I confirm that I will keep the content of this assignment confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this assignment. I acknowledge that a mark of 0 may be assigned for copied work. Jason Choquette 104337378

For each program (1-4) the implementations are fairly similar. They each calculate π using the *Monte Carlo* method. However, they each use threading in different ways in the *Monte Carlo* process. Comparisons of each of the program's execution time and value of π are provided after the program's analysis.

All Programs:

Each program has the following basic setup:

main: setup threads and wait for threads to complete. Then calculate π.

generateRandomPoints: This is the function that the threads execute.

randomDouble: Returns a random double

calculatePi: calculates
$\pi = 4 \; x \; (\textit{number of points in circle}) \,/\, (\textit{total number of points})$

The analysis will therefore only focus on main and generateRandomPoints.

The following header file is used in each program iteration. Note the hit_count is a global variable as well as the mutex handle ghMutex.

```
#ifndef _A3_PROTOTYPES
#define _A3_PROTOTYPES

#include <stdio.h>                // printf()
#include <conio.h>                // _getch()
#include <windows.h>      // Mutex functions
#include <math.h>                 // sqrt()
#include <time.h>                 // timer
#include <omp.h>          // parallelism library

#define NUMBER_OF_SLAVES 40
#define NUMBER_OF_POINTS 1000000

HANDLE ghMutex;
DWORD hit_count;

double randomDouble();
double calculatePi(DWORD);

#endif  _A3_PROTOTYPES
```

The following are the details of each program and its implementation. For this assignment, I chose to use the Windows API since I have not programmed to this API before.

Program 1 – Single Thread

Write a multi-threaded version of this algorithm that creates a separate thread (the *slave-thread*) to generate a number of random points. The slave-thread will count the number of points that occur within the circle (the `hit_count`) and store that result in the global variable `circle_count`. When the slave-thread has exited, the parent thread (the *master-thread*) will calculate and output the estimated value of π.

```
int main()
{
	DWORD ThreadId;
	HANDLE ThreadHandle;
	DWORD number_of_simulations = NUMBER_OF_POINTS;
	srand((unsigned)time(NULL));	// seed random number generator

	clock_t t = clock();
	// create thread to calculate random number of points
	ThreadHandle =
	  CreateThread(
		NULL,	// default security attributes
		0,		// use default stack size
		(LPTHREAD_START_ROUTINE)generateRandomPoints, // thread function name
		&number_of_simulations,  // argument to thread function
		0,			  // use default creation flags
		&ThreadId);	// returns the thread identifier


	// when slave thread has ended master will calculate estimate of pi.
	if (ThreadHandle != NULL)
	{
		WaitForSingleObject(ThreadHandle, INFINITE);
		CloseHandle(ThreadHandle);
		// call calculation...
		double pi = calculatePi(number_of_simulations);
		t = clock() - t;

		const double time = (double)t / CLOCKS_PER_SEC;

		printf("Calculated value of pi: %f\n", pi);
		printf("Total time taken by CPU: %f seconds\n", time);
	}

	_getch();

	return 0;
}
```

The CreateThread function takes a number of parameters but the relevant parameters are the thread function, the function argument and the threadId address.

A call to WaitForSingleObject blocks until the child thread has completed execution. Upon completion, the thread handle is closed and then the value of π is calculated and the results are printed to the console.

Next is the generateRandomPoints function.

```
void WINAPI generateRandomPoints(LPVOID number_of_simulations)
{
        const DWORD sims = *(DWORD *)number_of_simulations;
        DWORD count = 0;

        while (count++ <= sims)
        {
                double x = randomDouble() * 2.0 - 1.0;
                double y = randomDouble() * 2.0 - 1.0;
                if (sqrt(x*x + y*y) < 1.0)
                        hit_count += 1;
        }
}
```

All thread functions have the same signature: void WINAPI functionName (LPVOID). A cast is required to access the function argument.

For program1, this function is very simple since there is only a single thread accessing the function; the global variable hit_count is increased whenever a point falls inside the circle.

