SHPC4001 - Assignment 8

Ayden S. McCann

May 2021

Code Listing at the end

Introduction

Python is a very dynamic language, but when compared with compiled languages such as C in many cases it is significantly slower. The fact Python is an interpreted language means that CPU time is dedicated towards this interpretation. One way around this which has been explored in this assignment is the application of Cython. Cython is a compiler which effectively translates Python code into C code. As a result, speed ups can be observed with minimal to no alteration to the code. Another advantage of Cython is the ability to introduce multi-thread processes. In this assignment that has been explored through the creation of a Monte Carlo integration to estimate the volume of an N-dimensional unit sphere.

1 Part 1

1.1

The CyQuantumWalk module was altered from the original PyQuantumWalk in a few ways. The primary change was the introduction of cdef on a number of variables, and all functions within the module with the exception of QuantumWalk. The datatype of various input parameters was also introduced (int, list, double etc) and the H array (which was bright yellow on the Cython html report) was changed into C-doubles to reduce python interaction when it is called. The Cythonized module runs substantially faster than the original python module:

average original time: 0.41558630418777465
average new time: 0.1743643352985382
138.34 percent average improvement after 1000 iterations

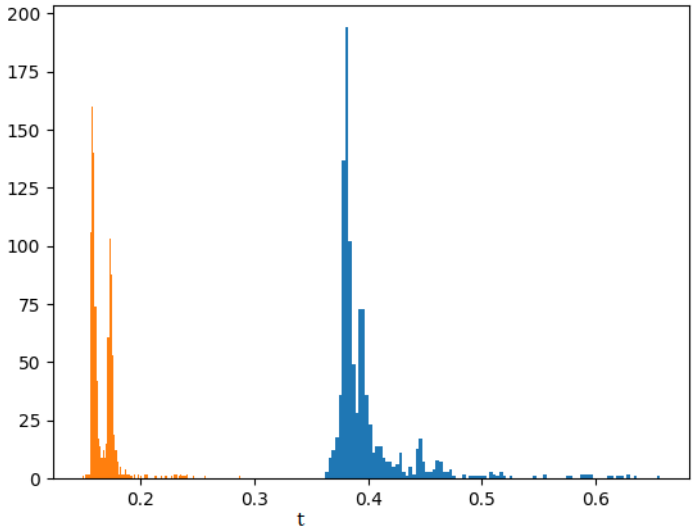

Figure 1: The distribution of computation times over 1000 iterations for the original module (blue) and the Cythonized module (orange).

However it is worth noting that at an earlier stage, when i was only achieving an approximately 5% increase in performance, I gave my code to a peer to test on his Macbook Pro and it reported an approx 610% percent speed up, over 100 times more than the improvement on my Windows laptop. I therefore believe that running the newest version of my Cython module on such a computer again will demonstrate far superior results to those I am able to report currently.

1.2

I believe that the PyQuantumWalk could be implemented in parallel. Sections of the code only depend on inputs from the Ith and Jth inputs from lists constructed in other functions, such as the ApplyCoinOperator function. It would be therefore possible to construct the module in such a way that the ApplyCoinOperator for example could be distributed over a number of threads. Whether or not this implementation would increase performance such that the computational overhead of distributing and managing said thread is overcome would have to be explored.

2

2.1

As detailed in Figure 2 my code has implemented parallelism in the generation of random numbers [1]. For a given number of points, iterations to average over per number of points and dimensions it needs to calculate:

$$N_{rand} = Iterations * Dimensions * (Points)$$

Given this large number of random points which all are calculated independently i decided that distributing this task over multiple threads could be a good implementation of parallelism.

