A1

Jonas B, Alexander M

October 2022

Part 1: Program Structure

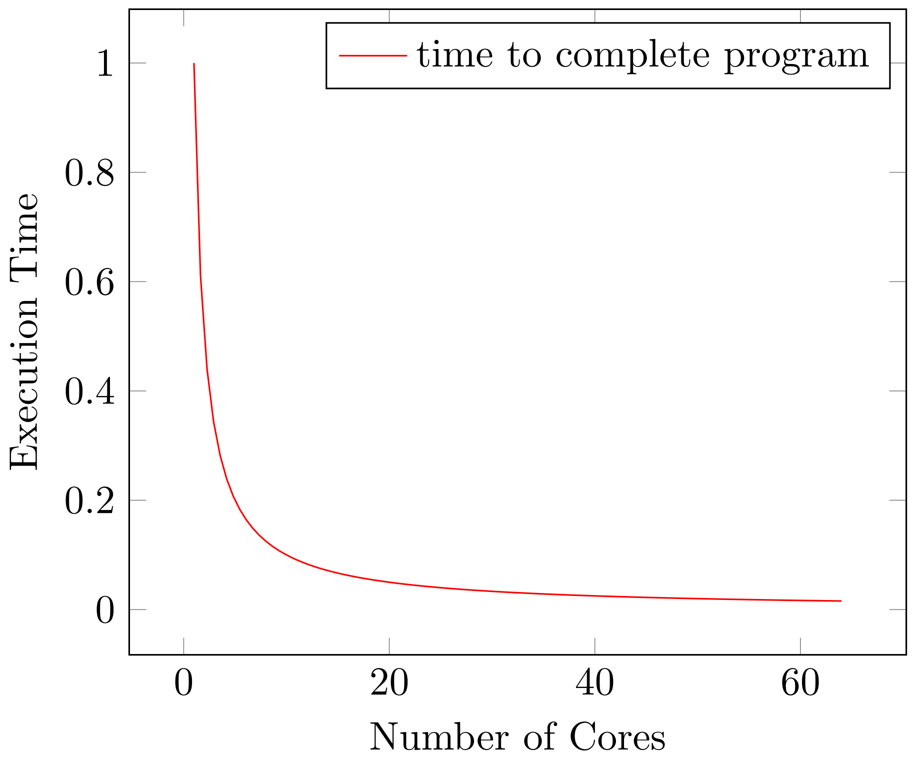
Calculating Pi

a. For this program, we parallelized the process of picking random points on a square and checking if they were inside or outside the unit circle. Then, serially, we added together all of the results, and performed the necessary calculation to get our approximation of pi.

b. The operation we spend the most time on is random number generation.

c. For calculating pi, we depend on the number of samples linearly, O(n), but can split up the work over the number of threads, so we get O(n/p).

d.