Program 2 – Thread using OpenMP

The OpenMP parallel library provides a very simple interface. The main function has no dependency on threads and appears as though it is a single threaded program:

```
int main()
{
        int number_of_simulations = NUMBER_OF_POINTS;
        srand((unsigned)time(NULL)); // seed random number generator

        clock_t t = clock();

        // call to threaded function
        generateRandomPoints(number_of_simulations);
        double pi = calculatePi(number_of_simulations);

        t = clock() - t;

        const double time = (double)t / CLOCKS_PER_SEC;

        printf("Calculated value of pi: %f\n", pi);
        printf("Total time taken by CPU: %f seconds\n", time);

        _getch();

        return 0;
}
```

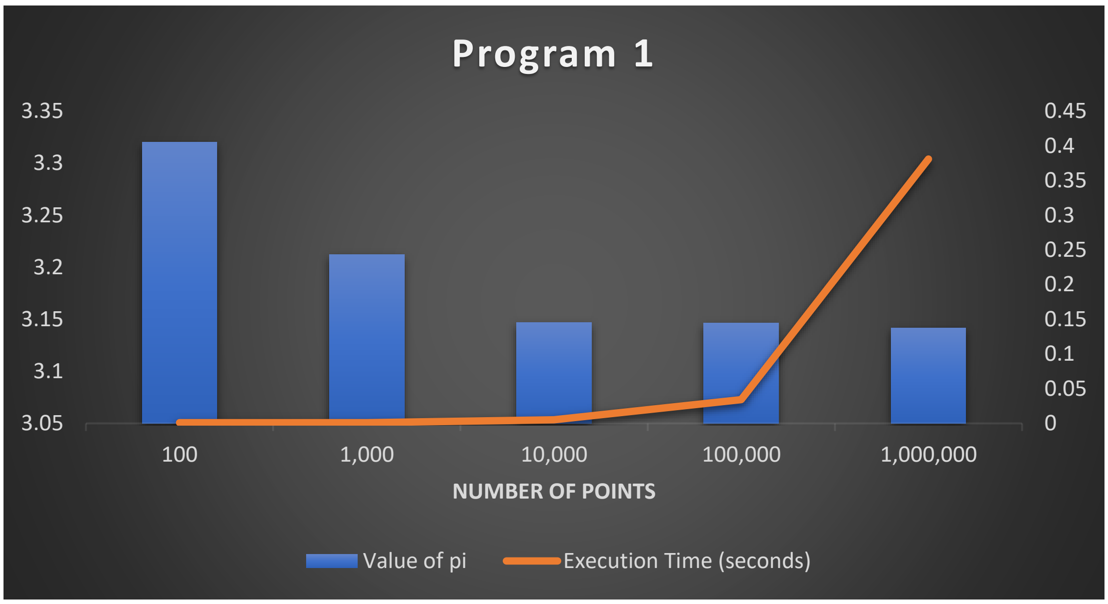
Next is the generateRandomPoints function:

```c
void generateRandomPoints(int number_of_simulations)
{
	int count = 0;

	#pragma omp parallel
	while (count++ <= number_of_simulations)
	{
		double x = randomDouble() * 2.0 - 1.0;
		double y = randomDouble() * 2.0 - 1.0;
		if (sqrt(x*x + y * y) < 1.0)
			hit_count += 1;
```

