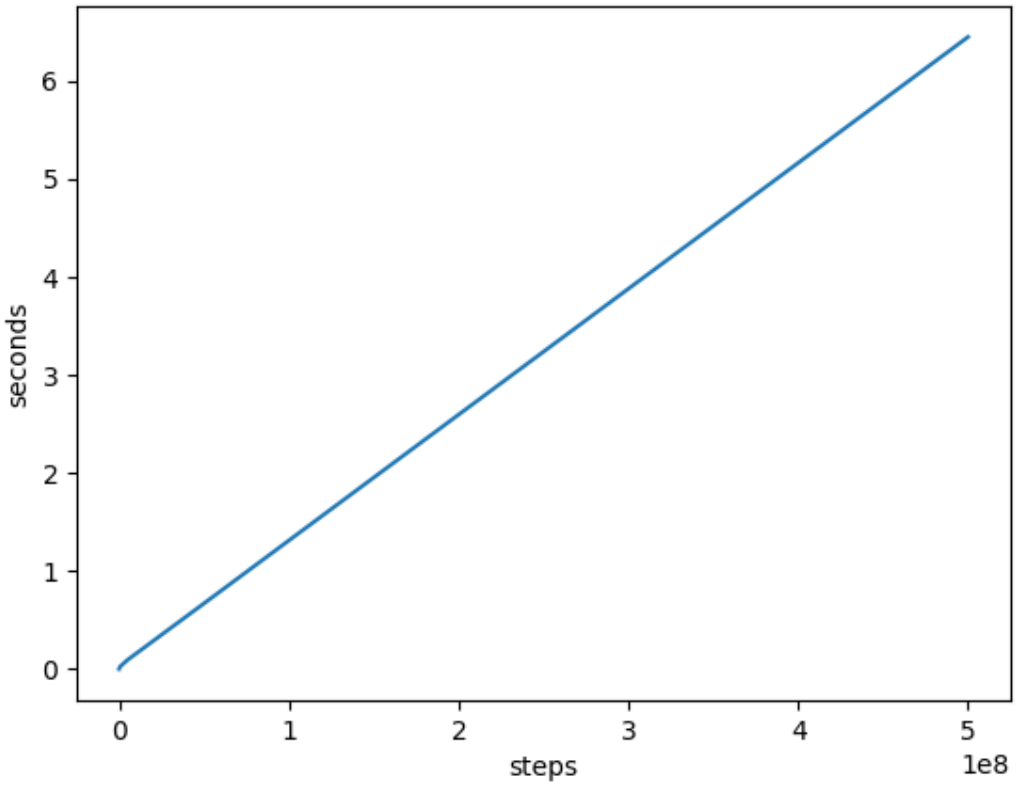
Ejecución:

Juan Esteban Navarro Camacho
Taller 2

```
bash: ./: Is a directory
junavarro@junavarro-Latitude-5480:/media/junavarro/work1/backup/projects/tec/arqui2/taller2$ ./pi.out 500

 pi with 500 steps is 3.141593 in 0.000024 seconds
 junavarro@junavarro-Latitude-5480:/media/junavarro/work1/backup/projects/tec/arqui2/taller2$ ^C
```

4. Mediante un gráfico de tiempo vs steps, pruebe 6 diferentes valores de steps. Explique brevemente el comportamiento ocurrido.



Este gráfico realiza cálculos de Pi cambios exponenciales desde 10^1 a 10^8.

El tiempo que tarda en realizar la integral es proporcional a la cantidad de pasos que realiza el algoritmo, por lo tanto entre mas paso deba a hacer, más preciso será el valor de pi y más tardará.

El código fuente pi loop.c presenta el mismo cálculo, pero utilizando regiones paralelas.

Para este segundo código debe realizar lo siguiente:

Juan Esteban Navarro Camacho
Taller 2

1. Explique cuál es el fin de los diferentes pragmas que se encuentran?

`omp_set_num_threads(i);`	Establece la cantidad de threads que realizaran el trabajo de calcular el valor de Pi
`#pragma omp parallel`	Establecer la región paralela.
`#pragma omp single`	Establecer las siguientes instrucciones como ejecutables solo para un thread.
`#pragma omp for reduction(+:sum)`	Combinar los aportes realizados por todos los threads para el cálculo de Pi

