Report: Determining the Number of Cores
CS574
Long Tran

Introduction

Modern CPUs come with multiple cores that are selling features, but we cannot always trust the companies from the advertisement. Thus, in this experiment, we will test the factuality of the CPU information reported by either the OS or the manufacturers on a small sample of 3 machines. The 3 machines tested will range from different OSes and CPU brands:

1) Machine 1: Linux with Intel.
2) Machine 2: MacOS with Apple Silicon.
3) Machine 3: Windows 11 with AMD.

The most straightforward way to test would be to write a program that spawns kernel threads to do some tasks.
The time taken should be about the same when the numbers of threads do not exceed the number of cores. However, when the numbers of threads double the number of cores, the time taken should at least double that of the previous case.

This should mean that we need to write a program that can make use of kernel threads as user threads cannot utilize multiple cores well. Thus, another experiment is to find the type of threads that a programming language uses. We tested 3 different languages: C, Java, and Python. We expect that C uses kernel threads since it is the closest to the hardware among the 3 languages, while the others will be figured out.

Methodology

The experiment was done by spawning a maximum number of threads from 1 thread to N threads where N is 4 times the number of cores of the test machine. The time taken for spawning each max number of threads is measured and reported and we will do analysis on the result to get answers.

The process that each thread has to execute: Each thread will loop 2^{20} times over an array of size 10. Then, it will sum the values of the array to the index 0 of the array.

Time Measurement

The time reported for each max number of threads was an average of 5 runs. Also, the standard deviation of the 5 runs was also reported.
The time was measured in nanoseconds using the perf_counter() function for Python, System.nanoTime() for Java, clock_gettime() for C-Linux, and QueryPerformanceFrequency() for C-Windows.

The machines tested

1. poobah.cs.nmsu.edu machine:
   - OS: OpenSUSE Leap 15.5.

- CPU: Intel i7-2600 (from [1] and [2]):
    - 4 cores, 8 threads, 1.6GHz-3.40 GHz per core.
    - L1 Cache: 128 KiB.
    - L2 Cache: 1 MiB.
    - L3 Cache: 8 MiB.

2. Macbook Air M2:
- OS: macOS 14.3 (23D56).
- CPU: Apple Silicone M2 (from [3] and [4]):
    - 4 efficiency cores:
        - L1 Instruction Cache: 128 KB.
        - L1 Data Cache: 64 KB.
        - L2 Cache: 4 MB.
    - 4 performance cores (higher clock frequency):
        - L1 Instruction Cache: 198 KB.
        - L1 Data Cache: 128 KB.
        - L2 Cache: 16 Mb.
    - In total, there are 8 cores/8 threads.
    - System Cache: 8 MB.

3. Windows Laptop:
- OS: Microsoft Windows 11 Home, Version 10.0.22631 Build 22631
- CPU: AMD Ryzen 7 5800H with Radeon Graphics (from [5] and [6]):
    - 8 cores, 16 logical processors (threads), 3.2GHz-4.4GHz.
    - L2 Cache: 4 MB.
    - L3 Cache: 16 MB.

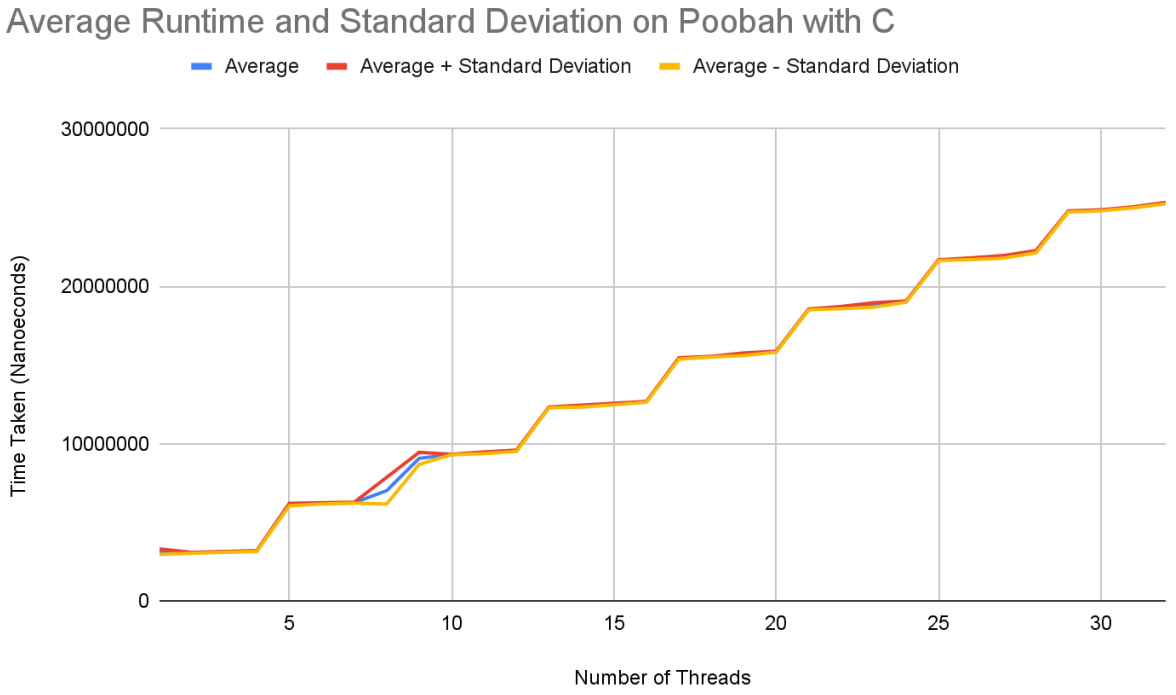
The languages used: *The code is put in Appendix A.*

