Determining the number of "cores" are on a machine

Submitted by: Md Ishtiaq Ahmed

Abstract:

In this experiment, we aim to determine the number of CPU cores present in three different computers running with different operating systems. In each machine three different programming language C, Python and Java are used. By using multiple thread to execute the programs, we observed how runtime varies with the increasing thread count as well as how it varies in different programming language and in different operating systems. In ideal scenario, if the system is using the cores equitably, we should have similar runtime until we create more than N thread where N is the reported number of cores.

1. Introduction:

Multi-core processors can handle computational tasks efficiently and allow multithreading. If the system efficiently utilizes its CPU cores, when the number of threads is less than or equal to the reported number of CPU cores, each thread can be assigned to a separate core, enabling parallel execution of tasks. So, the execution time will remain almost the same. But when the thread count is higher than the number of cores, then the system will start time sharing or other scheduling mechanisms to accommodate the additional threads which will result to higher execution time. By using this principle, we will try to determine the number of cores in a machine.

2. Methodology:

Firstly, the number of CPU core can be determined by checking the system configuration of a computer or by checking the manufacturer website.

Secondly, as per the assignments requirement, a program is designed to run using multiple threads. The program will run 5 times using one thread first, then two, then three and continue up to N thread. Average execution time will be calculated along with the standard deviation. Later the number of core will be determined by analyzing the execution time for different number of thread.

3. Experiment Design:

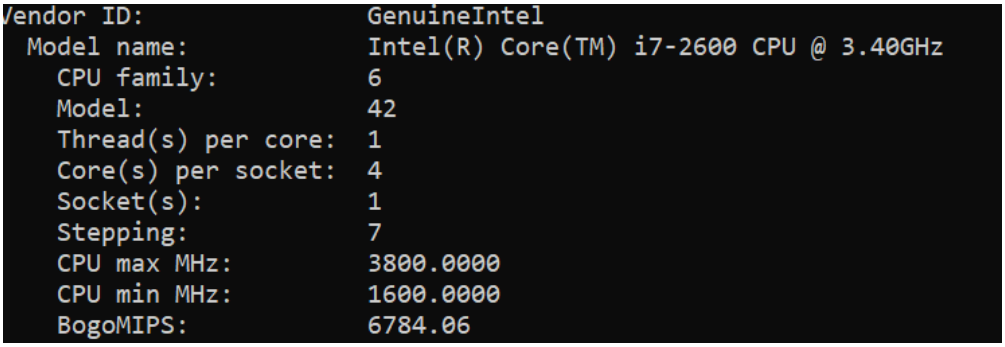
The experiment involves spawning multiple threads, each performing a similar computational task across three programming languages: C, Python, and Java in three different machines. Then the runtime for executing the program using different thread counts is measured and analyzed to find the number of CPU cores.

Machine used in the experiment:

1. Acer Nitro 5

Processor: 12th Gen Intel® Core™ i7-12650
Operating System: Windows 11 Home, 64 bit
Number of Cores[1]: 10
Number of Thread[1]: 16

2. MacBook pro
   Processor: M1
   Operating System: OSX
   Number of Cores[2]: 8
3. Poobah.nmsu.edu
   Processor: Intel® Core™ i7-2600 CPU @ 3.40GHz
   Operating System: Linux
   Number of Cores: 4



Programming language used in the experiment:

C, Python and Java programs are used in this experiment. All the program consists of a function that will fill up an array multiple number of times. The size of the array and number of times the array to be filled is different in different languages. For C, the array size is 10000 and it is filled up 10000 time. For python, array size is 10000 and it is filled up 100 times. For Java, the array size is 100000 and it is filled up 10000 times. It is intentionally made differently so that in each program it takes a reasonable amount of time for execution. Then the function will run through multiple threads of the machine and will return the execution time.

4. Experimental Procedure:

Each of the programs runs on three different machines by spawning through a number of threads, starting from one up to four times the reported number of CPU cores of individual machine.

Then the execution times and standard deviations are collected and processed in Microsoft excel. Graph is plotted for the execution time and standard deviation as error bar.

After analyzing the plots a decision is made that how many cores that particular machine has and verify that with the actual number of cores of that machine,

5. Results:

Below are the results of each program in different machines:

Table 1: Multithread Execution result in Poobah(Linux)

Language		C		Python		Java	
Machine	Number of Threads	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation
Poobah (Linux)	1	198618600	2901673.56	36326503.75	2717178.247	320761247.2	4028784.842
	2	202566600	854018.64	69043970.11	514729.0812	325997141.2	182836.3306
	3	207573400	90685.39	105207157.1	3631448.364	335042926.2	156103.5916
	4	212863000	424503.47	138485336.3	2680373.489	343956540.4	1439356.451
	5	316038000	1529491.03	173026084.9	4256475.114	483028595.6	26697716.61
	6	328068400	2675450.29	205708646.8	1407072.048	571663572.2	12689236.65
	7	385672200	5250382.71	242371130	3208143.237	625636255.4	7143474
	8	435999000	3791922.63	273957443.2	4118130.658	695820457.8	4675731.264
	9	488777800	8262996.78	309294223.8	4479190.839	791135183.6	14148297.08
	10	539985800	3836666.13	345883417.1	5623220.399	874898254.6	7005641.764
	11	593530800	4535018.96	379404544.8	1029858.843	955257021.4	3819533.458
	12	643311600	2635877.74	413944912	10132799.05	1031947780	3501894.382
	13	698114000	4693116.87	453427362.4	9888809.927	1125593109	1187128.856
	14	749754600	2054387.66	482338190.1	2848515.43	1212413049	6063633.482
	15	804802000	1582032.74	514067506.8	1725146.843	1295837787	6822080.984
	16	853680800	1695005.06	555628681.2	12031827.74	1376052678	6139997.462
	17	908632600	2146378.96	584926843.6	5405497.978	1466873306	2362711.772
	18	960822200	2891391.18	628945684.4	16697263.45	1554040044	3170547.548
	19	1015977800	2789842.03	653127861	3986809.642	1638069838	2765029.091
	20	1064381200	704143.28	690870380.4	8334812.664	1721626215	1979405.147

