	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jorge Solano
Asignatura:	Estructura de Datos y Algoritmos II
Grupo:	2
No de Práctica(s):	11
Integrante(s):	Martínez Ostoa Néstor Iván
Semestre:	2019-1
Fecha de entrega:	2/Noviembre/2018
Observaciones:	

CALIFICACIÓN: ______________

Desarrollo:

Ejercicios 1:

1. Programa que aproxime PI de manera serial junto con su tiempo de ejecución

```
[edaII02alu18@samba 29.10.18]$ ./pi
PI: 3.141593, time: 0.002666 miliseconds
[edaII02alu18@samba 29.10.18]$ ./pi
PI: 3.141593, time: 0.003062 miliseconds
[edaII02alu18@samba 29.10.18]$ ./pi
PI: 3.141593, time: 0.003064 miliseconds
[edaII02alu18@samba 29.10.18]$
```

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

long numSteps = 100000;
double dx;

int main() {
        int i;
        double x,pi,sum = 0.0;
        dx = 1.0/(double) numSteps;
        double start, end = 0.0;
        start = omp_get_wtime();
        for(i=0;i<numSteps;i++){
                x = (i+0.5)*dx;
                sum += 4.0/(1.0+x*x);
        }
        end = omp_get_wtime() - start;
        pi = sum*dx;
        printf("PI: %f, time: %f miliseconds \n", pi, end);
        return 0;
}
```

2. Programa que aproxime PI de manera paralela junto con su tiempo de ejecución

```
[[edaII02alu18@samba 29.10.18]$ ./pi_parallel
PI: 3.126384, time: 0.006583 miliseconds
[[edaII02alu18@samba 29.10.18]$ ./pi_parallel
PI: 3.141593, time: 0.006764 miliseconds
[[edaII02alu18@samba 29.10.18]$ ./pi_parallel
PI: 3.141593, time: 0.006690 miliseconds
[edaII02alu18@samba 29.10.18]$
```

```c
#include <omp.h>
#define NUM_THREADS 4

long numSteps = 100000;
double dx;

int main() {
	int i;
	double x,pi,start,end = 0.0;
	double sum[NUM_THREADS];

	dx = 1.0/(double) numSteps;
	start = omp_get_wtime();

	#pragma omp parallel
	{
		int id = omp_get_thread_num();
		sum[id] = 0.0;
		int j;
		for (j = id; j < numSteps; j=j+NUM_THREADS){
			x = (j+0.5)*dx;
			sum[id] += 4.0/(1.0+x*x);
		}
	}
	end = omp_get_wtime() - start;

	for (i = 1; i < NUM_THREADS; i++) {
		sum[0] += sum[i];
	}
	pi = sum[0]*dx;
	printf("PI: %f, time: %f miliseconds \n", pi, end);
	return 0;
}
```

Ejercicios 2:

1. Programa que prueba diversas funciones de OpenMP

```
[edaII02alu18@samba 31.10.18]$ ./ejercicio1
Procs: 4
Max threads: 0.000000
In parallel?: 0
Threads: 4
ID: 3
In parallel?: 1
Threads: 4
ID: 1
In parallel?: 1
Threads: 4
ID: 0
In parallel?: 1
Threads: 4
ID: 2
In parallel?: 1
[edaII02alu18@samba 31.10.18]$
```

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

int main() {
	//disable dynamic adjustment of number of threads
	omp_set_dynamic(0);
	int procs = omp_get_num_procs();
	printf("Procs: %d\n", procs);
	printf("Max threads: %f\n", omp_get_max_threads());
	omp_set_num_threads(procs);
	printf("In parallel?: %d\n", omp_in_parallel());

	#pragma omp parallel
	{

		int threads = omp_get_num_threads();
		printf("Threads: %d\n",threads);
		int id = omp_get_thread_num();
		printf("ID: %d\n", id);
		printf("In parallel?: %d\n", omp_in_parallel());
	}
}
```

2. Programas que demuestren la funcionalidad de las funciones de *scheduling,* tanto estáticos como dinámicos .

2.1 Scheduling estático

```
[[edaII02alu18@samba 31.10.18]$ ./static_scheduling
Thread 0 iteration 0
Thread 0 iteration 1
Thread 1 iteration 2
Thread 2 iteration 4
Thread 1 iteration 3
Thread 3 iteration 6
Thread 2 iteration 5
Thread 3 iteration 7
[edaII02alu18@samba 31.10.18]$
```

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

#define THREADS 4
#define N 8

int main() {

        int i;
        #pragma omp parallel for schedule(static) num_threads(THREADS)
                for (i = 0; i<N; i++){
                        sleep(i);
                        printf("Thread %d iteration %d\n", omp_get_thread_num(), i);
                }

        return 0;
}
```

2.2 Scheduling dinámico

```
[[edaII02alu18@samba 31.10.18]$ ./dynamic_schedule
Thread 3 iteration 15
Thread 3 iteration 16
Thread 3 iteration 17
Thread 3 iteration 18
Thread 3 iteration 19
Thread 2 iteration 10
Thread 2 iteration 11
Thread 2 iteration 12
Thread 2 iteration 13
Thread 2 iteration 14
Thread 0 iteration 0
Thread 0 iteration 1
Thread 0 iteration 2
Thread 0 iteration 3
Thread 0 iteration 4
Thread 1 iteration 5
Thread 1 iteration 6
Thread 1 iteration 7
Thread 1 iteration 8
Thread 1 iteration 9
```

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

#define THREADS 4
#define N 20

int main() {

        int i;
        #pragma omp parallel for schedule(auto) num_threads(THREADS)
                for (i = 0; i<N; i++){
                        //sleep(i);
                        printf("Thread %d iteration %d\n", omp_get_thread_num(), i);
                }

        return 0;
}
```

3. Programa que demostrara la funcionalidad del constructor *barrier*

```
[edaII02alu18@samba 31.10.18]$ gcc -fopenmp barrier.c -o barrier
[edaII02alu18@samba 31.10.18]$ ./barrier
Running thread with id: 0 from function 1
Running thread with id: 2 from function 1
Running thread with id: 1 from function 1
Running thread with id: 3 from function 1
Running thread with id: 3 from function 2
Running thread with id: 3 from function 2
Running thread with id: 3 from function 2
Running thread with id: 3 from function 2
```

```c
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 4
#define MAX_NUMBER 1000000000

int main(){
        omp_set_num_threads(NUM_THREADS);
        int A[NUM_THREADS], B[NUM_THREADS], id;
        #pragma omp parallel
        {
                id = omp_get_thread_num();
                A[id] = bigCall(id,1);
                #pragma omp barrier
                B[id] = bigCall(id,2);
        }
        return 0;
}

int bigCall(int id, int numberOfFunction){
        printf("Running thread with id: %i from function %i\n", id, numberOfFunction
);
        int num = 0;
        int i,j=0;
        for(i = 0; i<MAX_NUMBER;i++){
                for (j = 0; j < MAX_NUMBER; j++) {
                        id += 1;
                        num = id;
                }
        }
        return num;
}
```

En este último ejemplo logramos ver que la función 2 tardo mucho más tiempo en ejecutarse pues ocupamos el constructor barrier.