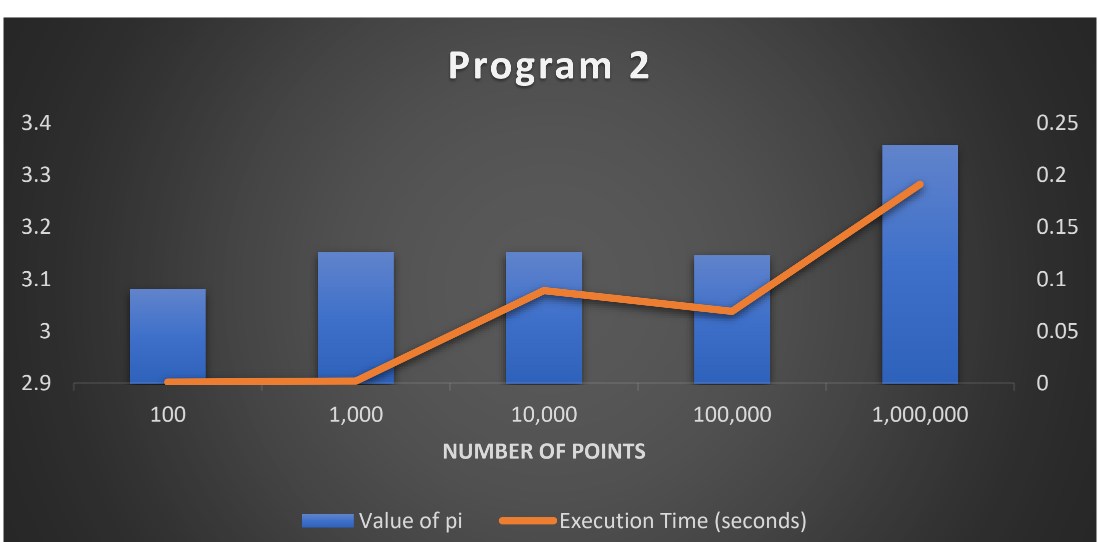
Note the compiler directive #pragma omp parallel. This simple directive is all that is needed to provided parallel processing of the function.

Program 1 and Program 2 – Execution times and values of π

Program 1	Number of Points				
	100	1,000	10,000	100,000	1,000,000
Value of pi	3.32	3.212	3.1468	3.14632	3.141452
Execution Time (seconds)	0.001	0.001	0.005	0.034	0.381



Program 2	Number of Points				
	100	1,000	10,000	100,000	1,000,000
Value of pi	3.08	3.152	3.1516	3.14516	3.357016
Execution Time (seconds)	0.001	0.002	0.089	0.069	0.191



In the preceding graphs, an average for 10 runs of each program was analyzed. In the analysis, for the lower number of points, the Windows API thread appears to be running faster. This is possibly due to the OpenMP library overhead (since it's a complier directive). However, on the largest number of points, the OpenMP library dominates in execution time as the overhead is spread out over more iterations and the advanced parallelism library has a greater effect.

For the accuracy of π, Program1 becomes much more accurate as the number of points increases. For the OpenMP implementation there is a fairly consistent value across each iteration of different numbers of points. However, the final iteration with 1,000,000 points is less accurate. It appears as though program2 is suffering from race conditions which become more apparent as the number of points being iterated increases.

Next is the programs 3 and 4 which use many threads of execution to perform the calculation.

Program 3 – Multithreaded version of program 1

```
// Create a mutex with no initial owner
	ghMutex = CreateMutex(
				NULL,		// default security attributes
				FALSE,		// initially not owned
				NULL);		// unnamed mutex

	if (ghMutex == NULL)
	{
		printf("CreateMutex error: %lu\n", GetLastError());
		return 1;
	}

	DWORD number_of_simulations = NUMBER_OF_POINTS / NUMBER_OF_SLAVES;
	DWORD ThreadId;
	HANDLE aThread[NUMBER_OF_SLAVES];

	// Create worker threads
	for (i = 0; i < NUMBER_OF_SLAVES; i++)
	{
		aThread[i] =
		 CreateThread(
			NULL,		// default security attributes
			0,		// default stack size
			(LPTHREAD_START_ROUTINE)generateRandomPoints,
			&number_of_simulations,  // thread function arguments
			0,		// default creation flags
			&ThreadId);	// receive thread identifier

		if (aThread[i] == NULL)
		{
			printf("CreateThread error: %lu\n", GetLastError());
			return 1;
		}
	}

	clock_t t = clock();

	// Wait for all threads to terminate

	/*  WaitForMultipleObjects has some weird behaviour...
	    If the required number of threads is less than 64,
		then the actual number of threads is required.
		In this case, NUMBER_OF_SLAVES
	*/
	if(NUMBER_OF_SLAVES >= 64)
		WaitForMultipleObjects(MAXIMUM_WAIT_OBJECTS /*64*/,
			aThread,
			TRUE,
			INFINITE);
	else
		WaitForMultipleObjects(NUMBER_OF_SLAVES, aThread, TRUE, INFINITE);

	// Close thread and mutex handles
	for (i = 0; i < NUMBER_OF_SLAVES; i++)
		CloseHandle(aThread[i]);

	CloseHandle(ghMutex);
```

The main function in program 3 is similar to program1 with the only differences being that a mutex is created (and applied in the generateRandomPoints function) and an array of threads is created in a for loop and each thread obtaining the exact same parameters as the CreateThread function in program1.