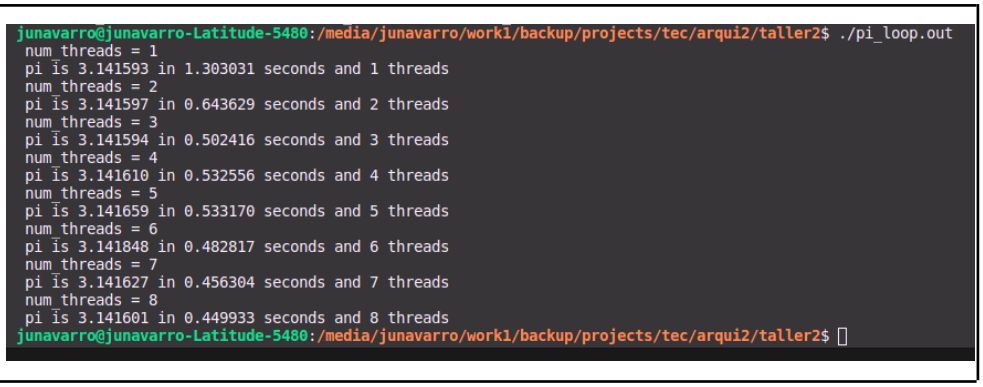
2. ¿Qué realiza la función omp get num threads()?

Obtener la cantidad de threads por defecto o definido en la variable de entorno: `OMP_NUM_THREADS`

3. En la línea 41, el ciclo se realiza 4 veces. Realice un cambio en el código fuente para que el ciclo se repite el doble de la cantidad de procesadores disponibles para el programa. Incluya un screenshot con el cambio.

Cambio en el código	`step = 1.0/(double) num_steps;`
	`for (i=1;i<=8;i++){`
	`sum = 0.0;`
	`omp_set_num_threads(i);`
	`start_time = omp_get_wtime();`

Juan Esteban Navarro Camacho
Taller 2

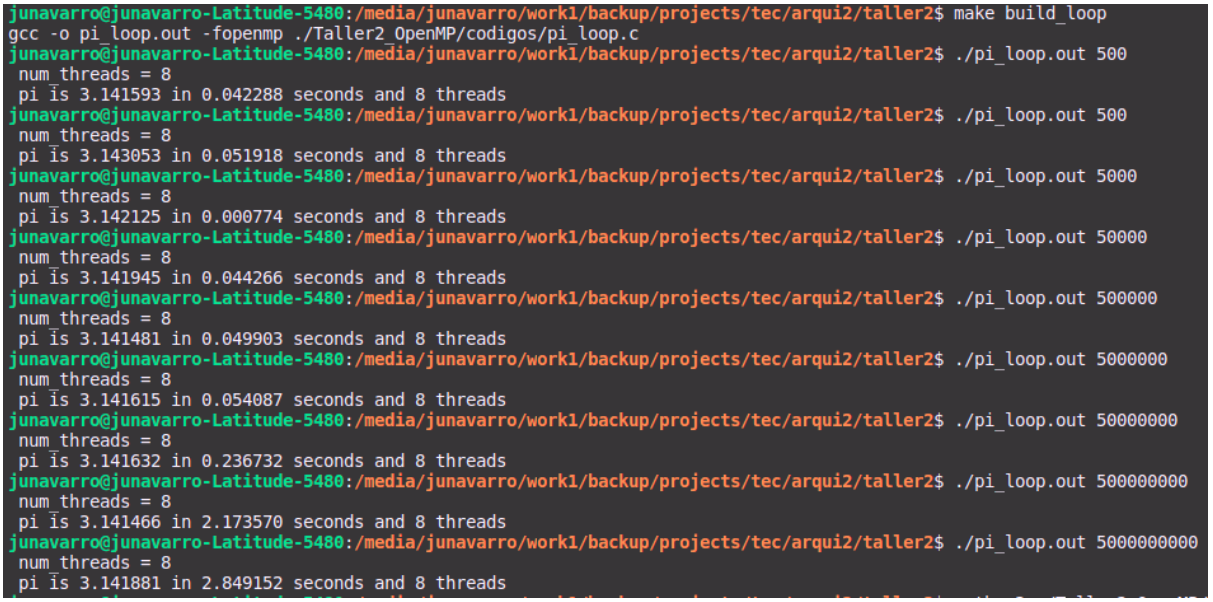
Resultado del cambio	

4. Compile haciendo uso de OpenMP y ejecute el código modificando el parámetro de número de steps.

Cambios realizados al código:

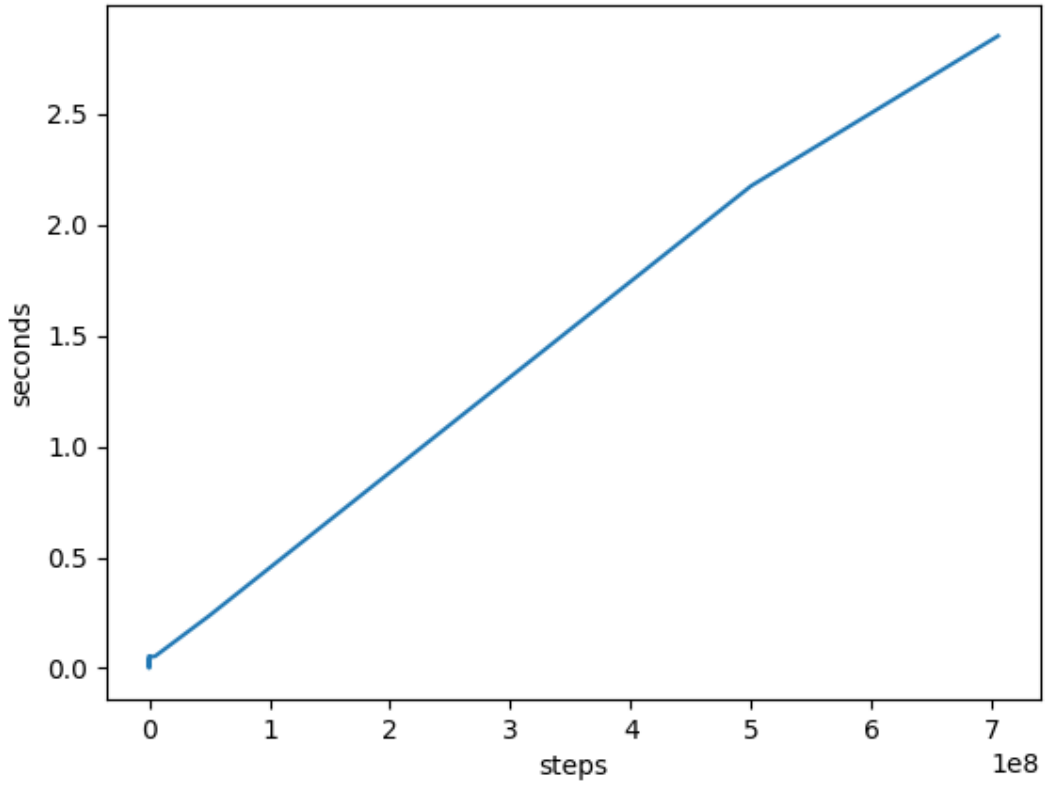
1. Se agrega `int argc, char *argv[]` para parametrizar los steps.
2. Se elimina bucle `for` y se establece una cantidad fija de threads a 8.
3. Se utiliza una función para guardar los cambios en un archivo `.dat`
4. Se realiza un script en python para ver los resultados.

Screenshot de los resultados:



Juan Esteban Navarro Camacho
Taller 2

5. Mediante un gráfico de tiempo vs steps, pruebe 6 diferentes valores de steps. Explique brevemente el comportamiento ocurrido.



La escala en el eje horizontal representa la cantidad de steps, desde 10^2 hasta el orden de 10^8.

6. Compare los resultados con el ejercicio anterior.

Para la solución que involucra threads es importante notar que para una cantidad pequeña de steps el overhead de crear hilos y luego unificar los resultados puede no ser despreciable respecto al tiempo total de la tarea.
Por otro lado, para una cantidad de steps alta 10^6, los beneficios de la paralelización se ven más apreciables versus su versión serial.

4.Ejercicios prácticos

1. Realice un programa en C que aplique la operación SAXPY de manera serial y paralelo (OpenMP). Compare el tiempo de ejecución de ambos programas para al menos tres tamaños diferentes de vectores.
[Referencia de SAXPY]

Juan Esteban Navarro Camacho
Taller 2

Versión serial

```c
#include <stdio.h>
#include <omp.h>
#include "backUpData.c"
int vectorSize = 100000000;
int main(int argc, char const *argv[])
{

    if (argc == 2)
    {
        vectorSize = atoi(argv[1]);
    }

       /* code */
       float *Y = malloc(vectorSize * sizeof(float));
       float *X = malloc(vectorSize * sizeof(float));
       float *Z = malloc(vectorSize * sizeof(float));
       float scalar = 3.5;
       double start_time, run_time;
       start_time = omp_get_wtime();
       for (size_t i = 0; i < vectorSize; i++)
       {
       X[i] = 5.5;
       Y[i] = 6.3;
       Z[i] = scalar*X[i]+ Y[i];
       /* code */
       }
       run_time = omp_get_wtime() - start_time;
       printf("SAXPY %f \n",run_time);
       return 0;
}
```

