Abstract

Given the source code for calculating pi using pthreads, the serial code is reproduced and stats are recorded for both the serial and parallelized code.

1 Serial Code

Below is the code reproduced serially (pi_serial.c)

Listing 1: Serial code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define STEPS 100000000
#define STEP_SIZE 1.0/STEPS

double calculate_pi()
{
    double lower = 0.5 * STEP_SIZE;
    double higher = 1; //1/N, 2/N... N/N--->1
    double sum = 0;
    //while fraction is less than 1
    while(lower < higher)
    {
        sum = sum + (STEP_SIZE * sqrt(1 - lower*lower));
        lower = lower + STEP_SIZE;
    }
    return sum*4;
}
int64_t millis()
{   struct timespec now;
    timespec_get(&now, TIME_UTC);
    return ((int64_t) now.tv_sec) * 1000 +
    ((int64_t) now.tv_nsec) / 1000000;
}
int main()
    {   int64_t start = millis();
        double sum = calculate_pi();
        int64_t end = millis();

        printf("Reference PI = %.10lf
        Computed PI = %.10lf\n", M_PI, sum);
        printf("Difference to Reference is
        %.10lf\n", M_PI - sum);
        double time_elapsed = (end - start);
        printf("Time: %f ms\n", time_elapsed);
    }
```

2 Time elapsed calculation

Time was calculated in milliseconds using a custom function *millis()* which utilizes *timespec* and *timespec_get* and returns current time in milliseconds. A variable *start* records the time upon execution, and a variable *end* records it after execution. Time elapsed is the difference between the two.

Listing 2: Getting time elapsed in ms

```
int64_t millis()
{
    struct timespec now;
    timespec_get(&now, TIME_UTC);
    return ((int64_t) now.tv_sec) * 1000 +
    ((int64_t) now.tv_nsec) / 1000000;
}

        int64_t start = millis();

<EXECUTE CODE>

        int64_t end = millis();

    double time_elapsed = (end - start);
```

3 Results

3.0.1 Sequential code

Array Size	Runtime (in ms)	Pi value	Pi difference
1000	0	3.1416035449	-0.0000108913
10000	1	3.1415929980	-0.0000003444
100000	1	3.1415926645	-0.0000000109
1000000	5	3.1415926539	-0.0000000003
10000000	44	3.1415926542	-0.0000000006
100000000	383	3.1415926483	0.0000000053

Table 1: Results when running the sequential code with different array sizes

3.0.2 Parallelized code

Array Size	Runtime (in ms)	Pi value	Pi difference
1000	0	3.1416035449	-0.0000108913
10000	0	3.1415929980	-0.0000003444
100000	0	3.1415926645	-0.0000000109
1000000	2	3.1415926539	-0.0000000003
10000000	19	3.1415926542	-0.0000000007
100000000	197	3.1415926475	0.0000000061

Table 2: Results when running the pthread-parallelized code with different array sizes

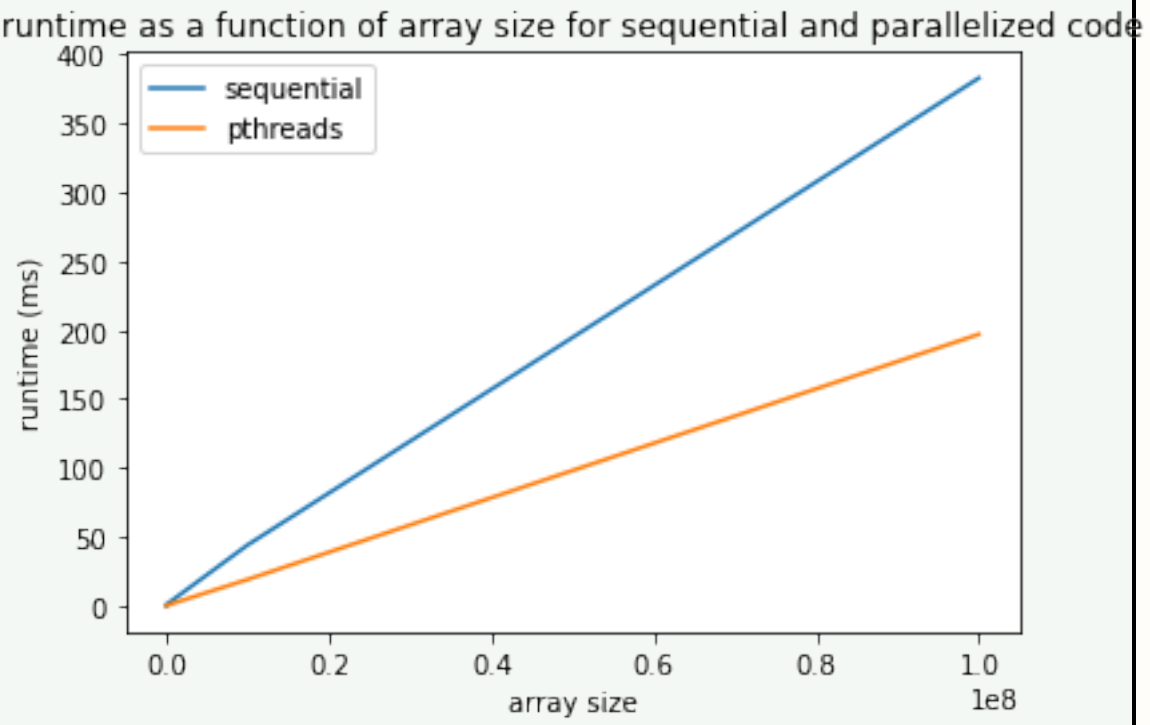
4 Plotting

The python library matplotlib was used for plotting in a jupyter notebook. Runtime was plotted against array size. The blue line represents the sequential code, the orange represents the parallelized code, as shown in the plot legend.

Listing 3: Plotting

```
import matplotlib.pyplot as plt

plt.xlabel('array size')
plt.ylabel('runtime (ms)')

#serial
plt.plot([1000, 10000, 100000, 1000000, 10000000, 100000000],
[0, 1, 1, 5, 44, 383], label="sequential")

#pthreads
plt.plot([1000, 10000, 100000, 1000000, 10000000, 100000000],
[0, 0, 0, 2, 19, 197], label="pthreads")

plt.title('runtime as a function of array size
for sequential and parallelized code')
# legend
plt.legend()
# Display figure
plt.show()
```

5 Discussion



As shown in the plot, obviously the parallelized code outperforms the sequential code, executing in less time for the same array size. (Array size is shown in a logarithmic scale). In general, as input size the time increases and the difference between computed and actual Pi value decreases. Accuracy increases for a larger input value. However, as shown in the tables, 10000000 seems to be the appropriate input size. For larger inputs, we see the calculated pi value drift away from the actual value, to the other direction.