Report

1- Describe the algorithm that was implemented.

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 area of the inscribed quarter circle to the square.

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 is maintained. The integral of f(x) within [a.b] is calculated to be y_sum * (b-a) / n. This algorithm is a generalised case of the pi approximation aforementioned.

1a. 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.

1b. Pi approximation:

Parallel phase:
The operation that dominates the execution time is generating 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. Arithmetic operations may dominate the random number generation for large, complex functions. However, the random number generation may take longer for smaller, simpler functions.
Our timing results have been generated with f(x) = 3x^3 + 2x^2 + x.
In this case, random number generation will dominate execution time.

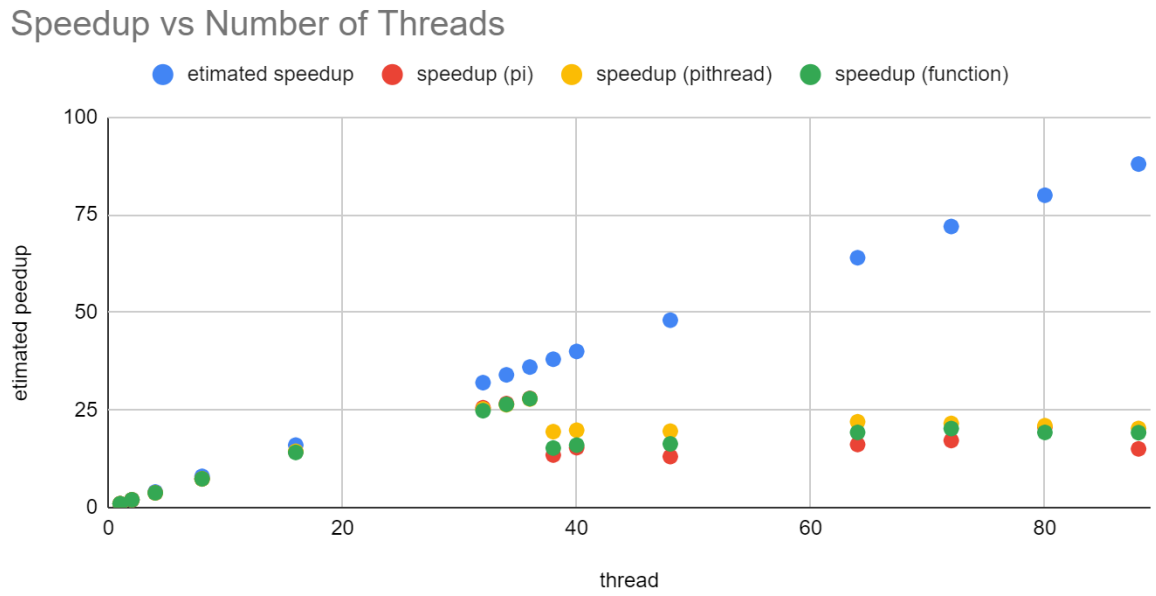
Serial phase:
The operation that dominates the execution time is division.

1c. 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)

1d. 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. Thread creation and joining will grow linearly with the number of threads.



2. Table

`function = 3x^3 + 2x^2 + x, a = 0, b = 5`

		Pi		pi_pthreads		integral		
thread	estimated speedup	time	speedup (pi)	time	speedup (pithread)	time	speedup (function)	calculated result
1	1	1.182	1	1.62	1	0.88	1	564.4627743
2	2	0.5929	1.993590825	0.8147	1.988462011	0.441	1.995464853	564.5014709
4	4	0.3134	3.771537971	0.4299	3.768318214	0.233	3.776824034	564.5510317
8	8	0.1608	7.350746269	0.2206	7.343608341	0.1196	7.357859532	564.6263309
16	16	0.08287	14.26330397	0.1124	14.41281139	0.06231	14.12293372	564.6693863
32	32	0.04621	25.57887903	0.06412	25.26512789	0.03546	24.81669487	564.5887435
34	34	0.04437	26.63962137	0.06151	26.33718095	0.03328	26.44230769	564.6002159
36	36	0.04224	27.98295455	0.05823	27.82071097	0.03154	27.901078	564.6398916
38	38	0.08786	13.45322103	0.08331	19.44544472	0.05772	15.24601525	564.625466
40	40	0.07701	15.34865602	0.08174	19.8189381	0.05501	15.99709144	564.5880903
48	48	0.09046	13.06654875	0.08278	19.56994443	0.05393	16.31744854	564.619203
64	64	0.07316	16.1563696	0.0737	21.98100407	0.0457	19.25601751	564.5782041

72	72	0.06881	17.17773579	0.07517	21.55115073	0.04348	20.23919043	564.5392128
80	80	0.05731	20.62467283	0.07723	20.97630454	0.04572	19.24759405	564.5219696
88	88	0.07862	15.03434241	0.07993	20.26773427	0.04585	19.19302072	564.5289897

3. From 1 thread to 36 threads, the speedups are roughly linear but slightly lower than the predicted values. This is because there is some overhead in creating and joining the threads, as well as slight overhead in the execution of a software thread on a physical core. After 36 threads, the speedup is roughly constant.

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. This is because some threads will have to run serially (after another thread) on one of the processor cores.

The skylake Xeon processors within a cluster have 36 cores, 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.

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.