Table 2: Multithread Execution result in OSX(mac)

Language		C		Python		Java	
Machine	Number of Threads	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation
MacBook (OSX)	1	98729600	9339965.15	18014621.73	5830557.036	3.19E+08	1.99E+08
	2	96473400	364813.71	26674175.26	247919.8779	1.60E+08	155100.3413
	3	96692000	398317.96	40132904.05	961168.432	1.65E+08	478351.8782
	4	98020800	275766.86	52638006.21	118931.2554	1.66E+08	1156981.145
	5	99069600	550733.91	65500783.92	285456.4649	1.66E+08	835017.8232
	6	101571200	611458.39	78859901.43	458045.0807	1.73E+08	5542943.273
	7	127480200	3738616.5	91537904.74	200661.9487	1.93E+08	3815879.829
	8	136936000	5556811.57	104115390.8	133739.0738	2.14E+08	1133848.556
	9	158665000	9139629.18	116962194.4	559874.9791	2.43E+08	6408086.831
	10	170624800	5824398.15	130162954.3	621665.0726	2.70E+08	1.08E+07
	11	182803000	2624403.93	143713331.2	929501.496	3.13E+08	1.80E+07
	12	195661000	1791730.89	156007576	910179.0684	3.38E+08	8983106.69
	13	221002600	3209576.96	169484376.9	1068063.811	3.77E+08	2.41E+07
	14	233136400	7763369.1	182298803.3	1075051.483	3.90E+08	7734620.949
	15	252850400	10202121.99	194644641.9	684478.9254	4.23E+08	9480748.217

	16	279700000	14472589.84	207145404.8	461228.2561	4.58E+08	2.00E+07
	17	305762600	10557663.09	220183181.8	921630.2531	4.59E+08	1.65E+07
	18	307346000	13769049.6	233839845.7	1197326.243	4.80E+08	1.83E+07
	19	319816200	6140603.73	247099161.1	2008306.111	5.03E+08	2921527.116
	20	327317000	7982382.93	260147476.2	2085736.297	5.37E+08	6465902.708
	21	355939600	10199449.96	271505165.1	883116.7291	5.47E+08	4635588.104
	22	365553600	7973327.8	285506391.5	954385.4268	5.77E+08	1.05E+07
	23	379215000	6250274.17	297908544.5	1019778.165	6.05E+08	1.18E+07
	24	397105400	6149898.46	310610866.5	419712.821	6.27E+08	5753639.713
	25	417096400	9128018.72	324439573.3	1792944.138	6.50E+08	9219532.249
	26	427810600	2728678.77	335991144.2	598063.6081	6.76E+08	3468502.148
	27	445214800	3858223.39	349032163.6	341024.5464	7.01E+08	5726754.726
	28	460151400	7325049.29	362892961.5	905401.0537	7.32E+08	8224828.905
	29	466879600	4596567.66	374433803.6	871442.8252	7.53E+08	7179123.212
	30	490586000	5506146.13	387953519.8	200058.9036	7.78E+08	2836658.016
	31	504821400	7545194.01	401297807.7	1266571.039	8.20E+08	1.55E+07
	32	523911800	9011715.61	413529825.2	562549.8474	8.46E+08	1.71E+07

Table 3: Multithread Execution result in Windows

Language		C		Python		Java	
Machine	Number of Threads	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation	Average Time (nano seconds)	Standard Deviation
Windows	1	69529400	2105930.07	16237115.86	2304973.613	6.04E+07	5377976.656
	2	68800000	748064.44	31218051.91	848207.076	5.75E+07	133595.5628
	3	70594000	507979.92	46891689.3	1050818.507	5.83E+07	201333.3614
	4	73945800	139308.87	62726306.92	996959.617	6.27E+07	2153749.481
	5	76184600	997551.12	76768207.55	1389366.287	6.40E+07	2806704.454
	6	83403800	5214582.07	94602489.47	3479376.679	7.32E+07	2899883.263
	7	82399800	1356230.13	108113002.8	776201.1043	8.73E+07	7830038.269
	8	85600000	1743789.09	124080514.9	1961306.671	1.02E+08	4281984.632
	9	92285400	4617596.72	139969635	1783290.029	1.12E+08	5173160.042
	10	103807200	9010476.16	154856634.1	955186.5301	1.22E+08	9330133.253
	11	120548400	2959617.31	170817804.3	3336974.428	1.34E+08	1.04E+07
	12	115204800	4163427.55	187771177.3	4020312.761	1.44E+08	2487986.674
	13	130603800	4310905.54	200249099.7	2061144.726	1.72E+08	1.93E+07
	14	150257200	11097036.03	220513582.2	9179203.691	1.83E+08	6718391.038
	15	137402800	4177545.57	232613945	1957905.584	1.92E+08	1.18E+07
	16	146277800	6887502.52	250587940.2	5878328.696	2.08E+08	1.19E+07
	17	161008200	15438435.15	262708902.4	1398147.93	2.07E+08	4703141.839
	18	159914400	5914271.61	280860996.2	1798073.635	2.21E+08	7275924.213
	19	197237400	22117545.89	295586919.8	1957954.214	2.41E+08	9827714.688
	20	181563400	9045303.15	312124109.3	2475750.33	2.28E+08	2558221.159
	21	204419200	8953407.61	327203226.1	1641099.969	237271140	4635710.379
	22	198877800	7017384.65	341850137.7	1659073.295	250170640	13173033.78
	23	234016600	5963882.38	358400058.7	3098672.838	258944000	5380366.079
	24	232582200	9323827.61	373332262	2590550.838	283591260	11261519.29