As noted in the source code comment, the API function WaitForMultipleObjects is dependent on the number of threads required (which took about 5 hours of debugging to solve). For the number of threads 64 and greater, the maximum objects waited for (allowable threads in the wait state) is 64. If less than 64, then the actual number of threads is required. And then a final for loop closes each thread in the array before moving on to calculate π, print the results to the console, etc…

```
void WINAPI generateRandomPoints(LPVOID number_of_simulations)
{
	const DWORD sims = *(DWORD*)number_of_simulations;
	DWORD count = 0;

	const DWORD dwWaitResult = WaitForSingleObject(
		ghMutex,    // handle to mutex
		INFINITE);  // no time-out interval
	switch (dwWaitResult)
	{
		// The thread got ownership of the mutex
	case WAIT_OBJECT_0:
		__try
		{
			while (count++ < sims)
			{
				const double x = randomDouble() * 2.0 - 1.0;
				const double y = randomDouble() * 2.0 - 1.0;
				if (sqrt(x*x + y * y) < 1.0)
					hit_count += 1;
			}
		}
		__finally
		{
			// Release ownership of the mutex object
			if (!ReleaseMutex(ghMutex))
				printf("ReleaseMutex error: %lu\n", GetLastError());
		}

		printf("Thread %lu: wait succeeded\n", GetCurrentThreadId());
		break;

	case WAIT_TIMEOUT:
		printf("Thread %lu: wait timed out\n", GetCurrentThreadId());
		break;

		// The thread got ownership of an abandoned mutex
	case WAIT_ABANDONED:
		return;
	default:;
	}
}
```

The generateRandomPoints function is slightly more complicated. A call to WaitForSingleObject (passing the reference of the mutex) is made with a switch statement immediately following based on the WaitForSingleObject function’s return value.

Inside the switch statement, there is a case statement that will be entered if a thread obtains the mutex. The mutex is required to lock the global variable hit_count. The test of points inside the circle is wrapped in a try-finally statement to ensure proper handling of the mutex lock.

Program 4 – Multithreaded version of program 2

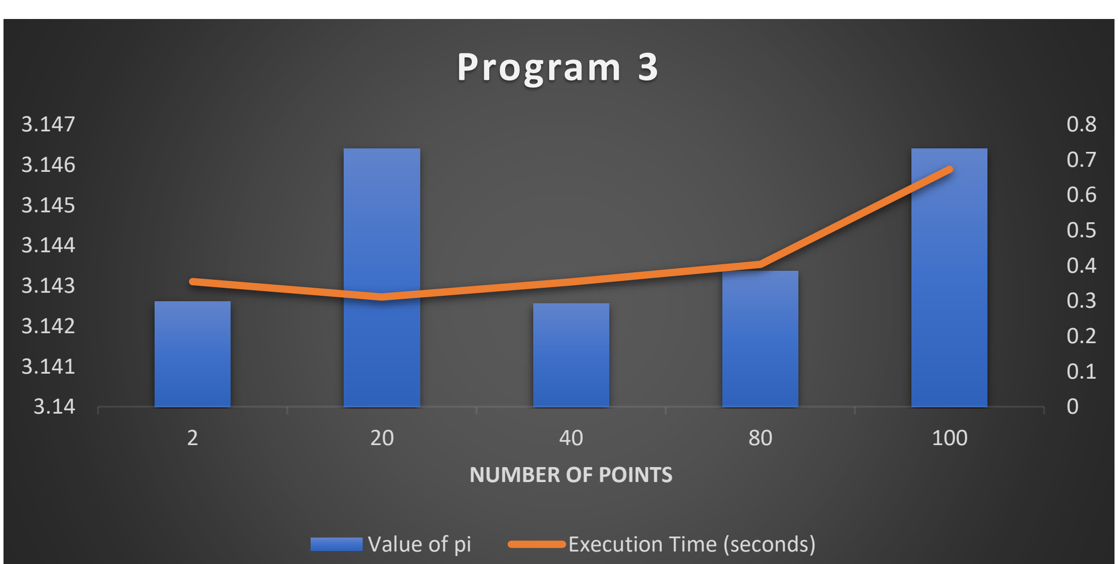
```
void generateRandomPoints(int number_of_simulations)
{
        int sims = number_of_simulations;
        int count = 0;

        #pragma omp parallel num_threads(NUMBER_OF_SLAVES)
        while (count++ <= sims)
        {
                double x = randomDouble() * 2.0 - 1.0;
                double y = randomDouble() * 2.0 - 1.0;
                #pragma omp critical  //-------- Program 4 diff from Program 2
                if (sqrt(x*x + y * y) < 1.0)
                        hit_count += 1;
        }
}
```

