Assignment 1 Group 11

1a. For both part 1 and 3, there are two sequential phases with one parallel phase in between. Monte-Carlo sampling is the part we parallelized in both programs, because it can perform millions or even billions of independent calculations. It takes the most amount of time and is very natural for parallelization.

First sequential phase is for initialization of variables and setting the number of threads for omp. In the final sequential phase, the result (pi in part 1 and integral in part 3) is computed and returned.

In the parallel phase, we first initialize a random number generator for each thread, because its underlying implementation is not thread-safe and sharing a random number generator between threads significantly affected performance for some reason.

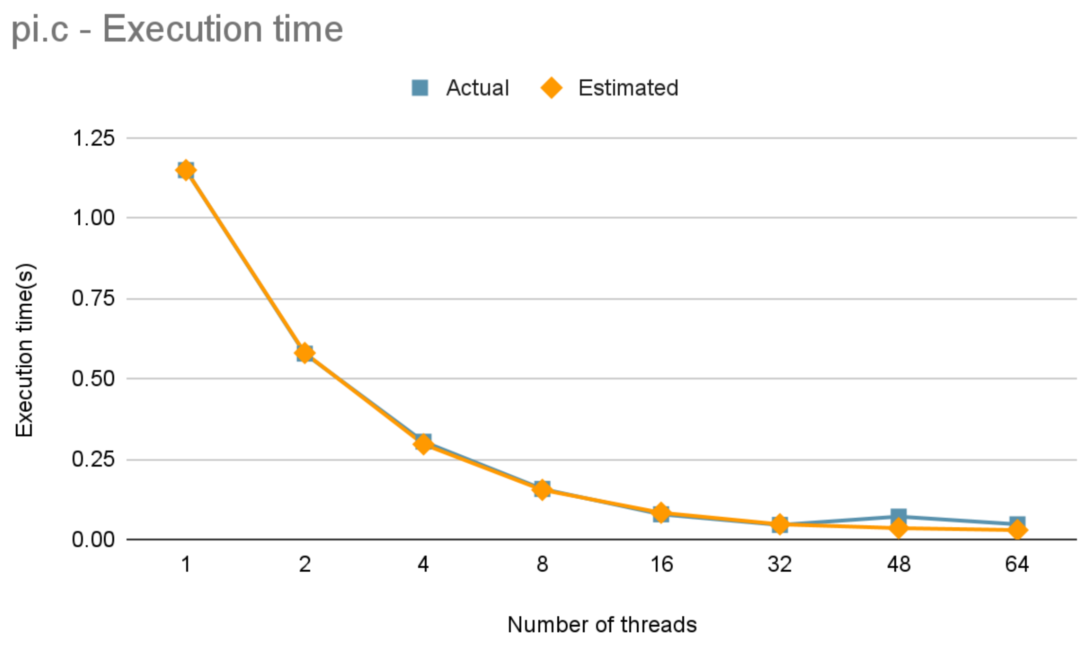
We then parallelize the sampling for loop. In part 1, we generate two random numbers as coordinates, and increment a counter if they fall in the circle. In part 3, we generate one random number and scale it to the given range. We then calculate the (estimated) integral and add to a sum.

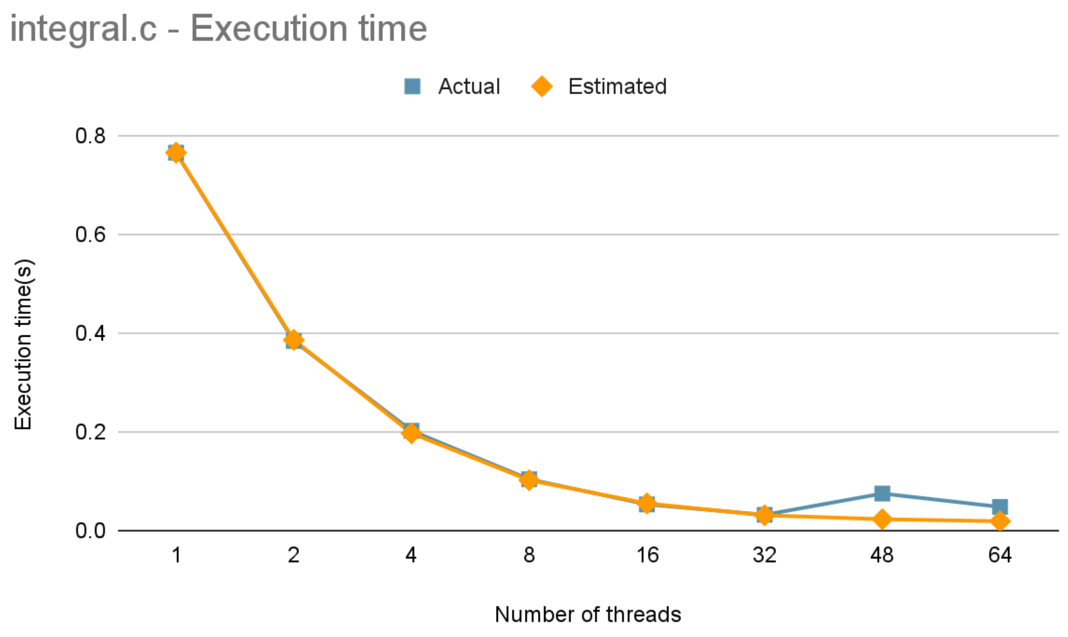
For the critical section in the parallel part (increment of counter in part 1, and sum of integral in part 3), we choose to use the omp reduction directive instead of the atomic directive for better performance. Reduction directive is OpenMP creating a private copy of the shared variable (counter and sum of integral) for each thread, and then adding them up when all threads finish their work. This also avoids the need to specify a omp barrier to sync threads.