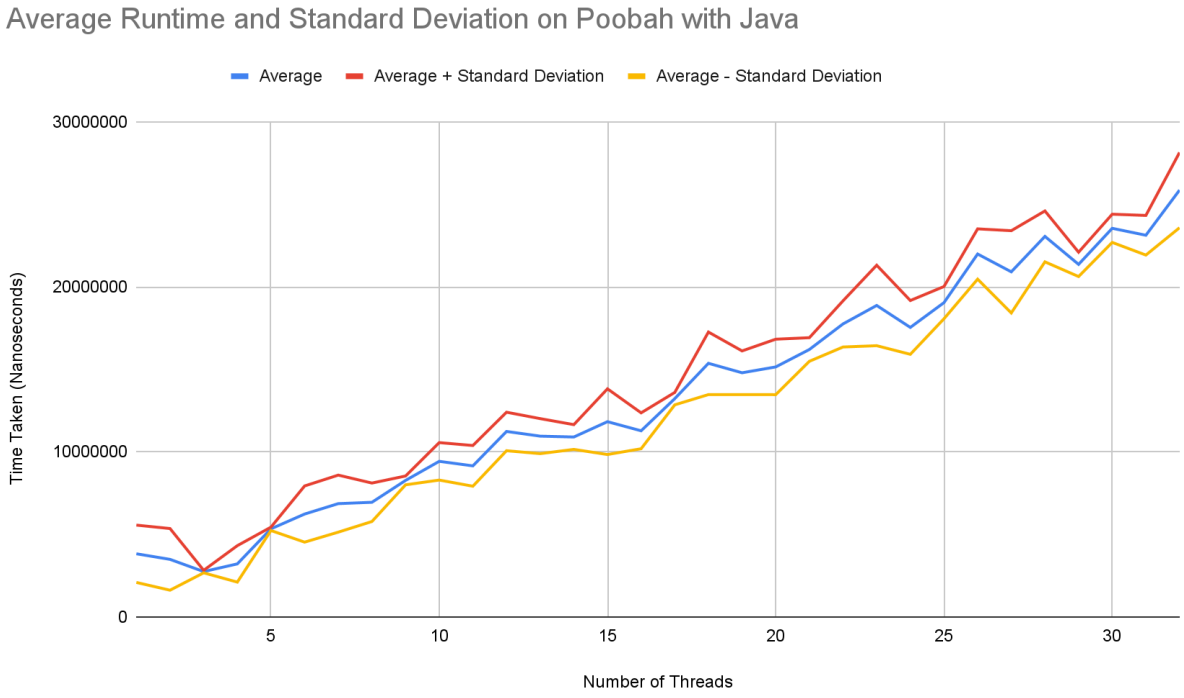
1. Python (the default Python is CPython).
2. Java.
3. C:
    a. Windows API threads (for Windows).
    b. POSIX threads (for MacOs and Linux).

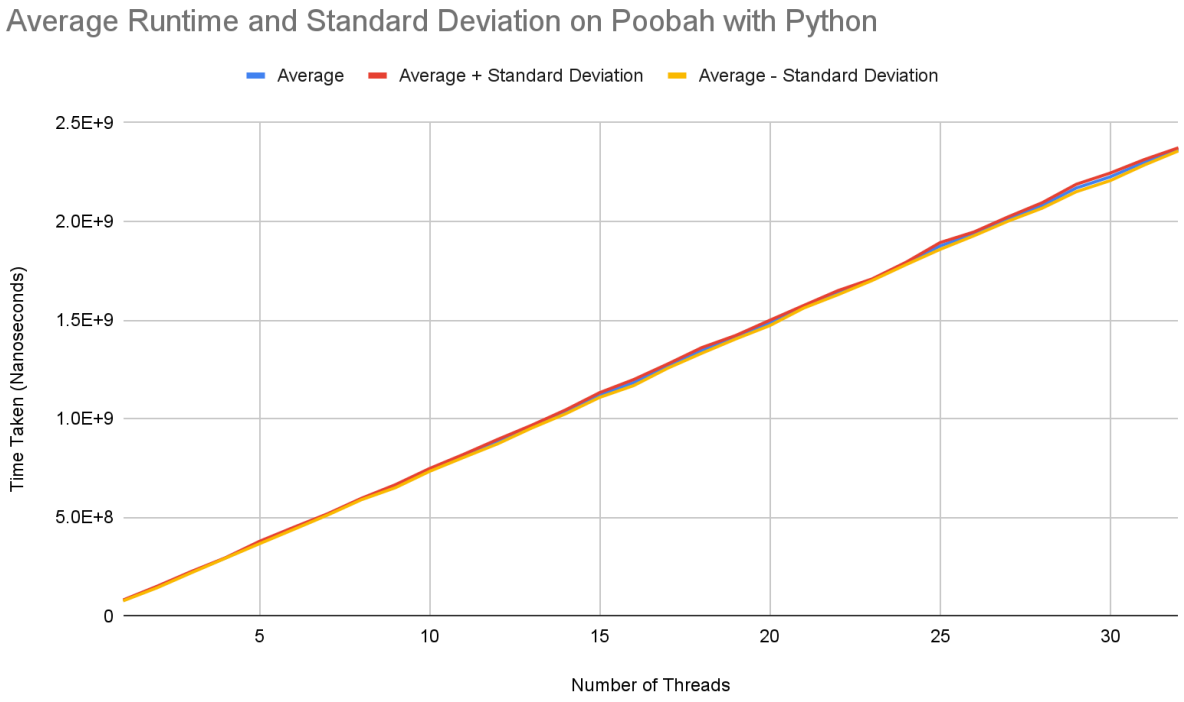
Result

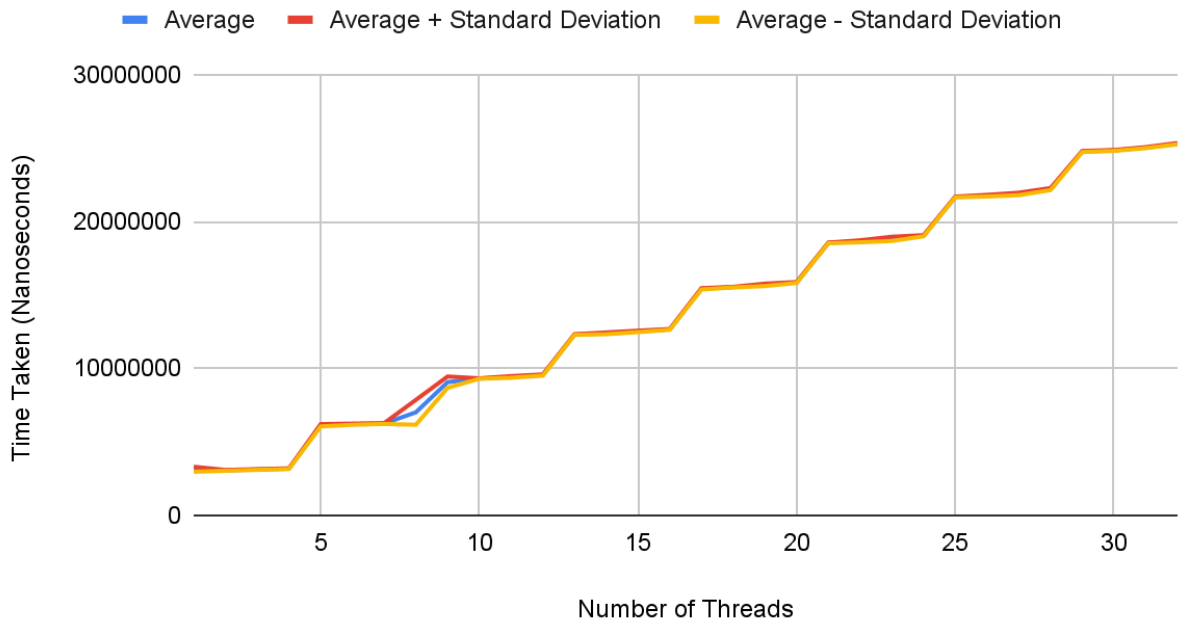
The numerical data table is put in Appendix B.
In the graphs below:

