Tharen Candi
Herman Thong

CS-307 Assignment 1

<u>Question 1</u>

Pi approximation:

A random number x and a random number y were generated in the domain [0,1] n times. Each randomly generated point is judged to be within the quarter circle if (x^2 + y^2 < 1). The ratio of points inside the circle to the total number of points is roughly equal to the ratio of the area of the inscribed quarter circle to the square. Therefore, pi can be approximated to 4* this ratio.

Integration:

A random number x within the bounds of [a,b] is generated, and its corresponding y value to the input function f(x) is computed n times. A sum, y_sum, of all f(x) values are maintained. The integral of f(x) within [a,b] is calculated to be y_sum * (b-a) / n. In other words, the integral is computed as the average area of all n rectangles that all have widths b-a and random heights on f(x).

a)

Pi approximation:

Generating the random numbers x and y, checking if it is under the quarter circle, and incrementing the count of the number of points inside the circle were parallelized.
After this parallel phase, there is a serial phase where all counts produced by the parallel phases are summed, and pi is calculated.

Integration:

Generating a random number x, calculating f(x) and summing f(x) was parallelized.
Afterwards, there is a serial phase where all f(x) sums produced by the parallel phases are summed, and a transformation is applied to obtain an approximation of the integral.

b)

Pi approximation:

Parallel phase:

- The operation that dominates the execution time is generating the random numbers.

Serial phase:

- The operation that dominates the execution time is division.

Integration:

Parallel phase:

- The dominant execution time is dependent on the input function that is being integrated. For large, complex functions, the arithmetic operations may dominate the random number generation. However, for smaller, simpler functions, the random number generation may take longer. Our timing results have been generated with f(x) = 3x^3 + 2x^2 + x. In this case, random number generation will dominate execution time.

Serial phase:

- The operation that dominates the execution time is division.

c)

For all programs:

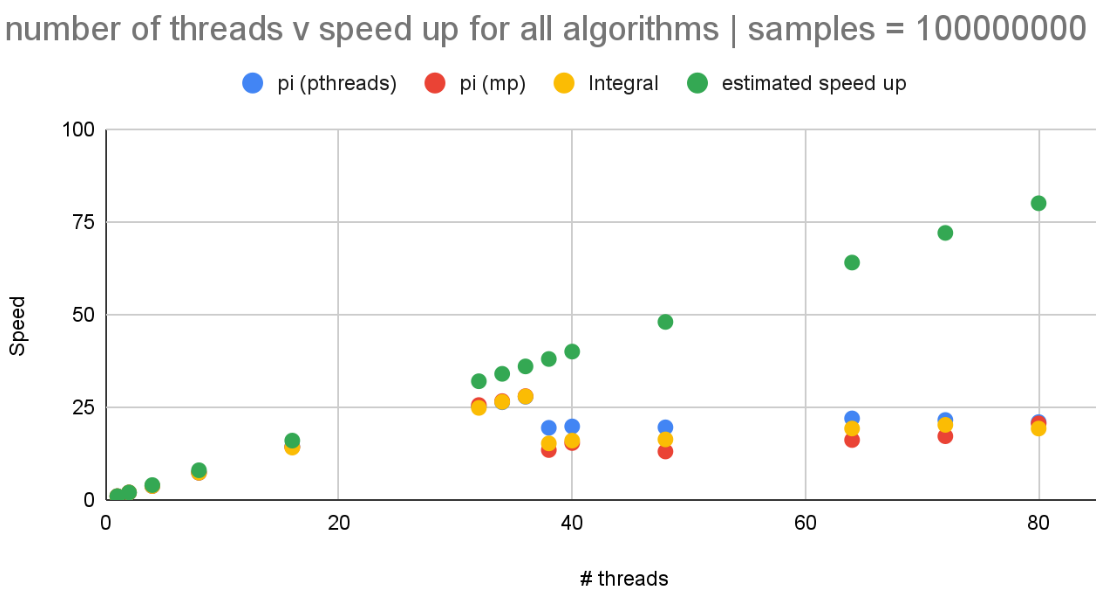
Without hardware limitations. Let N be the number of samples and M be the number of threads. The time complexity is O(N/M)

Tharen Candi
Herman Thong

d) We estimate the speedup of the multithreaded program to be slightly lower than the number of threads being used. This is because there is a degree of overhead in the creation and joining of threads. Therefore, a theoretical linear increase in speed up to the number of threads will be slightly lowered by these factors. The time of thread creation and joining will grow linearly with the number of threads.