1b. For the first sequential part, the function omp_set_num_threads() takes the most of time. In the second sequential part, division takes the most time. For the parallel part, the generation of random numbers is the biggest part.

1c. Both sequential phases are only executed once and are not affected by the program arguments. The execution time is therefore O(1). For the parallel phase, the execution time is proportional to the number of samples, but with work divided among threads. The execution time is thus O(num_samples / num_threads).

1d. Because we execute the program with 100 million samples, we assume this parallelizable phase to account for 99% of the execution time. We also assume zero overhead for parallelization for ease of calculation. Therefore the speedup factor is linear to the number of threads. We derive the following graphs using Amdahl's Law.





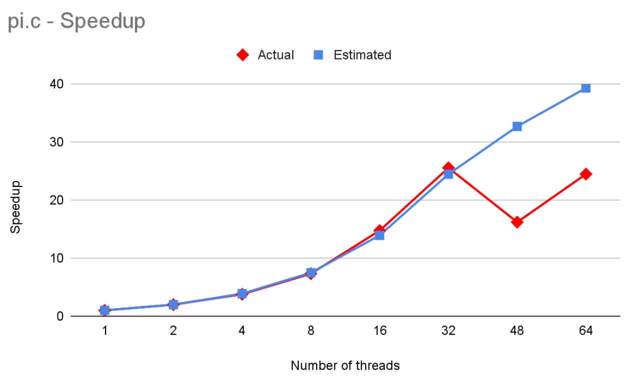
2. Both part 1 and part 3 are executed with all 36 cores of a SCITAS node, despite that changing the number of cores did not affect actual speedup for some reason.

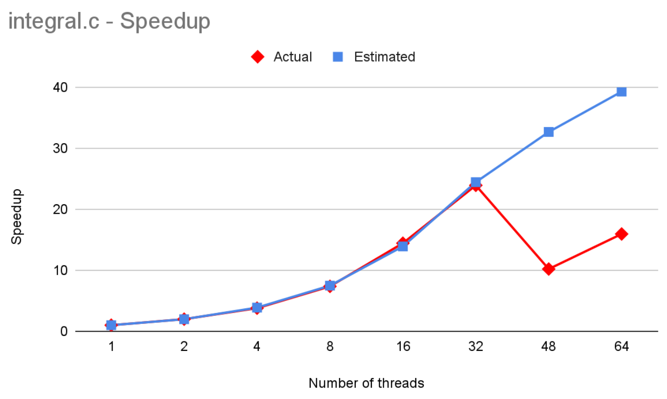
Part 1 pi.c:

num_threads	Expected time	Expected speed up	Actual time	Actual speed up
1	1.149	1	1.149	1
2	0.580	1.98	0.578	1.99
4	0.296	3.88	0.304	3.78
8	0.154	7.48	0.157	7.32
16	0.083	13.91	0.078	14.73
32	0.047	24.43	0.045	25.53
48	0.035	32.65	0.071	16.18
64	0.029	39.26	0.047	24.45

Part3 integral.c:

num_threads	Expected time	Expected speed up	Actual time	Actual speed up
1	0.765	1	0.765	1
2	0.386	1.98	0.384	1.99
4	0.197	3.88	0.202	3.79
8	0.102	7.48	0.104	7.36
16	0.055	13.91	0.053	14.43
32	0.031	24.43	0.032	23.91
48	0.023	32.65	0.075	10.20
64	0.019	39.26	0.048	15.94





3. After comparison, we conclude that in both part 1 and part 3, the actual speedup is perfectly in line with our estimation when we increase the number of threads from 1 to 32. But when thread count goes above 32, the speedup does not keep increasing and is even lower sometimes. This is likely due to hardware limitations

because one SCITAS node has 36 cores only and more threads may not be fully parallelized. It can also be due to the parallelization overhead of threads and also that communication costs between threads grows with the number of threads.

4. Using pthreads demands significantly more programming efforts and is also more error-prone, but however leads to worse performance than OpenMP (max speedup is only around 3x). Pthreads is also less flexible than OpenMP in some regard, requiring a routine with fixed signature (void pointer as both parameter and return type).

It might make sense to use pthreads, however, when it requires finer or more flexible control of the parallel phase, or more complicated inter-thread communication and synchronization.