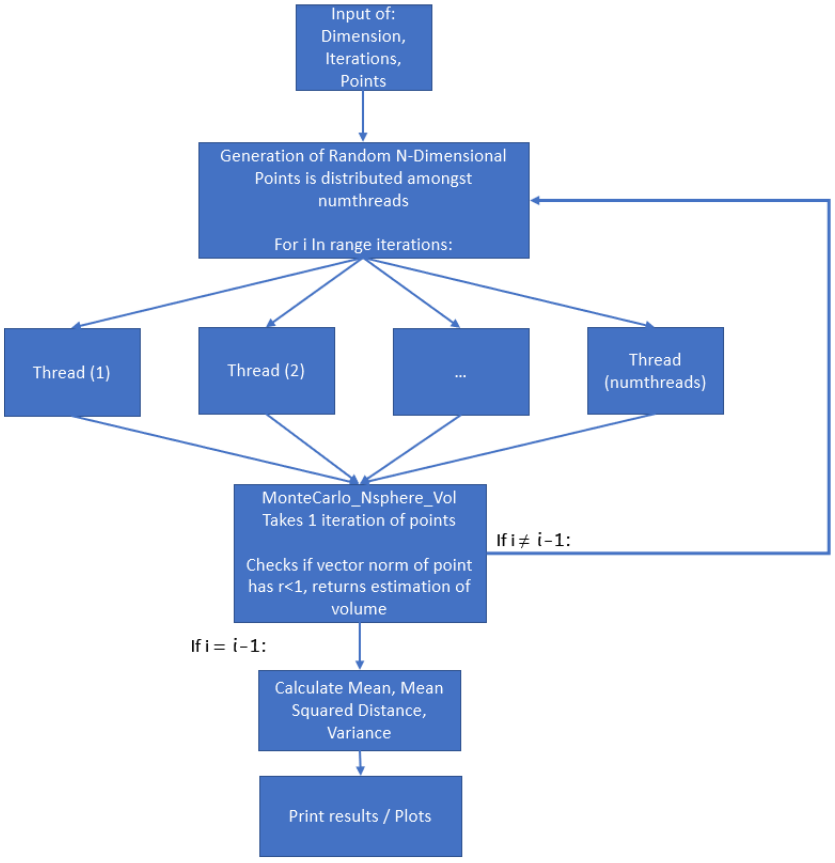

Figure 2: The flowchart outlining how parallelism was implemented in my algorithm.

However it is clear in Figure 3 that the overhead for managing and distributing tasks to these threads is computationally more expensive than any benefit gained. As a result, performance decreases as the number of threads increases.

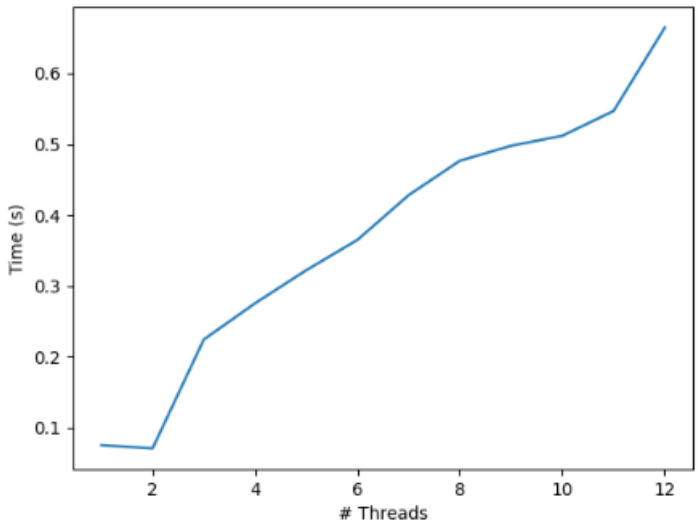

Figure 3: An example of the performance of the system for calculating the volume of a 10-Dimensional Sphere for a number of utilised threads.

As a result, my calculations of variance and accuracy have been performed using a single threaded version, where the random number generation occurs inside the MonteCarlo function (Part2SingleThread.pyx). This code was modified to iterate over an increasing number of points and plot the variance and volume.

The following plots have been created using: 2 dimensions, 2 iterations and 10000 maximum points.