	25	254618000	12106535.67	389379024.5	6009109.757	283345920	7725805.054
	26	255054800	6888370.18	400412941	2131447.827	299438240	9524742.467
	27	252028000	8767434.74	412527179.7	1113926.233	305978140	13725290.59
	28	261185600	8195673.31	428052759.2	1582546.019	312648240	6340305.763
	29	272244000	16311034.68	443332290.6	1863573.431	336146420	12660454.44
	30	277607800	4193890.19	460501384.7	969281.2754	337392120	4980487.341
	31	284007000	9438012.16	474955749.5	2696306.909	354371200	10321312.09
	32	293125600	5060244.8	492675590.5	1964421.119	360682420	8894273.045
	33	297415800	4940684.71	510048103.3	3333207.692	377638580	7159476.9
	34	305198600	4483645.46	528714084.6	4683946.805	373249840	3256278.234
	35	314182200	4729141.04	540404415.1	2061679.309	397674340	8677416.346
	36	325400800	6276807.19	556137275.7	5775625.93	403341140	13078174.9
	37	317506200	6993300.38	610839748.4	43153076.26	411370120	6316806.732
	38	326847200	5351268.24	596998310.1	23253055.31	419169820	7693956.355
	39	340027800	4040089.77	603423070.9	4586153.804	425240740	4107120.505
	40	345485600	7612253.19	625516462.3	2113491.711	439367760	3172956.508

6. Analysis:

The results demonstrate how the runtime varies with the number of threads spawned, indicating the efficiency of core utilization. By analyzing the average runtime and standard deviation, we can get an idea that how many cores a CPU has.

Different performance was observed for different language in different machines.

** In all the plots, the error bars represent one standard deviation from the mean. This indicates the variability or spread of the data points around their average value.

6.1. Performance on C program

C programs are usually faster and more efficient[3]. C allows better control over the system which leads to a better utilization of CPU cores.

Figure 1 shows the execution time of implementing the C program using multithread in poobah. We can see a sharp increase in execution time after 4 threads. Poobah is running with Linux operating system and the system has 4 cores. Here, up to 4 threads, execution time almost similar then the time is linearly increasing with number of threads.

Figure 2 shows the performance in a mac OS. The trend is like Linux. The mac machine has 8 cores. Up to 6 threads, execution time is similar and after that it starts to increase in linear fashion. Probable reason that other two core was busy with other task and that's why time start to increase after 6 threads.

But in windows the trend seems different (Fig 3). The time started to increase from the beginning in a linear fashion. Although, up to 8-9 thread the increment in execution time is little but after that execution time increased more. To analyze why this is happening, I looked at the thread utilization from system tool. This windows machine has 10 cores but 16 thread. When the program

is not running, 6 to 7 thread was already occupied with other programs (Fig 4). When the program was running with higher number of thread, all 16 thread of the machine start to run with 100% utilization (Fig 5). That implies that the multithreading program is working correctly and utilizing all the cores of the machine.

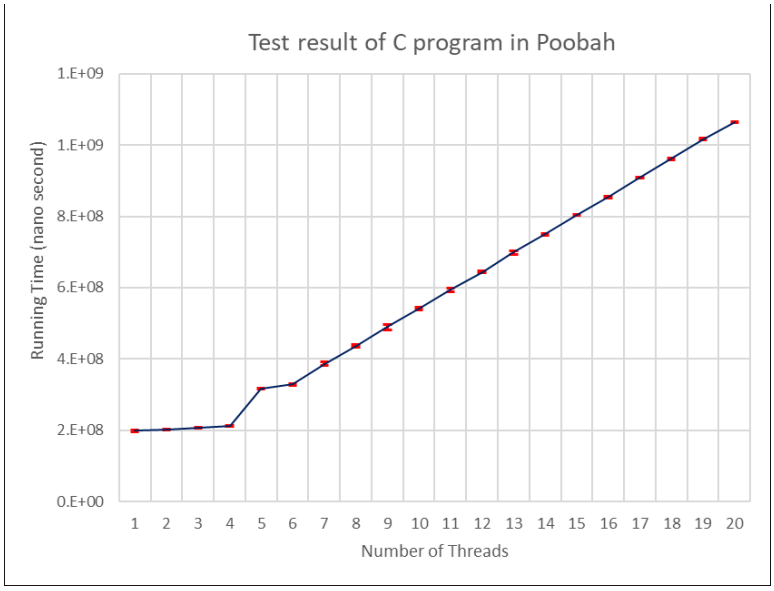


Figure 1: C multi-thread result in Linux

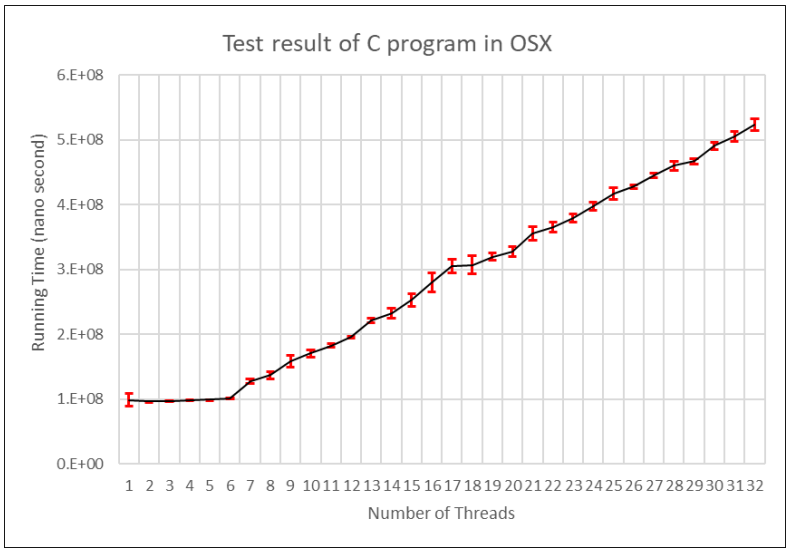


Figure 2: C multi-thread result in OSX(mac)

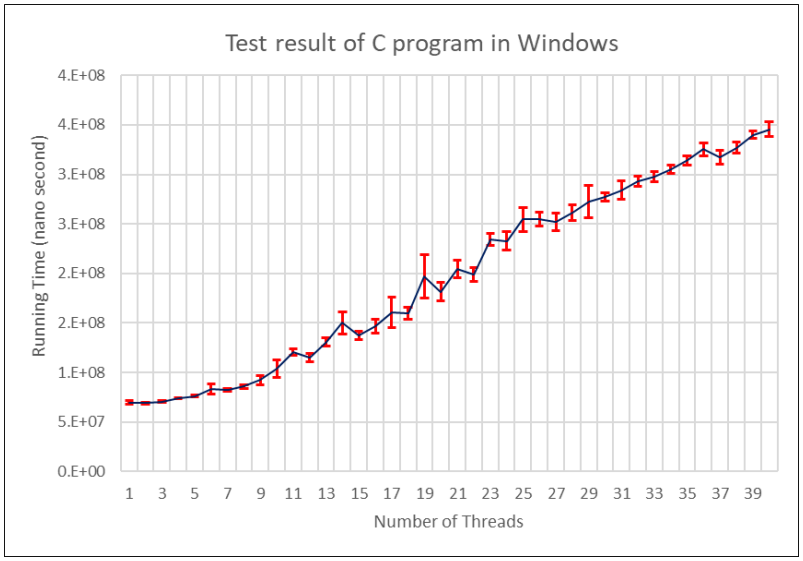


Figure 3: C multi-thread result in Windows

Standard deviation is also higher in windows compared to linux and mac OS.