However, we expect that the degree of overhead will be very minimal. Thus, our expected results are exactly a linear increase in the chart and graph in section 2.

Question 2



For integral: f(x) = 3x^3 + 2x^2 + x, a = 0, b = 5

Input and estimation			**PI - openMP**		**PI - pthreads**		**Integral**	
samples	threads	estimated speed up	time	actual speedup	time	actual speedup	time	actual speedup
100000000	1	1	1.182	1	1.62	1	0.88	1
100000000	2	2	0.5929	1.993590825	0.8147	1.988462011	0.441	1.995464853
100000000	4	4	0.3134	3.771537971	0.4299	3.768318214	0.233	3.776824034
100000000	8	8	0.1608	7.350746269	0.2206	7.343608341	0.1196	7.357859532
100000000	16	16	0.08287	14.26330397	0.1124	14.41281139	0.06231	14.12293372
100000000	32	32	0.04621	25.57887903	0.06412	25.26512789	0.03546	24.81669487
100000000	34	34	0.04437	26.63962137	0.06151	26.33718095	0.03328	26.44230769

100000000	36	36	0.04224	27.98295455	0.05823	27.82071097	0.03154	27.901078
100000000	38	38	0.08786	13.45322103	0.08331	19.44544472	0.05772	15.24601525
100000000	40	40	0.07701	15.34865602	0.08174	19.8189381	0.05501	15.99709144
100000000	48	48	0.09046	13.06654875	0.08278	19.56994443	0.05393	16.31744854
100000000	64	64	0.07316	16.1563696	0.0737	21.98100407	0.0457	19.25601751
100000000	72	72	0.06881	17.17773579	0.07517	21.55115073	0.04348	20.23919043
100000000	80	80	0.05731	20.62467283	0.07723	20.97630454	0.04572	19.24759405
100000000	88	88	0.07862	15.03434241	0.07993	20.26773427	0.04585	19.19302072

Question 3) From 1 thread to 36 threads, the speedups are roughly linear but slightly lower than the predicted values. The difference between the expected linear speedup and the recorded speedup is proportional to the number of threads. We can deduce that this is due to a growing overhead in thread creation and joining, as well as a possible slight overhead in the execution of a software thread on a physical core. This overhead is more significant than what we expected.

After 36 threads, the speedup is noticeably worse - on average, it immediately decreased a factor of 11.85 from 36 to 38 cores. Following this, the speedup is roughly constant (no increase).

Linear speedup is not expected when the number of threads created in the program outnumbers the number of cores that run in parallel on a processor. Since there is not a core for every thread, there is considerable time managing multiple thread executions on the same cores. Following this, some threads will have to run serially (after another thread) on one of the processor cores. This additional overhead explains the immediate drop in performance. The performance then stays consistent due to the consistent use of 36 cores of true parallelism.

The skylake Xeon processors within a cluster have 36 cores total, but support hyperthreading. Therefore, each cluster supports a total of 72 logical threads with hyperthreading or 36 without hyperthreading. We suspect that hyperthreading has been disabled, which would explain the decrease in performance after 36 threads.

Question 4) Compared to using OpenMP, pthreads has an average of 5% higher speedup from 1 to 36 threads. However, using pthreads requires substantially higher programming effort, as a structure has to be defined to pass information, and the threads have to be manually created and joined. Additionally, implementing parallelisation with pthreads requires an acute awareness of how threads will interact and share memory, and therefore increases the possibility of poor performance due to programmer or design error. For monte-carlo implementation, the 5% increase in performance is not worth the additional effort.