Assignment #2 – PI by Tylor Series

Name -Surname : **Onur Çetin**

ID : **20200808050**

Course : **CSE 440 - Parallel Computing**

This project aims to calculate the number Pi using C programming language with multi-threading techniques. Pi will be calculated using Taylor series and this calculation will be performed by parallel computing methods.

Problem and My Solution Strategy

Within the scope of the project, the total number of processes to be calculated and the number of threads to be used for the calculation will be obtained from the user. Based on this information, process allocation will be done and each thread will communicate with the main process by computing its own part.

The process allocation is done in the following steps:

1-The user enters the total number of operations in the total_operations variable.
2-The user is asked to enter the number of threads to be used in the variable num_threads.
3-As many threads as num_threads are created and the interval to be calculated for each of them is determined.
4-Each thread performs the operations in the range assigned to it and adds the result to the total_sum variable.
5-Parallel computation is completed by synchronizing the threads.

With this method, the calculation process is performed in parallel, while data integrity and synchronization between threads are ensured and the correct result is obtained.

gcc -pthread find_pi.c -o find_pi →Provides multi-threading support.

```c
void *computePi(void *args) {
    struct ThreadArgs *threadArgs = (struct ThreadArgs *)args;
    double sum = 0.0;

    for (int i = threadArgs->start; i < threadArgs->end; i++) {
        if (i % 2 == 0) {
            sum += 1.0 / (2 * i + 1);  // Adding positive term
        } else {
            sum -= 1.0 / (2 * i + 1);  // Subtracting negative term
        }
    }

    threadArgs->result = sum;
    pthread_exit(NULL);
}
```

→This code block defines a function to calculate the approximate value of Pi using multi-threading. The function performs addition and subtraction operations between the start and end values in an incoming structure using the Taylor series method, thus distributing the computational operations in parallel and sharing the results between threads. This method increases the efficiency of the program by reducing computation time.

```c
struct ThreadArgs threadArgs[numThreads];
pthread_t threads[numThreads];

int opsPerThread = totalOps / numThreads;
int remainingOps = totalOps % numThreads;
int currentStart = 0;

for (int i = 0; i < numThreads; i++) {
    threadArgs[i].start = currentStart;
    threadArgs[i].end = currentStart + opsPerThread + (i < remainingOps ? 1 : 0);
    currentStart = threadArgs[i].end;

    pthread_create(&threads[i], NULL, computePi, (void *)&threadArgs[i]);
}

struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

double totalPi = 0.0;
for (int i = 0; i < numThreads; i++) {
    pthread_join(threads[i], NULL);
    totalPi += threadArgs[i].result;
}
```

→This block of code starts the parallel computation process by determining the interval at which each thread will compute, based on the total number of operations (addition/subtraction operations) specified by the user and the desired number of threads. threadArgs structures hold the start and end indices at which each thread will compute. the opsPerThread variable determines the number of operations each thread will perform, while the remainingOps variable is used to properly distribute the remaining operations (if any) in the thread. Inside the loop, the pthread_create function is used to create threads and the computePi function is assigned to these threads to start parallel computation. This distributes the workload and improves the efficiency of the program by increasing the performance of parallel computation.

Time Comparison

OPERATION NUMBER	TIME	THREAD NUMBER	PI
1000	0.000058 seconds	16	3.1405926538
10000	0.000070 seconds	16	3.1414926536
100000	0.000236 seconds	16	3.1415826536
1000000	0.000273 seconds	16	3.1415916536
10000000	0.005804 seconds	16	3.1415925536
100000000	0.054892 seconds	16	3.1415926436

```
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 1000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1405926538
Computation time: 0.000058 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 10000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1414926536
Computation time: 0.000070 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1415826536
Computation time: 0.000236 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 1000000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1415916536
Computation time: 0.000273 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 10000000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1415925536
Computation time: 0.005804 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1415926436
Computation time: 0.054892 seconds
onuce@DESKTOP-7S6BRTU:~$
```