1) *"Average" is the average runtime.*
2) *"Average + Standard Deviation" is the average runtime plus the standard deviation.*
3) *"Average - Standard Deviation" is the average runtime minus the standard deviation.*

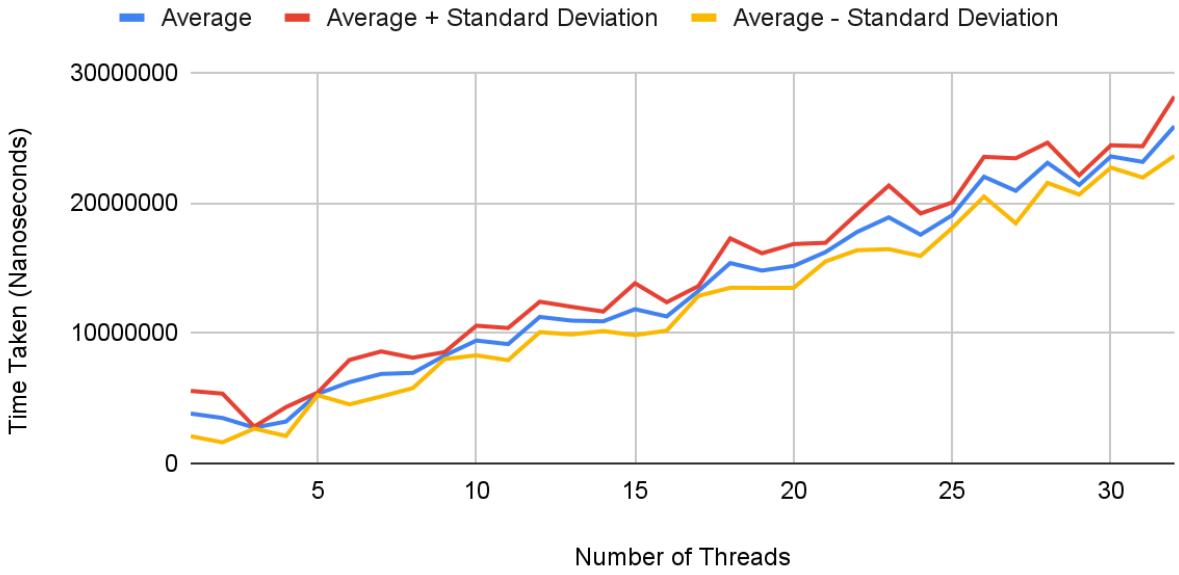

Average Runtime and Standard Deviation on Poobah with C
Average
Average + Standard Deviation
Average - Standard Deviation
30000000
20000000
10000000
0
Time Taken (Nanoseconds)
5
10
15
20
25
30
Number of Threads


Average Runtime and Standard Deviation on Poobah with Java
Average
Average + Standard Deviation
Average - Standard Deviation
30000000
20000000
10000000
0
Time Taken (Nanoseconds)
5
10
15
20
25
30
Number of Threads

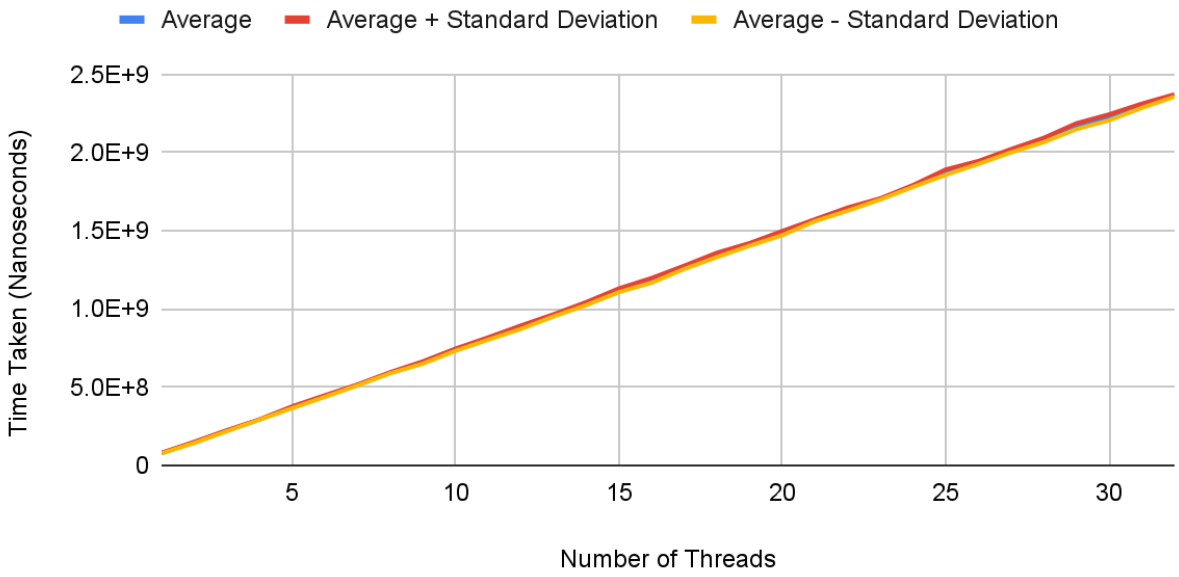

Average Runtime and Standard Deviation on Poobah with Python
Average
Average + Standard Deviation
Average - Standard Deviation
2.5E+9
2.0E+9
1.5E+9
1.0E+9
5.0E+8
0
Time Taken (Nanoseconds)
5
10
15
20
25
30
Number of Threads


Average Runtime and Standard Deviation on Apple M2 with C
Average
Average + Standard Deviation
Average - Standard Deviation
30000000
20000000
10000000
0
Time Taken (Nanoseconds)
5
10
15
20
25
30
Number of Threads