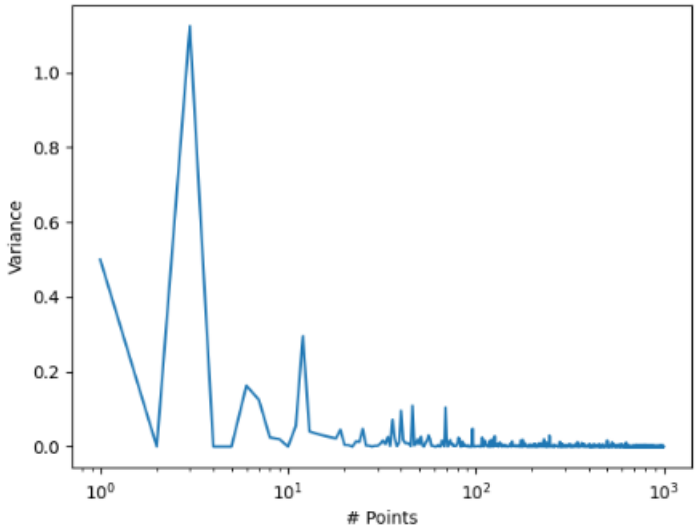

Figure 4: The convergence of variance for the system described above

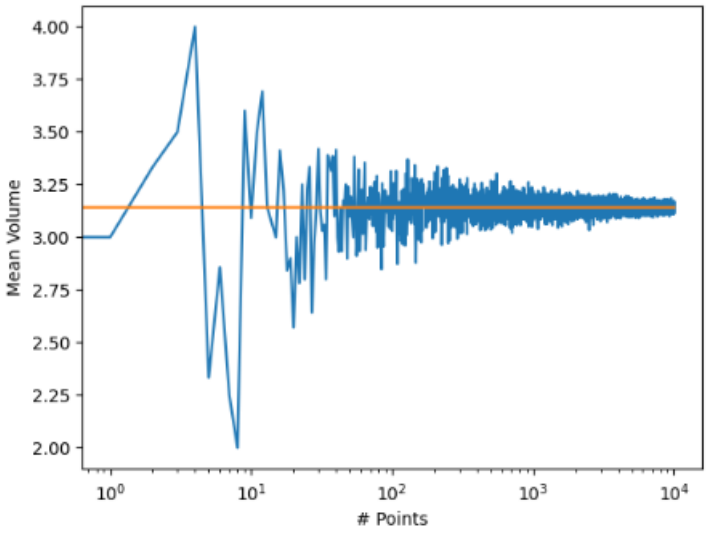

Figure 5: The convergence of volume for the system described above (actual volume printed in orange).

Given that conventional Monte Carlo techniques utilise a significantly larger number of points, the accuracy of this code is substantially lower than other implementations. This however was done purposefully such that I could plot the progression of the mean and variance with respect to increasing points whilst not taking a long amount of time to do so. If i had chosen for example 100,000 points or higher, it would take a long amount of time (15+ minutes). Modifying this code to only perform the volume estimation for a single number of points would be more appropriate for the sole purpose of obtaining an estimate but information regarding its progression etc would be lost.

Conclusion

As shown in Part 1 Cython is an effective method for increasing the performance of Python code. In order to observe increases in performance only minimal alterations are required, more comprehensive implementations can exhibit vastly greater increases in performace. Although the application of multi-thread processing in Part 2 did not result in an increase in performance it was a valuable exercise in understanding the way in which these algorithms must be written and carefully designed such that increases in performance outweigh any overheads associated with introducing multiple threads.

3 Code Listing

Question 1:
CyQuantumWalk.pyx

Question 2:
N/A

Question 3:
Part2MultiThread.pyx
Part2SingleThread.pyx

References

[1] Matwiejew E, 2021, Edric-Matwiejew/Cython. `https://github.com/Edric-Matwiejew/Cython/blob/main/examples/3_openmp/cdef_random_numbers.pyx`