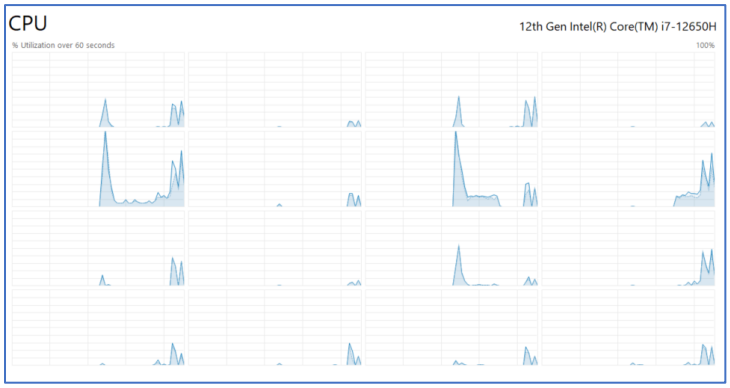


Figure 4: Thread Utilization in Windows When Program is not running

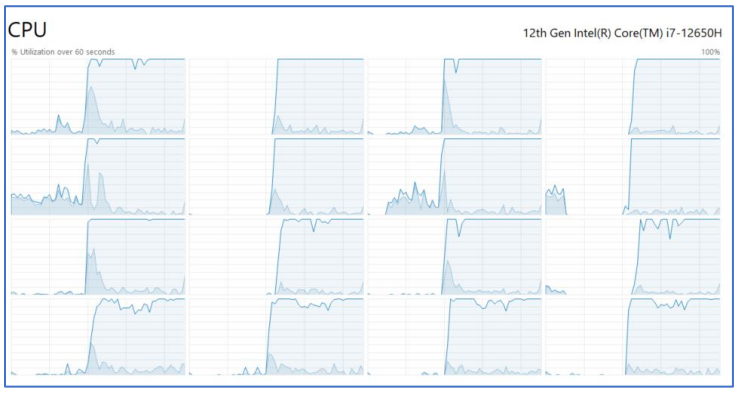


Figure 5: Thread Utilization in Windows When Program is running

6.2. Performance on Python Program

The result of running Python program using multithread in all three operating system is straight forward. Python is not a thread-safe interpreter, that means the interpreter can execute only one thread at given time. This limitation is enforced by the Python Global Interpreter Lock (GIL), which essentially limits one Python thread to run at a time[4]. The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter[5]. Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all using multithreading in python[6]. The result from the experiment also support this statement. We can see from the below plots; the execution time is linear with the number of threads. It shows similar trend in Linux, Windows and OSX operating systems, implying that using python it is difficult to determine the number of cores in CPU.

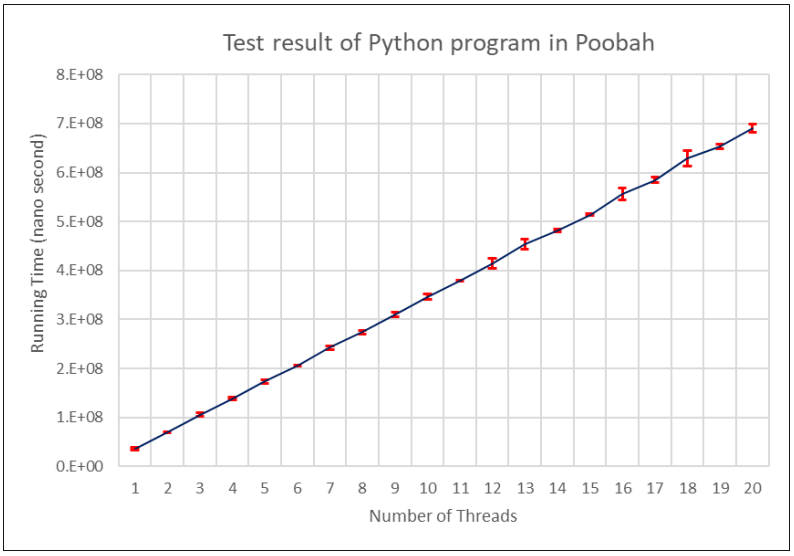


Figure 6: Python multi-thread result in Linux

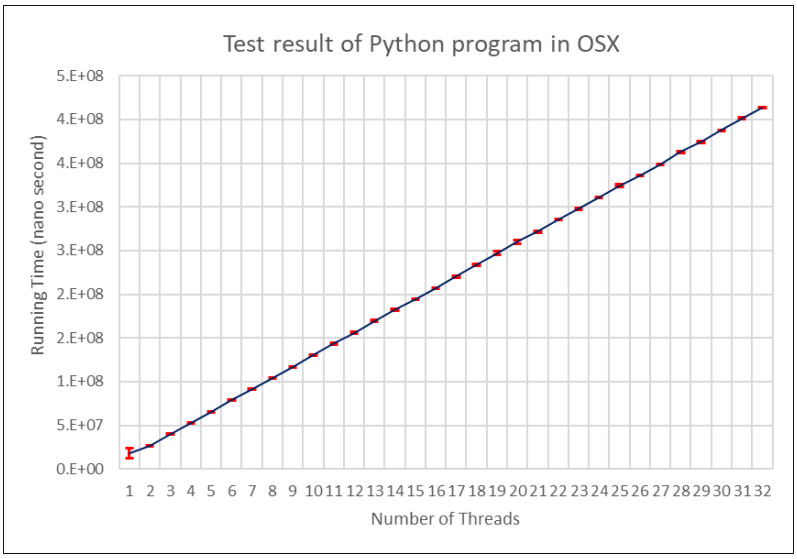


Figure 7: Python multi-thread result in OSX(mac)