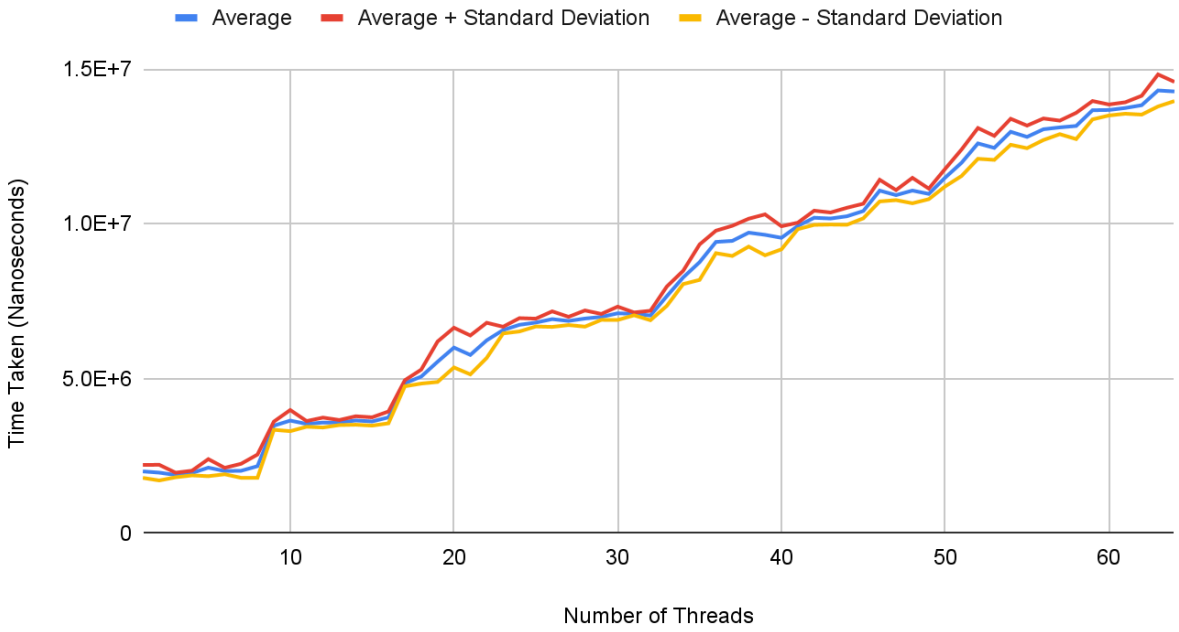
Average Runtime and Standard Deviation on Apple M2 with Java



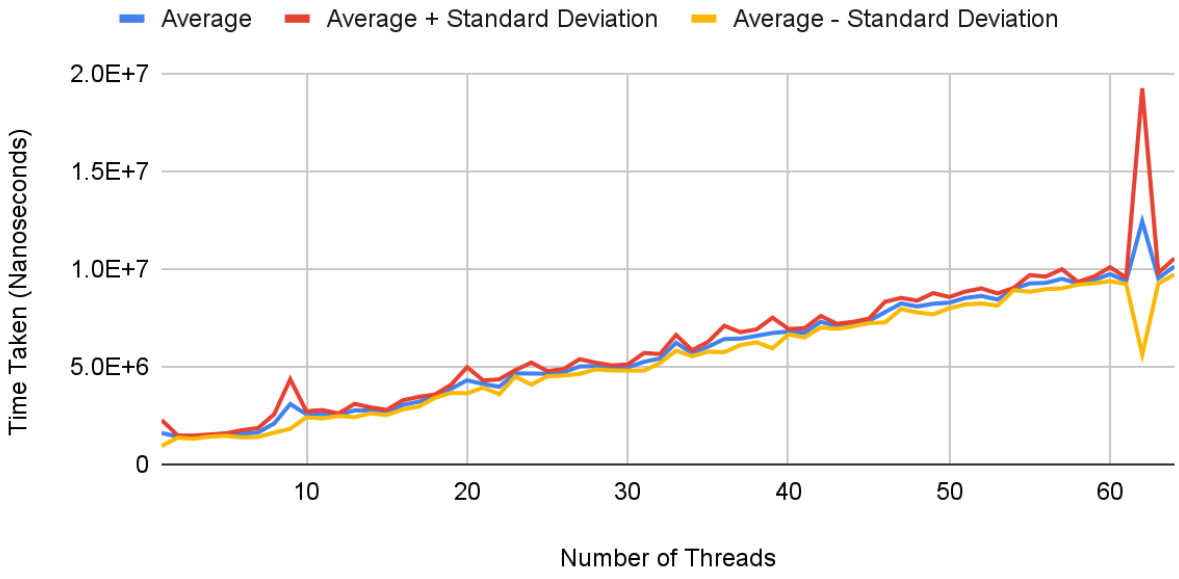
Average Runtime and Standard Deviation on Apple M2 with Python



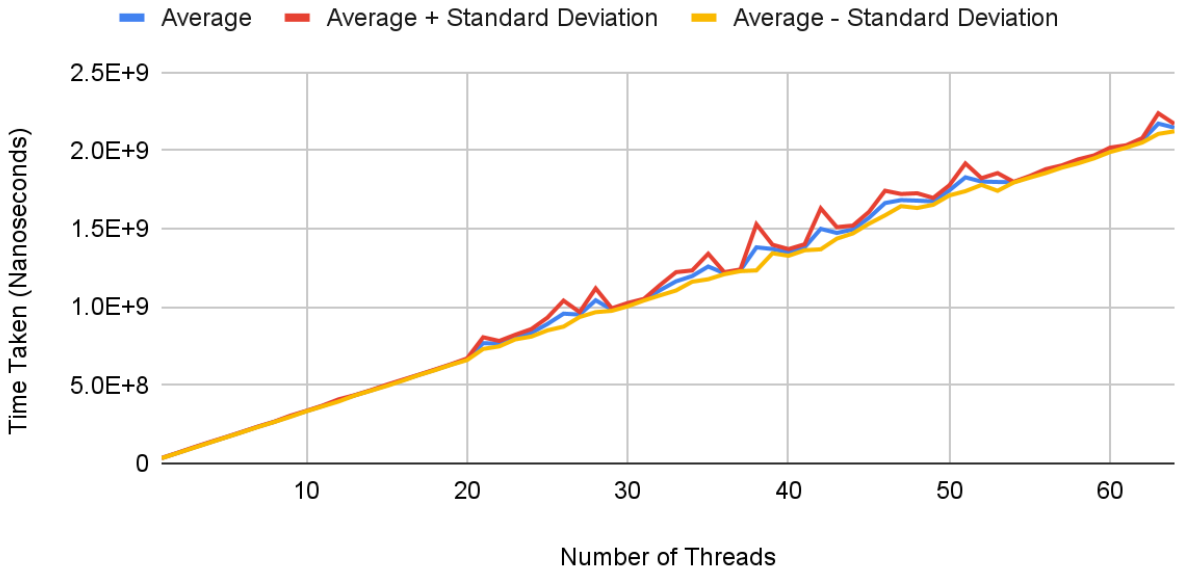
Average Runtime and Standard Deviation on Windows 11 with C