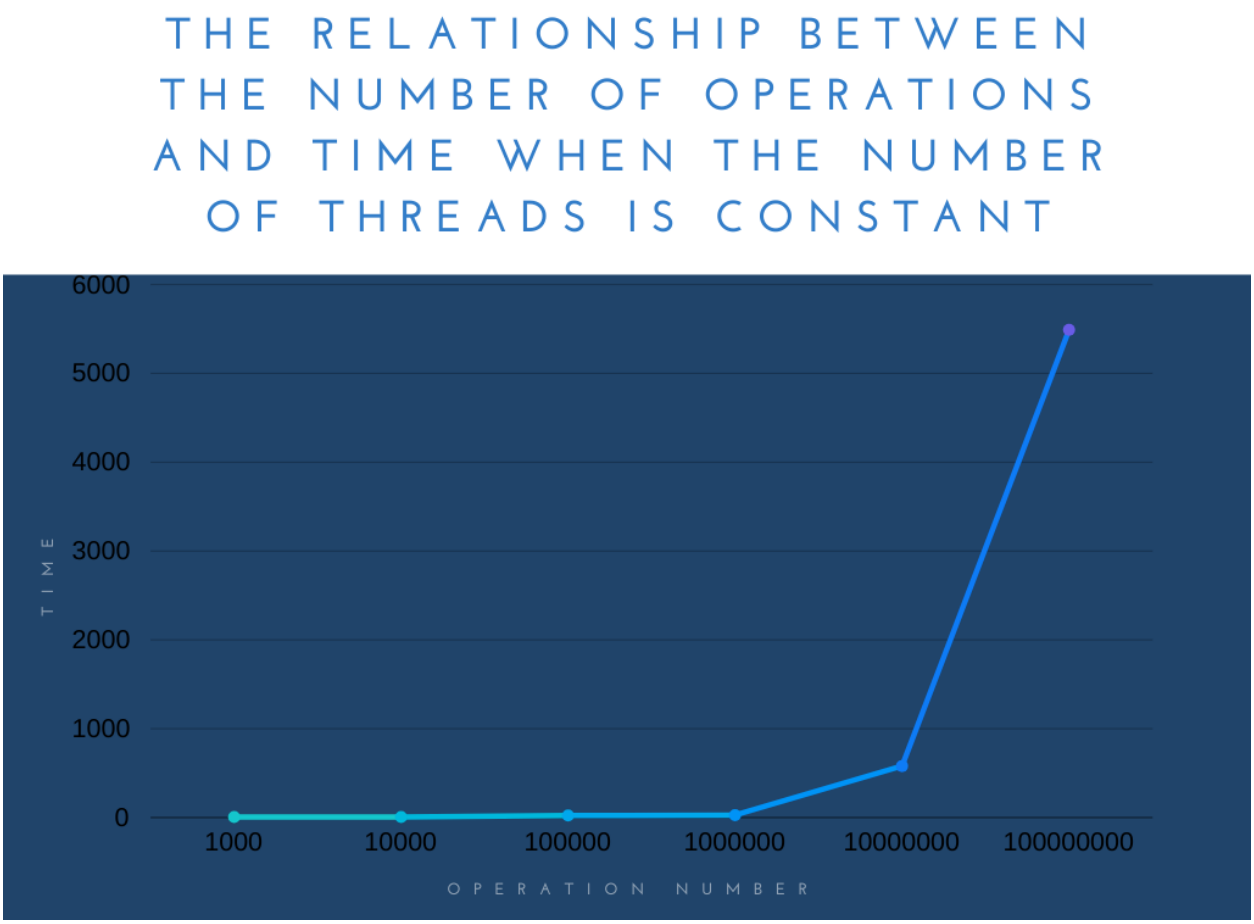
OPERATION NUMBER	TIME	THREAD NUMBER
100.000	0.000516 seconds	1
100.000	0.000298 seconds	2
100.000	0.000279 seconds	4
100.000	0.000227 seconds	8
100.000	0.000119 seconds	16

100.000	0.000351 seconds	32
100.000	0.000268 seconds	64

Since the operation number remains constant, of course the pi number is 3.1415826536 for each thread number.