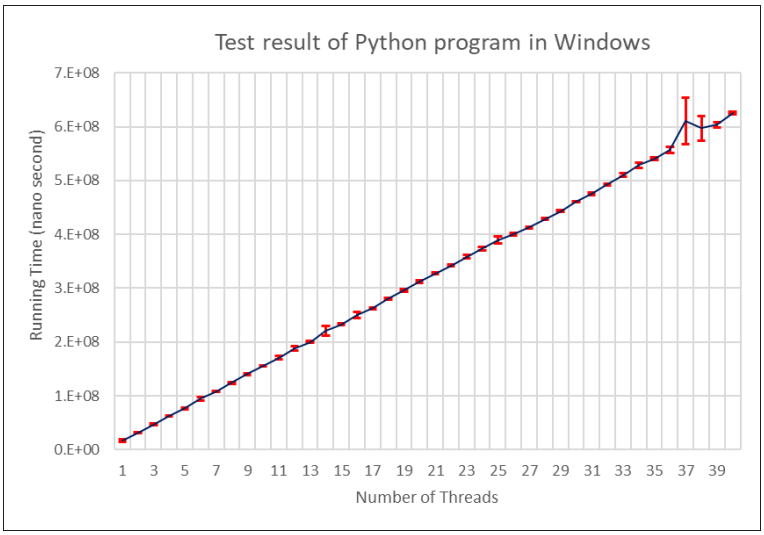


Figure 8: Python multi-thread result in Windows

6.3. Performance on JAVA Program

In Linux operating system Java program output mirrors the C program output. After 4 threads, the execution times becomes linear. But the output is different in both mac OS and windows. Standard deviations are also higher in mac and windows, implying that in each iteration with same number of thread, the execution time varies. Specially, in the first iteration the execution time and standard deviation looks very unusual. We can say that Java program displays different behavior in different operating system. One of the reason might be that Java threads are implemented entirely in the Java Virtual Machine (JVM) [7]. So, it is kind of unpredictable that how Java program will behave.

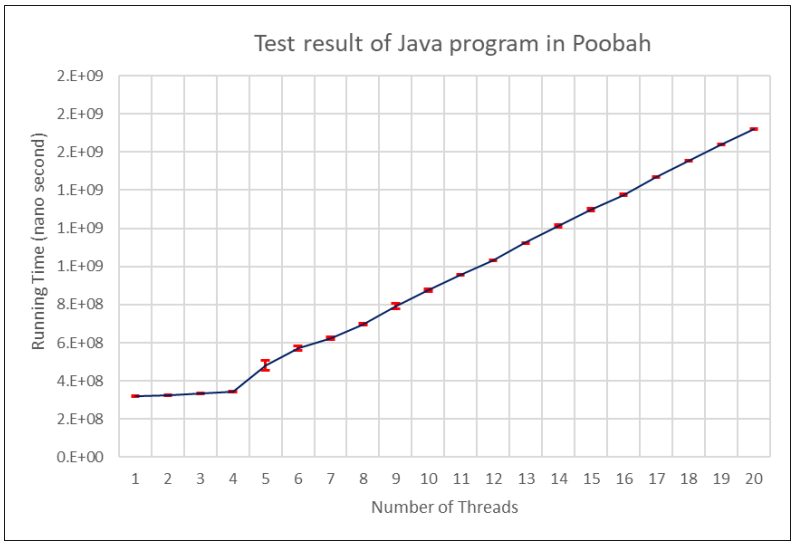


Figure 9: Java multi-thread result in Poobah(linux)

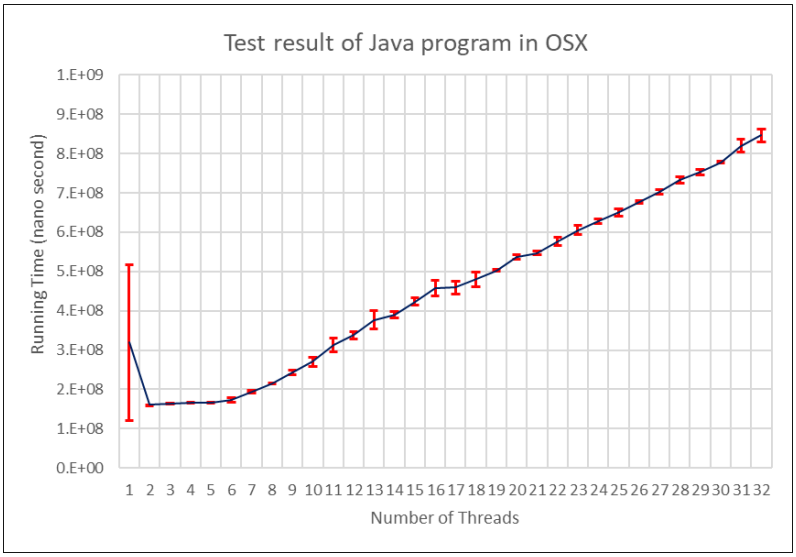


Figure 10: Java multi-thread result in OSX(mac)