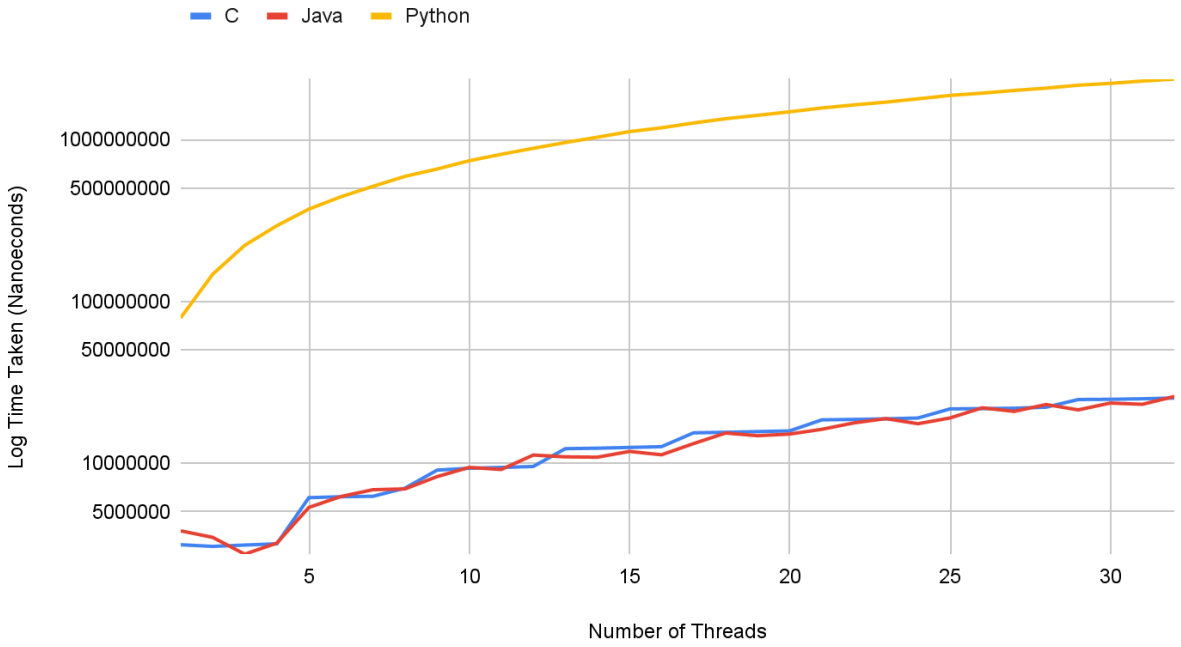
Average Runtime and Standard Deviation on Windows 11 with Java