Program4's main is identical to program2. So, the focus will be on the generateRandomPoints function. Here the compiler directive is passed an argument of num_threads with that argument taking the number of threads to be produced.

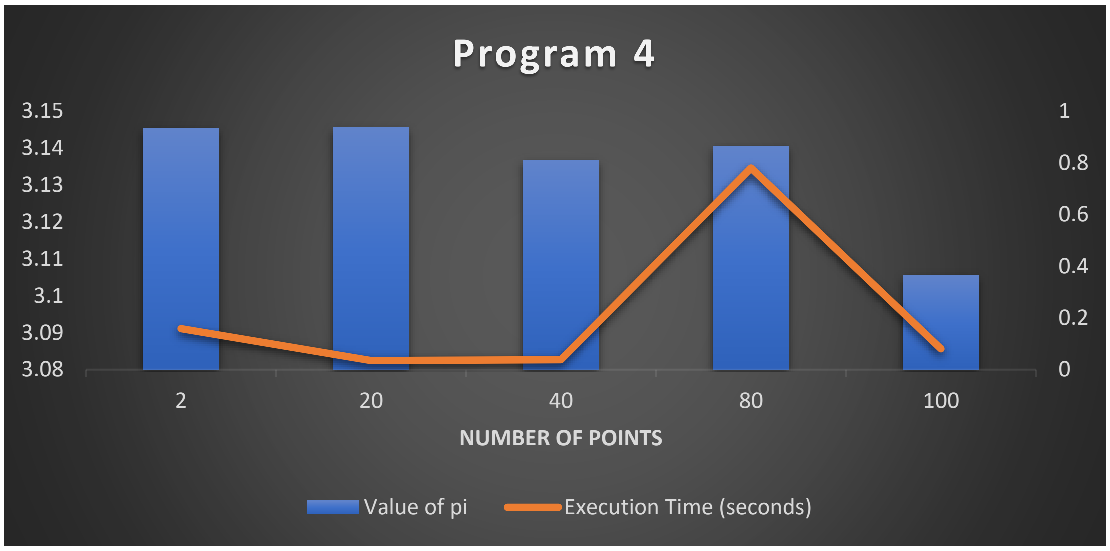
Instead of explicitly creating a mutex, another compiler directive is used to create a critical region for the global variable. Again, the ease of the OpenMp library is quite remarkable.

Program 3 and Program 4 – Execution times and values of π

Program 3 (1,000,000 points)	Number of Slaves				
	2	20	40	80	100
Value of pi	3.142608	3.1464	3.14256	3.14336	3.1464
Execution Time (seconds)	0.355	0.311	0.354	0.404	0.673



Program 4 (1,000,000 points)	Number of Slaves				
	2	20	40	80	100
Value of pi	3.145488	3.14552	3.1368	3.140481	3.1056
Execution Time (seconds)	0.159	0.035	0.038	0.78	0.081



For both programs, both execution times are somewhat slower and π values were more accurate than the averages of program1 and program2.

With program4 utilizing the critical region, the value of π is much more accurate.