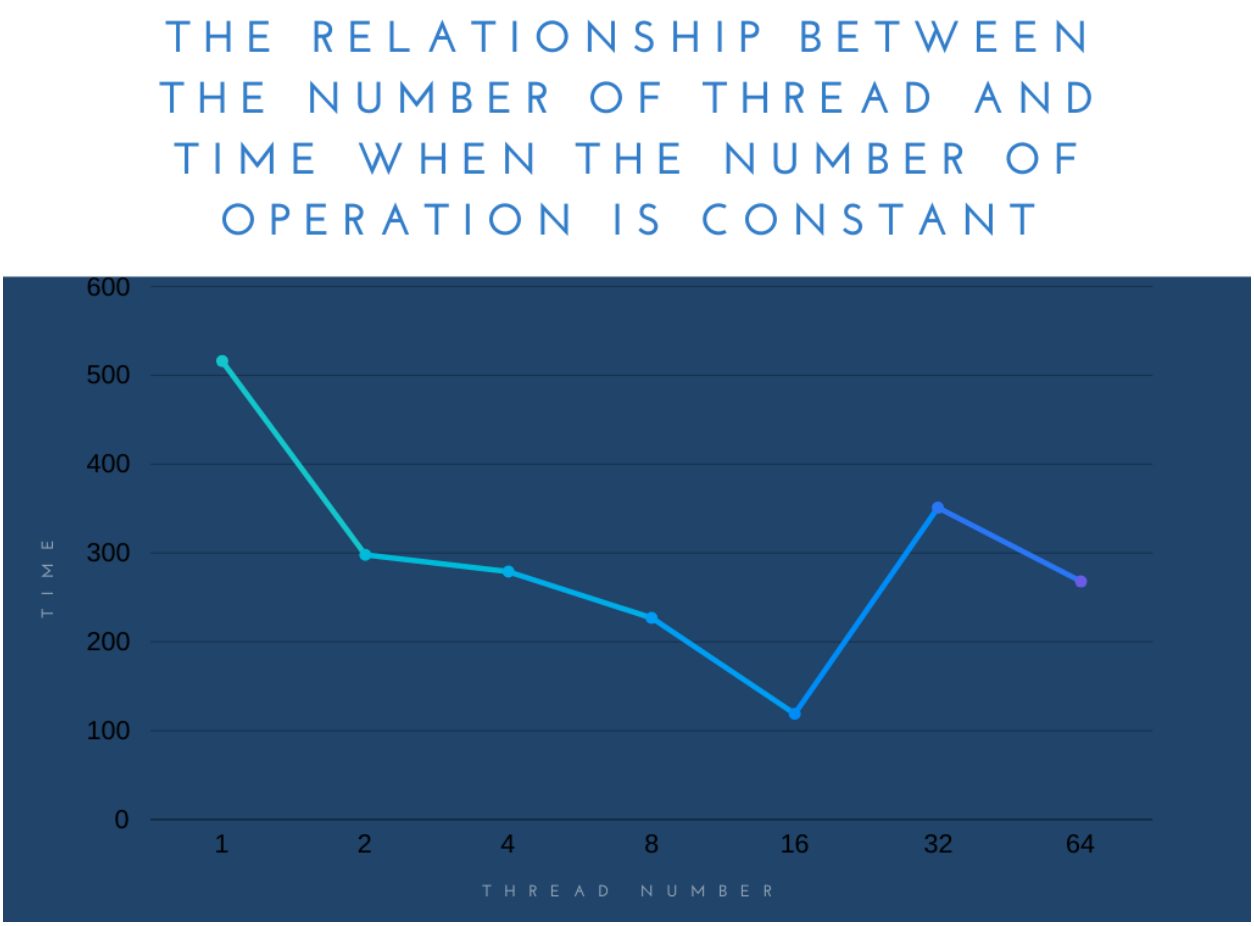
```
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 1
Estimated value of Pi: 3.1415826536
Computation time: 0.000516 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 2
Estimated value of Pi: 3.1415826536
Computation time: 0.000298 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 4
Estimated value of Pi: 3.1415826536
Computation time: 0.000279 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 8
Estimated value of Pi: 3.1415826536
Computation time: 0.000227 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 16
Estimated value of Pi: 3.1415826536
Computation time: 0.000119 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 32
Estimated value of Pi: 3.1415826536
Computation time: 0.000351 seconds
onuce@DESKTOP-7S6BRTU:~$ ./find_pi
Enter the total number of operations (add/subtract): 100000
Enter the number of compute nodes (threads, 1-64): 64
Estimated value of Pi: 3.1415826536
Computation time: 0.000268 seconds
onuce@DESKTOP-7S6BRTU:~$
```

Data Visualizations



→At first, as the number of operations increases, it seems as if there is not much difference, but after a certain stage, the difference becomes strikingly obvious.

→You can see all the net results in the Time comparison section.



→In general, as the number of threads increases, the processing time decreases, but at 32 and 64, there is a slight increase and decrease in the number of threads. When I researched the reason for this, I found that there is something called processor contention in multi-threading. Processor contention occurs when multiple threads try to access processor resources at the same time. It can also vary depending on the current usage of the computer.

→You can see all the net results in the Time comparison section.

Conclusion

1-Thread Count and Workload Balance: For programs with smaller workloads, the management and synchronization tasks associated with using a large number of threads can consume

time and resources. Therefore, balancing the number of threads with the workload usually results in better performance.

2-Pi count and Operation Count: I noticed that the higher the number of operations, the closer the Pi number gets to the actual result. It is similar to an approach similar to the area calculation in the integral, the smaller you divide the area into smaller parts, the closer you find to the actual area. Here, the largest number of operations was the closest to the actual Pi number.

3-Optimal Thread Count: The optimal number of threads may be limited by the number of cores of the processor or the number of physical processors. In this case, it can be understood that using more threads will not provide additional performance.

4-Parallelization Efficiency: Parallelization can be more efficient with large workloads. This is because larger workloads allow threads to be used at full capacity, while smaller workloads are offset by management and synchronization costs.

5-Optimization and Optimization: To improve performance, it is important to optimize the code and determine the optimal number of threads and workload through experiments. Longer experiments with large workloads can yield clearer results.