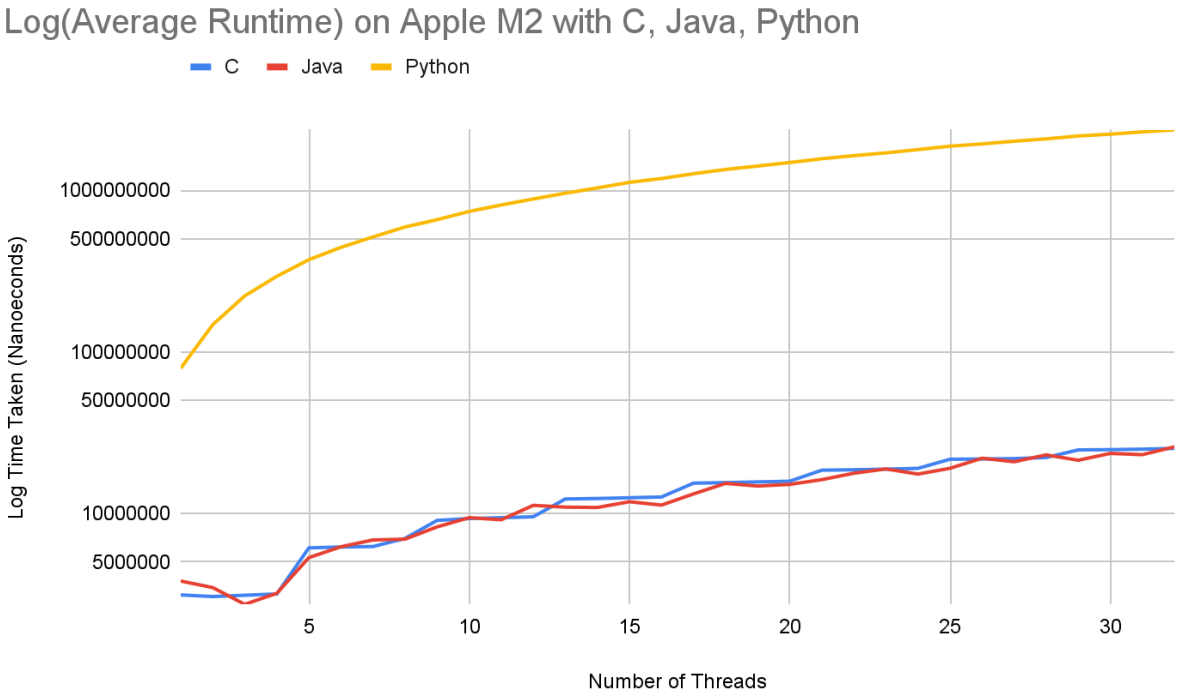


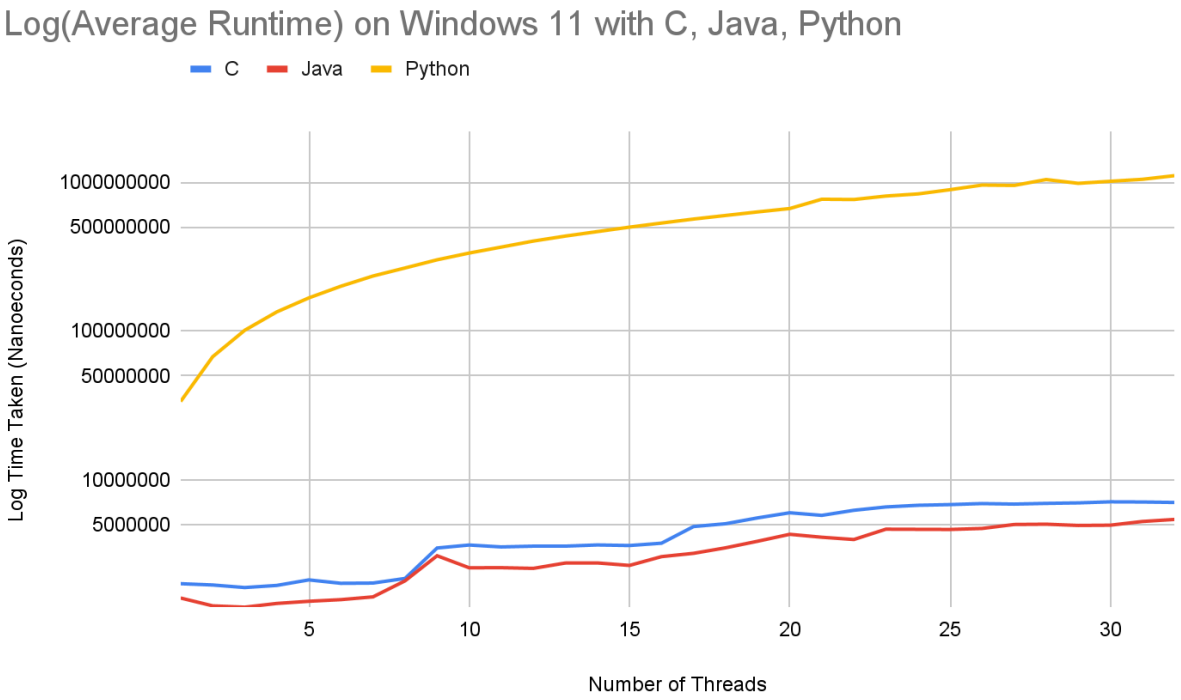
Average Runtime and Standard Deviation on Windows 11 with Python



Log(Average Runtime) on Poobah with C, Java, Python







Analysis

The Number of Cores

Looking at the result of C across all different machines, manufactures seem to have reported quite correctly. On the Intel machine with 4 cores (poobah), we can see that the time taken remained relatively the same when the number of threads was less than 5. Similarly, on the AMD machine with 8 cores (Windows 11), the time taken when the number of threads was less than 9 remained relatively the same. Then, when the 2 machines crossed that threshold of cores, the time taken at least doubled.
Hence, we can conclude that Intel and AMD reported their cores correctly. We can also see that although each CPU has 2 hyper threads per core, when the number of threads doubled the number of cores, the time taken was still doubled. Hence, hyper threads technology does not seem to improve the performance much.

The only outlier was the Apple M2 chip with 4 efficiency and 4 performance cores (8 cores in total); we can see that the time taken remained relatively the same when the number of threads was less than 5, but the time doubled when threads was greater than 4. This could be either because Apple reported the information wrongly or because MacOs decided to not let the performance cores involved.
According to Hoakley (2022) in an article about Apple M-series architecture, ARM architecture divides CPU into clusters and does cluster switching to switch a task from a cluster of performance cores to a cluster of efficiency cores and vice versa if needed to save battery life [7]. This initially suggested to me that it is not possible for a process to run on both types of cores at the same time so switching implementation is easier. However, according to Apple, it is possible for a process with multiple threads to run on both types at the same time, but the decision is made mostly based on the observations of the OS and also the user's explicit request [9]. Regardless, this means that the limitation was due to the operating system.

I looked into the methods to explicitly request the usage of all cores from the article [9], and it seemed that I would need to write the programs in Objective-C or Swift. It seems that I will have to use the Grand Central Dispatch (GCD) API to spawn and manage the threads instead of the normal POSIX thread. Thus, I think this can be left for another experiment in the future.

Languages Analysis
Given the base result from C and the graph for Python, we can see that Python does not use kernel threads. This is because as we increased the number of threads, the time taken linearly increased across all 3 test machines. Thus, Python would be unsuitable for determining the number of cores following the mentioned method. The reason is because only one thread can be executed at a time in the CPython implementation due to Global Interpreter Lock [8].

As for Java, the “average+deviation” and “average-deviation” lines across all 3 machines showed that the time taken by Java was not as consistent (big gap between the “average”, the “average+deviation” and “average-deviation” line) as C. This suggested that the threads were user threads scheduled by the Java Virtual Machine.
However, it seems that the Java Virtual Machine also makes use of kernel threads to run user threads because the increasing rate of the runtime was not linear like that of Python when the number of threads increased to double that of the number of cores across the machines.

This was partly confirmed by the documentation from Oracle. According to Oracle in [10], the threads that were used in my experiment code is platform thread where the JVM maps and runs the Java threads on kernel threads [10].
This operation definitely comes with extra overhead compared to C, so I think Java is also not suitable for determining the number of cores using the method mentioned in the methodology. However, unexpectedly, the average runtime of Java and C (POSIX threads) is relatively the same according to the log-scale graphs on poobah and Macbook machines. Also, in the Windows 11 machine, Java is even faster than C (win32 api threads). This suggested that the overhead is not as much as I expected and the method I used was not able to bring out clearly the overhead of JVM compared to bare C code.
Yet, I think C is a more suitable language than Java for the task because the runtime of C is more consistent than that of Java as discussed.

Conclusion

In conclusion, we found that Intel and AMD seemed to tell the truth about the number of cores while Apple is unknown since I have not been able to utilize all the cores fully. Also, we learned that C is the best language for figuring out the number of cores by spawning multiple threads to execute some tasks and see the runtime, while Java and Python use user threads so it is not suitable for such a method.