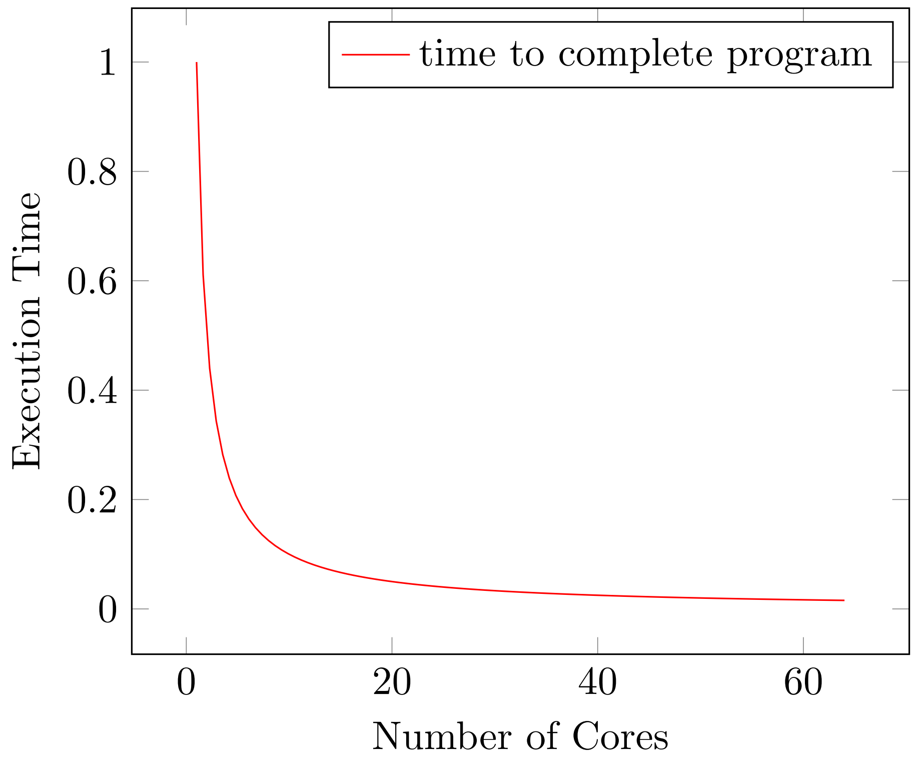
Monte Carlo Integrals

a. This section is very similar to the section for pi. Now, we pick points on regular intervals, and we evaluate them on the provided function and sum up all of our results. This part can be parallelized by assigning equal sub intervals to each core (as integrals are linear, $\int_a^b f(x)dx + \int_b^c f(x)dx = \int_a^c f(x)dx$). Then we sum the results of the parallel components serially, and take the the average over the number of samples.

b. The operation we spend the most time depends on the provided function. If it is complicated to compute, it could easily take the most time. If it's simple to compute, the random number generation would take longer.

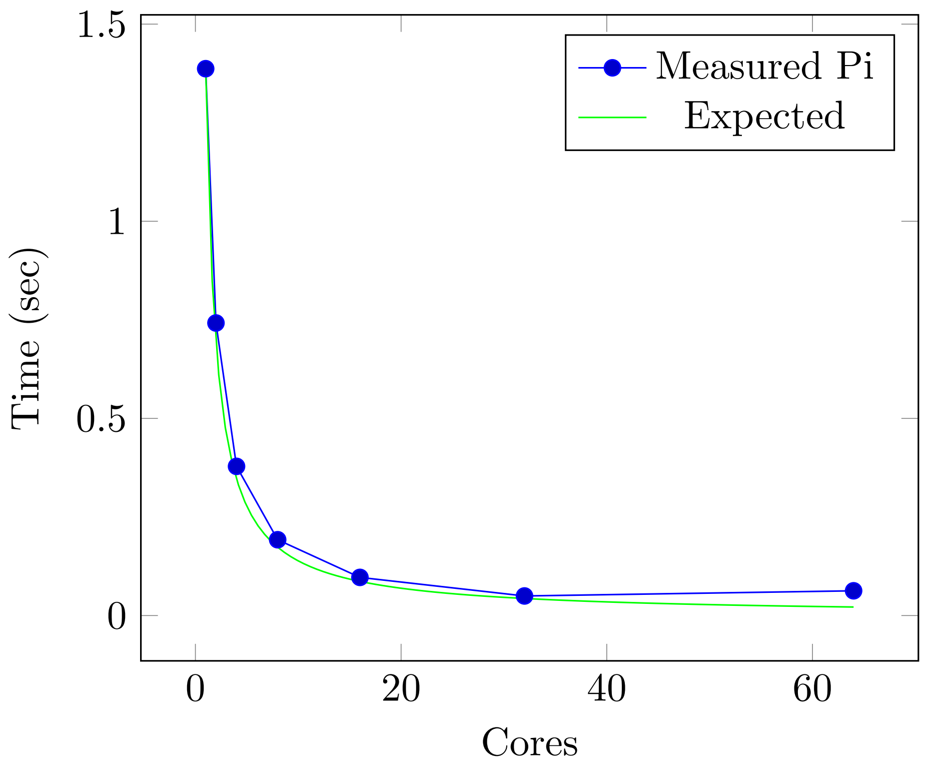
c. Our running time on a single thread depends solely on the number of samples, O(n). By parallelizing, we can split this work up over p threads, for an O(n/p) runtime.

d.



Part 2: Real-World Speedup

No. Cores	Pi (sec)	Monte Carlo (sec)
1	1.39	1.35
2	0.74	0.74
4	0.38	0.32
8	0.19	0.19
16	$9.68 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$
32	$4.97 \cdot 10^{-2}$	$5.1 \cdot 10^{-2}$
64	$6.28 \cdot 10^{-2}$	$5.2 \cdot 10^{-2}$



Part 3: Analysis

We see that our prediction holds very accurately up until 64 threads. The reason that it stops holding at this point is because the machine only has ∼40 cores, and so the 20 some threads that are defined are still run sequentially, and we in fact gain time due to having to allocate these threads.

Part 4: OpenMP vs pthreads

While OpenMP is certainly less code to write, and very practical to understand, the power of pthreads cannot be denied. It gives the programmer much more control over individual threads, even if it is more cumbersome to write. When something complicated has to be done, where threads can start new tasks when they finish their old ones, or jobs are not easily partitioned into equal work loads, pthreads is an obvious choice.