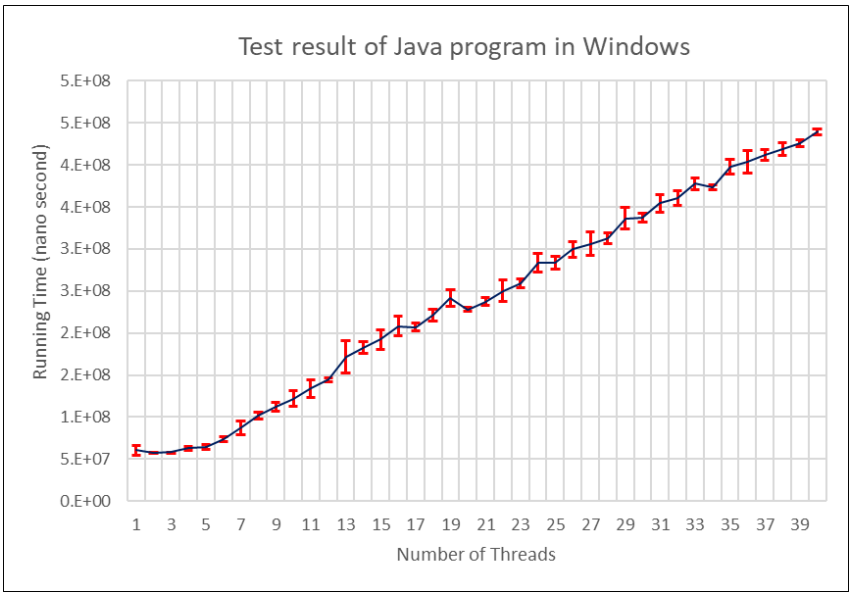


Figure 11: Java multi-thread result in Windows

6.4. Performance Comparison in different machine

We have seen from the above plots the performance of different multithread program in different machines. Below is a comparison of how execution time increased over thread counts in each machine. Since, the array size is different in different program, the comparison will not give the comparison of how fast the language performs but how it behaves with the increasing number of threads.

From Figure 12, we can clearly see that in the Poobah(Linux) machine with 4 core, when number of thread exceed 4, there is sharp increase in execution time both in C and Java but the Python execution time is linear from single thread. Standard deviation also less which implies that the result of the execution time in each iterations are consistence.

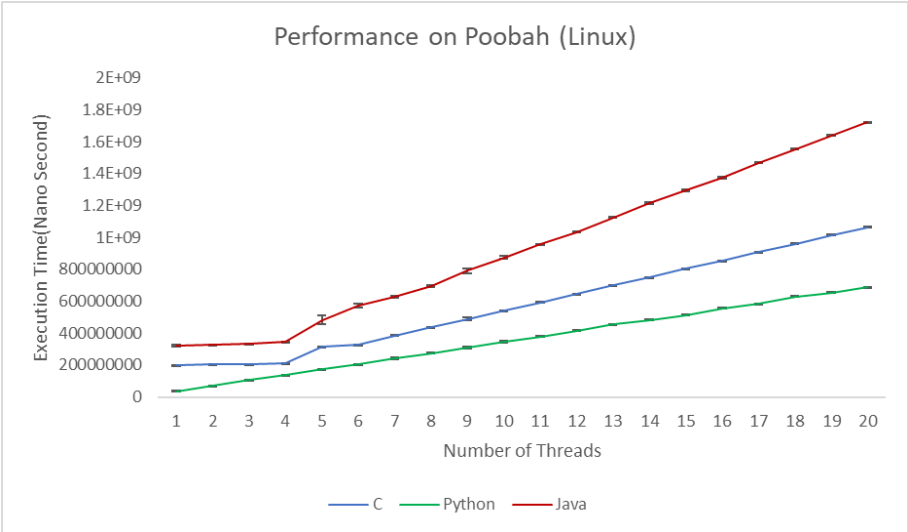


Figure 12: Change of execution time with the increase of thread count on Poobah(Linux)

The following figure shows the performance in a Mac OS with 8 cores. Like Linux, here also a sharp increase observed in Java and C after 6 threads. The execution time in Java using single thread is higher and abnormal from the trend. Thread creation overhead in Mac OS might be a reason for this. Python keeps a similar linear trend as Linux.

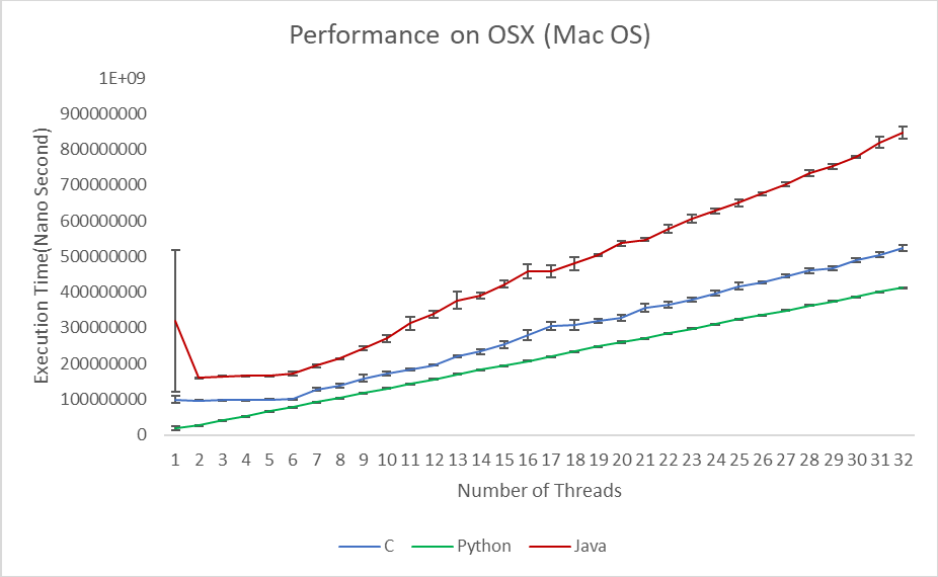


Figure 13: Change of execution time with the increase of thread count on OSX(Mac OS)

The performance on Windows platform is completely different from Linux and Mac OS. Python is in linear trend. But execution time is increasing from the beginning for both C and Java. But up to 9 thread the increment is less and after that increment in each step is higher. Different behavior in windows mostly contributed by a number of programs that running in backgrounds , occupying the threads which is also discussed in section 6.1.