References

[1] Intel CPU Information. Retrieved from “lscpu” on the poobah.cs.nmsu.edu machine:

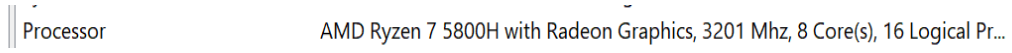
```
  Model name:            Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
    CPU family:          6
    Model:               42
    Thread(s) per core:  1
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            7
    CPU max MHz:         3800.0000
    CPU min MHz:         1600.0000
    BogoMIPS:            6784.85
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic
                         erfmperf pni pclmulqdq dtes64 monitor ds
                         d xsaveopt dtherm ida arat pln pts md_cl
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   128 KiB (4 instances)
  L1i:                   128 KiB (4 instances)
  L2:                    1 MiB (4 instances)
  L3:                    8 MiB (1 instance)
```

[2] Intel CPU Information. *Intel Official Website*. Retrieved from https://www.intel.com/content/www/us/en/products/sku/52213/intel-core-i72600-processor-8m-cache-up-to-3-80-ghz/specifications.html

[3] Apple Silicone M2 Information. Retrieved from “sysctl -a | grep machdep.cpu”:

```
(base) ➜  DetermineCores sysctl -a | grep machdep.cpu
machdep.cpu.cores_per_package: 8
machdep.cpu.core_count: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
machdep.cpu.brand_string: Apple M2
```

[4] Apple Silicone M2 Information. *Wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Apple_M2
[5] AMD Ryzen 7 5800H Information. Retrieved from the System Information application of Windows 11:



[6] AMD Ryzen 7 5800H Information. *AMD Official Website*. Retrieved from https://www.amd.com/en/products/apu/amd-ryzen-7-5800h#product-specs
[7] Hoakely (October 3rd 2022). *Making the most of Apple silicon power: 1 M-series chips are different.* Retrieved from https://eclecticlight.co/2022/10/03/making-the-most-of-apple-silicon-power-1-m-series-chips-are-different/
[8] Threading in Python. *Python Official Documentation*. Retrieved from https://docs.python.org/3/library/threading.html
[9] Optimize for Apple Silicon. *Apple. Retrieved from* https://developer.apple.com/news/?id=vk3m204o#:~:text=Apple%20Silicon%20Macs%20are%20AMP,over%20a%20period%20of%20time.
[10] Threading in Java. *Oracle Java Documentation.* Retrieved from https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-2BCFC2DD-7D84-4B0C-9222-97F9C7C6C521

Appendix A

Windows - C (uses Windows.api threads)

```
// Program to spawn from 1 to MAX_THREADS_RUN number of threads
// Then fill an array 1 << 20 times.
// Each thread number is run five times.
// The run time of each run is printed to STDERR
// The program will also calculate the average and standard deviation.
// Then, it will be printed out on STDOUT.
// Author: Long Tran
// Date: Feb 10, 2024
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include <heapapi.h>
#include <process.h>

#define MAX_THREADS_RUN <...max number of threads to test here...>
#define AVERAGE_RUN 5
```

```cpp
double get_average(long long* values, int length) {
    double result = 0;
    for (int i = 0; i < length; ++i) {
        result += (values[i] / length);
    } // end for i

    return result;
} // end average

double std(long long* values, int length, double mean) {
    double result = 0;
    for (int i = 0; i < length; ++i) {
        double diff = values[i] - mean;
        diff = diff * diff;
        diff = diff / (length-1);
        result += diff;
    } // end for i

    return sqrt(result);
} // end average

DWORD WINAPI myThreadFunction(void *vargp)
{
    int length = 1 << 20;
    int array[10];
    for (int i = 0; i < length; ++i) {
        array[i % 10] = 0;
    } // end for i

    for (int i = 0; i < 10; ++i) {
        array[0] += array[i];
    } // end for i

    return 0;
} // end myThreadFunction

//https://learn.microsoft.com/en-us/copp/build/walkthrough-compiling-a-native-cpp-program-on-the-command-line?view=msvc-170&viewFallbackFrom=vs-2019
```

```
//https://learn.microsoft.com/en-us/windows/win32/procthread/creating-threads
void runThreads(int threadNum) {
   // thread id
   HANDLE * threads = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
sizeof(HANDLE) * threadNum);

   // create threads
   for (int i = 0; i < threadNum; ++i) {
      // create thread
      threads[i] = CreateThread(NULL, 0, myThreadFunction, NULL, 0, NULL);
   } // end for i

   // wait for threads
   WaitForMultipleObjects(threadNum, threads, TRUE, INFINITE);

   // free mem and close handle
   for (int i = 0; i < threadNum; ++i) {
      CloseHandle(threads[i]);
   } // end for i
   // free mem
   HeapFree(GetProcessHeap(), 0, threads);
   threads = NULL;
} // end runThreads

int main()
{
   for (int threadNum = 1; threadNum <= MAX_THREADS_RUN; threadNum++) {
      long long durations[AVERAGE_RUN];
      for (int round = 0; round < AVERAGE_RUN; ++round) {
         LARGE_INTEGER liFrequency = {0};
         // Get the Frequency
         if(QueryPerformanceFrequency(&liFrequency))
         {
            // Start Timing
            // adapted from
https://stackoverflow.com/questions/12956608/get-time-in-nanoseconds-with-c-on-windo
ws-without-chrono
            LARGE_INTEGER liStart = {0};
```

```
            if(QueryPerformanceCounter(&liStart))
            {
                // Do Stuff
                runThreads(threadNum);
                // Get Time spent...
                LARGE_INTEGER liStop = {0};
                if(QueryPerformanceCounter(&liStop))
                {
                    LONGLONG duration = (LONGLONG)((liStop.QuadPart - liStart.QuadPart) *
1000000000.0 / liFrequency.QuadPart);
                    durations[round] = duration;
                    fprintf(stderr, "Run %d with %d: %lld nanosec\n", round, threadNum,
duration);
                } // end if
            } // end if
        } // end if
    } // end for round
    double average = get_average(durations, AVERAGE_RUN);
    double standardDeviation = std(durations, AVERAGE_RUN, average);
    printf("%d, %f, %f\n", threadNum, average, standardDeviation);
  } // end for threadNum

  // exit
  return 0;
} // end main
```