Versión paralela 8 threads

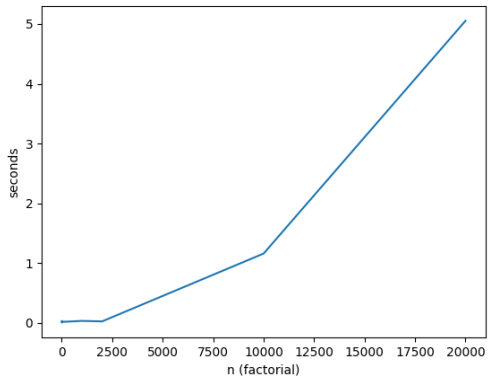
```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include "backUpData.c"
int vectorSize = 100000000;
int main(int argc, char const *argv[])
{
       if (argc == 2) vectorSize = atoi(argv[1]);
       omp_set_num_threads(8);
       float *Y = malloc(vectorSize * sizeof(float));
       float *X = malloc(vectorSize * sizeof(float));
       float *Z = malloc(vectorSize * sizeof(float));
       float scalar = 3.5;
       double start_time, run_time;
       start_time = omp_get_wtime();
#pragma omp parallel
       {
#pragma omp for
```

Juan Esteban Navarro Camacho
Taller 2

```
        for (size_t i = 0; i < vectorSize; i++)
        {
                X[i] = 5.5;
                Y[i] = 6.3;
                Z[i] = scalar * X[i] + Y[i];
        }
        }
        run_time = omp_get_wtime() - start_time;
        printf("SAXPY %f \n", run_time);
        return 0;
}
```

Cantidad de elementos para cada vector	Tiempo ejecución [s]	
	SAXPY Serial	SAXPY Paralelo 8 Threads
50000	0.009846	0.023639
5000000	0.071443	0.035521
500000000	25.955990	6.866580

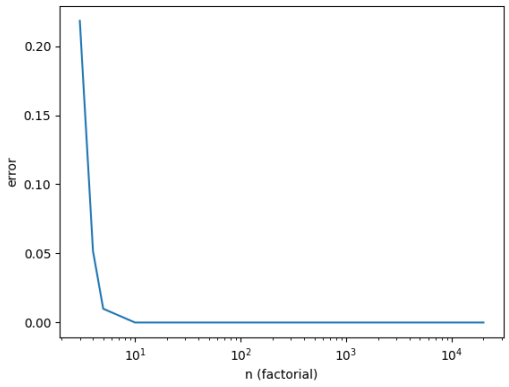
2. Realice un programa en C utilizando OpenMP para calcular el valor de la constante e. Compare los tiempos y qué tan aproximado al valor real para 6 valores distintos de n.



Se visualiza la cantidad de segundos requeridos en función de n para el cálculo de factoriales.

Tarea abordada con 8 threads.

Juan Esteban Navarro Camacho
Taller 2



Error asociado a un valor constante de:

```
#define REAL_EULER
2.71828182845904523536028747135 27
```

respecto al orden del parámetro n mas alto calculado para su factorial respectivo.

euler.c paralelo

Pruebas realizadas:

```
./euler.out 3
./euler.out 4
./euler.out 5
./euler.out 10
./euler.out 20
./euler.out 100
./euler.out 200
./euler.out 1000
./euler.out 2000
./euler.out 10000
./euler.out 20000
```

El parámetro suministrado es el n asociado al factorial respectivo de la sumatoria.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include "backUpData.c"
#include <math.h>
#define REAL_EULER 2.71828182845904523536028747135 27
long num_steps = 1000;
double step;

long double factorial(int n)
{
      long double result = 1.0L;
      if (n == 0)
      return 1;

      for (int i = 1; i <= n; i++)
```

```
	{
	result = result * i;
	}
	return result;
}

int main(int argc, char *argv[])
{
	if (argc == 2)
	{
	num_steps = atoi(argv[1]);
	}
	int i;
	long double euler = 0.0L;
	double start_time, run_time;
	omp_set_num_threads(8);
	start_time = omp_get_wtime();
#pragma omp parallel
	{
#pragma omp for reduction(+ \
                          : euler)
	for (i = 0; i < num_steps; i++)
		euler = euler + (1 / factorial(i));
	}
	run_time = omp_get_wtime() - start_time;
	printf("\n euler is %Lf in %f seconds and %d threads\n", euler,
run_time, 8);
	char *dataToStore = (char *)malloc(100 * sizeof(char));
	long double error = fabsl((long double)REAL_EULER - euler);
	sprintf(dataToStore, "%ld %Lf %lf %.20Lf \n", num_steps, euler,
run_time, error);
	printf(" Error:  %.20Lf \n", error);
	writeData(dataToStore);
}
```