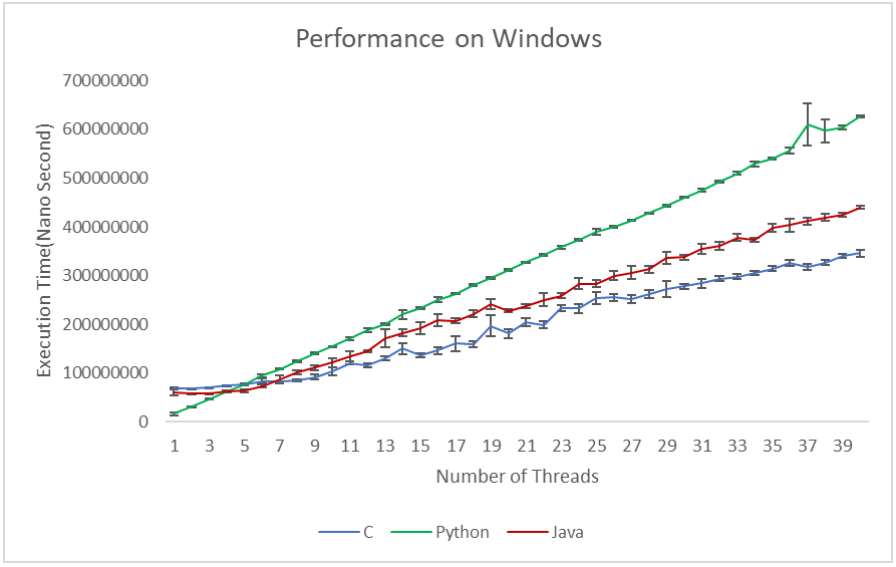


Figure 14: Change of execution time with the increase of thread count on Windows

7. Conclusion:

Through this experiment, we can say that, determining number of cores by running a program using multithreading depends on both programming language used and the operating system. While C offers more direct control over threads and memory management, threading in Java is handled by JVM and python normally don't allow to run multiple thread at once. Operating system also played a critical role in this experiment. We observed that the multithreading worked more efficiently in poobah(Linux) . One probable reason that this machine is free of unnecessary software's that runs in background. So, all of the thread is available to execute the program.

In the end, the number of CPU cores was successfully determined using C and Java program using the Linux machine(poobah). It was not possible to determine the number of cores using the python program. Running C and Java in mac OS and windows can give an idea on core count but cannot give exact number of cores.

References:

[1] Intel® Core™ i7-12650H Processor official webpage, retrieved from Intel Core i712650H Processor 24M Cache up to 4.70 GHz Product Specifications

[2] Apple M1 wiki page, Retrieved from Apple M1 - Wikipedia

[3] Benefits of C language over other programming languages, retrieved from https://www.geeksforgeeks.org/benefits-c-language-programming-languages/

[4] Multithreading in Python: The Ultimate Guide, retrieved from Multithreading in Python: The Ultimate Guide (with Coding Examples) (dataquest.io)

[5] What Is the Python Global Interpreter Lock (GIL)?, retrieved from What Is the Python Global Interpreter Lock (GIL)? – Real Python

[6] An Intro to Threading in Python, retrieved from An Intro to Threading in Python – Real Python

[7] Difference Between Java Threads and OS Threads, retrieved from Difference Between Java Threads and OS Threads - GeeksforGeeks

Appendix:

[C Code]

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
```

```c
#define num_iteration 5 // Number of iterations for each thread count


// Pre condition: None
//Post Condition: Filling up an array repeatedly
void *fill_array(void *arg)
{
    int array_size = 10000; // Size of the array
    int new_array[array_size]; // The array to be filled
    int i, j;
    for (j = 0; j < 10000; j++) // Outer loop for repeated filling
    {
        for (i = 0; i < array_size; i++) // Inner loop to fill the array
        {
            new_array[i] = i; // Assigning values to array elements
        } // end for
    } // end for
    pthread_exit(NULL); // Exit the thread
} // end fill_array

// Pre Condition : A function to fill up an array
// Post Condition: Run the fumction using multple thread and return execution
time and standard deviation
int main()
{
    int num_threads_max; // variable for maximum number of thread to use in the
program
    printf("Enter the maximum number of threads: ");
    scanf("%d", &num_threads_max); // Input for the maximum thread number
    printf("\n");
    pthread_t threads[num_threads_max]; // Array of pthreads to store thread IDs
    double results[num_iteration]; // Array to store execution times for each
iteration of thread

    // Loop through different thread counts
    for (int num_threads = 1; num_threads <= num_threads_max; num_threads++) //
Outer loop iterating over different thread counts
    {
        printf("Running with: %d thread:\n", num_threads);
        // Loop for multiple iterations
        for (int k = 0; k < num_iteration; k++) // Inner loop for multiple
iterations for num_threads
        {

            struct timeval start, end; // Structure to hold start and end times
```

```
            gettimeofday(&start, NULL); // Execution start time

            // Create and run threads
            for (int i = 0; i < num_threads; i++) // Loop to create multiple
threads
            {
                pthread_create(&threads[i], NULL, fill_array, NULL); // Create a
thread and call fill_array function
            } // end for

            // Wait for threads to finish
            for (int j = 0; j < num_threads; j++) // Join all  threads
            {
                pthread_join(threads[j], NULL); // Wait for thread to finish
execution
            } // end for


            gettimeofday(&end, NULL); // Execution end time

            // Convert time spent to nanosecond
            long long startTime = start.tv_sec * 1000000000LL + start.tv_usec *
1000LL;
            long long endTime = end.tv_sec * 1000000000LL + end.tv_usec * 1000LL;

            // Calculate execution time
            double cpu_time_used = endTime - startTime; // Calculate elapsed time
            printf("Iteration %d running time: %.0f nano seconds\n", k+1,
cpu_time_used);
            results[k] = cpu_time_used; // Store the execution time in the
results array
        }// end for

        // Calculating the average running time
        long total = 0; // Variable to store total execution time of all the
iteration
        for (int i = 0; i < num_iteration; i++) // Loop to calculate total
execution time
        {
            total += results[i]; // total execution time of all the iteration
        } // end for

        // Calculate average
        long avg = total / num_iteration; // Calculate average execution time
```

```
        printf("Average Running time using %d thread: %ld nanoseconds\n",
num_threads, avg);

        // Calculating Standard Deviation

        // Calculate the squared differences from the mean
        double squared_diff_sum = 0.0; // Variable to store sum of squared
differences
        for (int i = 0; i < num_iteration; ++i) // Loop to calculate squared
differences
        {
            double diff = results[i] - avg; // Calculate difference from mean
            squared_diff_sum += diff * diff; // Sum of squared differences
        } // end for

        // Calculate the variance
        double variance = squared_diff_sum / num_iteration;

        // Calculate the standard deviation (square root of variance)
        double standard_deviation = sqrt(variance); // Calculate standard
deviation

        // Print standard deviation
        printf("Standard Deviation using %d thread: %.2f\n\n", num_threads,
standard_deviation);
    } // end for

    return 0;
} // end main
```

[Python Code]

```
import threading
import time
import statistics
import os

cpu_count = os.cpu_count() # Get the core count of the computer
print(f'Number of CPUs: {cpu_count}\n' )

# Function to fill the array
# Pre condition: None
# Post Condition: Filling up an array repeatedly
def fill_array():
    array_size = 10000  # Size of the array
```

```
    # Fill the array 100 times
    for _ in range(100):

        new_array = [i for i in range(array_size)]
    # end for
# end fill_array

# Function to run the threads
# Pre condition: A function to fill up an array multiple times
# Post Condition: fill_array function will be executed using multiple threads and
return the execution time

def thread_count(num_threads_max):
    iteration = []  # List to store execution times of each iteration
    print(f'Running with {num_threads_max} thread:')

    # Run 5 iterations
    for i in range(1, 6):
        start_time = time.time()  # Get the start time of the iteration
        threads = []  # List to store the thread objects

        # Create and start multiple threads
        for _ in range(num_threads_max):
            # Create a thread targeting the fill_array function
            thread = threading.Thread(target=fill_array)
            threads.append(thread)  # Append the thread to the list
            thread.start()  # Start the thread
        #end for

        # Wait for all threads to finish
        for thread in threads:
            thread.join()
        # end for

        end_time = time.time()  # End time of the iteration

        # Calculate the execution time of the iteration in nanoseconds
        run_time = (end_time - start_time) * 1e9
        iteration.append(run_time)  # Append the execution time to the list


        print(f'Iteration {i} running time: {run_time} nanoseconds')
    # end for
```

```
    # Calculate and print the average and standard deviation
    standard_deviation = statistics.stdev(iteration) # Standard Deviation
    avg = statistics.mean(iteration) # Average
    print(f'Average Running time using {num_threads_max} thread:  {avg}
nanoseconds')
    print(f'Standard Deviation using {num_threads_max} thread:
{standard_deviation} \n')
# end thread_count


# Run the thread_count function
num_threads = int(input("Enter the maximum number of threads: ")) # get the max
number of thread from user input
# Execute the thread_count function using 1 thread to num_thread
for i in range(1, num_threads+1):
    thread_count(i) # call the thread_count function
#end for
```

[Java Code]

```
import java.util.Arrays;
import java.util.Scanner;

//Pre condition: None
//Post condition: Execute the fill_array method using multiple thread for
multiple iteration. Return average and standard deviation
// of runnning time for each thread
public class multi_thread {
    public static void main(String[] args) {
        int cpuCount = Runtime.getRuntime().availableProcessors(); // Get the
core count of the computer
        System.out.println("Number of CPUs: " + cpuCount);
        System.out.println();

        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the maximum number of threads: ");
        final int num_threads_max = scanner.nextInt(); // Maximum number of
threads to use
        scanner.close();
        final int numIterations = 5; // Number of iterations per thread
        final int arraySize = 100000; // Size of the array

        // Iterate over each thread configuration
        for (int numThreads = 1; numThreads <= num_threads_max; numThreads++) {
            System.out.println("Running with " + numThreads + " thread:");
```

```
            long[] executionTimes = new long[numIterations]; // Array to store
execution times for each iteration

            // Run multiple iterations
            for (int i = 0; i < numIterations; i++) {
                long startTime = System.nanoTime(); // Start time of the
iteration
                fill_array(arraySize, numThreads); // Run the iteration
                long endTime = System.nanoTime(); // End time of the iteration
                long iterationTime = endTime - startTime; // Calculate iteration
time
                executionTimes[i] = iterationTime; // Store iteration time in the
array
                System.out.println("Iteration " + (i + 1) + " running time: " +
iterationTime + " nanoseconds");
            } // end for

            // Calculate and print average and standard deviation
            double average = get_average(executionTimes); // Calculate average
execution time
            double standard_deviation = get_std_dev(executionTimes); // Calculate
standard deviation
            System.out.println("Average Running time using " + numThreads + "
thread: " + average + " nanoseconds");
            System.out.println(
                    "Standard deviation using " + numThreads + " thread: " +
standard_deviation + " nanoseconds");

            System.out.println();
        } // end for
    } // end main

    // Method to fill the array
    // Pre condition: None
    // Post Condition: Filling up an array repeatedly using multiple threads
    public static void fill_array(int arraySize, int numThreads) {
        Thread[] threads = new Thread[numThreads]; // Create an array of threads

        for (int i = 0; i < numThreads; i++) {
            // final int threadIndex = i;
            threads[i] = new Thread(() -> {
                int[] new_array = new int[arraySize];
                for (int j = 0; j < 10000; j++) // Outer loop for repeated
filling
                {
```

```
                    for (int k = 0; k < arraySize; k++) // Inner loop to fill the
array
                    {
                        new_array[k] = k; // Assigning values to array elements
                    }
                }
            });
            threads[i].start(); // Start each thread
        } // end for

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join(); // Wait for each thread to finish
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } // end for
    } // end fill_array

    // Method to calculate average
    public static double get_average(long[] numbers) {
        long sum = 0;
        for (long num : numbers) {
            sum += num; // Sum up all the numbers
        }
        return (double) sum / numbers.length; // return the average
    }

    // Method to calculate standard deviation
    public static double get_std_dev(long[] numbers) {
        double mean = get_average(numbers); // Calculate the mean
        double squared_diff_sum = 0; // Variable to store sum of squared
differences
        for (long num : numbers) { // Loop to calculate squared differences
            squared_diff_sum += (num - mean) * (num - mean); // Sum of squared
differences
        }
        // Calculate the variance
        double variance = squared_diff_sum / numbers.length;
        return Math.sqrt(variance); // Return the standard deviation
    }
}
```