MacOs/Linux - C (uses POSIX threads)

```
// Program to spawn from 1 to MAX_THREADS_RUN number of threads
// Then fill an array 1 << 20 times.
// Each thread number is run five times.
// The run time of each run is printed to STDERR
// The program will also calculate the average and standard deviation.
// Then, it will be printed out on STDOUT.
// Author: Long Tran
// Date: Feb 10, 2024
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <pthread.h>
#include <math.h>

#define MAX_THREADS_RUN 32
#define AVERAGE_RUN 5

double get_average(long long* values, int length) {
  double result = 0;
  for (int i = 0; i < length; ++i) {
     result += (values[i] / length);
  } // end for i

  return result;
} // end average
 double std(long long* values, int length, double mean) {
  double result = 0;
  for (int i = 0; i < length; ++i) {
     double diff = values[i] - mean;
     diff = diff * diff;
     diff = diff / (length-1);
     result += diff;
  } // end for i

  return sqrt(result);
} // end average

void *myThreadFunction(void *vargp)
{
  unsigned int length = 1 << 20;
  int array[10];
  for (int i = 0; i < length; ++i) {
     array[i % 10] = 0;
  } // end for i
```

```
  for (int i = 0; i < 10; ++i) {
     array[0] += array[i];
  } // end for i
  return NULL;
} // end myThreadFunction

void runThreads(int threadNum) {
  // thread id
  pthread_t * threads = malloc(threadNum * sizeof(pthread_t));

  // create threads
  for (int i = 0; i < threadNum; ++i) {
     // create thread
     pthread_create(&threads[i], NULL, myThreadFunction, NULL);
  } // end for i

  // wait for threads
  for (int i = 0; i < threadNum; ++i) {
     // get thread id
     pthread_t thread_id = threads[i];
     // wait for a thread
     pthread_join(thread_id, NULL);
  } // end for i

  // free the memory
  free(threads);
} // end runThreads

long long get_duration(struct timespec start, struct timespec end) {
  long j = end.tv_sec - start.tv_sec;
  long k = end.tv_nsec - start.tv_nsec;
  return j * 1000000000 + k;
} // end get_duration
```

```
int main()
{
  for (int threadNum = 1; threadNum <= MAX_THREADS_RUN; threadNum++) {
     long long durations[AVERAGE_RUN];
     for (int round = 0; round < AVERAGE_RUN; ++round) {
        struct timespec start, end;

        // run the process
        clock_gettime(CLOCK_MONOTONIC, &start); /* mark start time */
        runThreads(threadNum);
        clock_gettime(CLOCK_MONOTONIC, &end);   /* mark the end time */

        // get duration
        long long duration = get_duration(start, end);

        durations[round] = duration;

        fprintf(stderr, "Run %d with %d: %lld nanosec\n", round, threadNum, duration);
     } // end for round
     double average = get_average(durations, AVERAGE_RUN);
     double standardDeviation = std(durations, AVERAGE_RUN, average);
     printf("%d, %f, %f\n", threadNum, average, standardDeviation);
  } // end for threadNum

  // exit
  return 0;
} // end main
```

Java

```
// Program to spawn from 1 to MAX_THREADS_RUN number of threads
// Then fill an array 1 << 20 times.
// Each thread number is run five times.
// The run time of each run is printed to STDERR
```

```java
// The program will also calculate the average and standard deviation.
// Then, it will be printed out on STDOUT.
// Author: Long Tran
// Date: Feb 10, 2024
import java.util.Arrays;

public class Main {
  final static int MAX_THREADS_RUN = 32;
  final static int AVERAGE_RUN = 5;

  public static class MyRunnable implements Runnable {
    public void run() {
      int length = 1 << 20;
      int[] array  = new int[10];
      for (int i = 0; i < length; ++i) {
        array[i % 10] = 0;
      } // end for i

      for (int i = 0; i < 10; ++i) {
        array[0] += array[i];
      } // end for i
    } // end run
  } // end MyThreadk

  public static void runThreads(int threadNum) {
    Thread[] threads = new Thread[threadNum];
    for (int i = 0; i < threadNum; ++i) {
      threads[i] = new Thread(new MyRunnable());
      try {
        threads[i].start();
      } catch(IllegalThreadStateException e) {
        System.err.println(e.getMessage());
        System.exit(1);
      } // end catch
```

```
    } // end for i

    for (int i = 0; i < threadNum; ++i) {
      try {
        threads[i].join();
      } catch(InterruptedException e) {
        System.err.println(e.getMessage());
        System.exit(1);
      } // end catch
    } // end
  } // end runThreads

  public static void main(String[] args) {
    for (int threadNum = 1; threadNum <= Main.MAX_THREADS_RUN; threadNum++) {
      long[] durations = new long[Main.AVERAGE_RUN];
      for (int round = 0; round < Main.AVERAGE_RUN; ++round) {

        long start = System.nanoTime();
        // run the process
        runThreads(threadNum);
        long end = System.nanoTime();
        long duration = end - start;

        durations[round] = duration;

        System.err.printf("Run %d with %d: %d nanosec\n", round, threadNum,
duration);
      } // end for round
      double average = Arrays.stream(durations).average().getAsDouble();
      double standard_deviation= std(durations, average);
      System.out.printf("%d, %f, %f\n", threadNum, average, standard_deviation);
    } // end for threadNum
  } // end main
```

```
  public static double std(long[] values, double mean) {
     double sum = 0;
     for (long value: values) {
        double diff = (double) (value) - mean;
        diff = diff * diff;
        diff = diff / (values.length-1);
        sum += diff;
     } // end for

     return Math.sqrt(sum);
  } // end std
} // en Main
```

Python

```
# Program to spawn from 1 to MAX_THREADS_RUN number of threads
# Then fill an array 1 << 20 times.
# Each thread number is run five times.
# The run time of each run is printed to STDERR
# The program will also calculate the average and standard deviation.
# Then, it will be printed out on STDOUT.
# Author: Long Tran
# Date: Feb 10, 2024
import numpy as np
import threading
import time
import sys

MAX_THREADS_RUN = 32
AVERAGE_RUN = 5

def my_thread_func():
  length = 1 << 20;
  array  = [i for i in range(10)]
  for i in range(length):
```

```
        array[i % 10] = 0

    for i in range(10):
        array[0] += array[i]

def run_threads(thread_num):
    threads = [i for i in range(thread_num)]
    for i in range(thread_num):
        threads[i] = threading.Thread(target=my_thread_func)
        threads[i].start()

    for thread in threads:
        thread.join()

if __name__ == "__main__":
    for threadNum in range(1, MAX_THREADS_RUN+1, 1):
        durations = []
        for round in range(AVERAGE_RUN):
            start = time.perf_counter()
            run_threads(threadNum)
            end = time.perf_counter()
            duration = (end - start) * 1000000000

            durations.append(duration)

            print(f"Run {round} with {threadNum}: {duration} nanosec", file=sys.stderr)
        average = np.average(durations)
        std= np.std(durations, ddof=1)
        print(f"{threadNum}, {average}, {std}")
```

Appendix B
Poobah - C

Number of Threads	Average Runtime	Standard Deviation

1	3136171	170853.9997
2	3063011	27830.97872
3	3121849	24846.43061
4	3177872	24807.54413
5	6129589	79345.49834
6	6206800	39713.38692
7	6248456	29369.3521
8	7012624	845629.7967
9	9064931	385458.2172
10	9311140	13433.0968
11	9420675	57190.06382
12	9548087	40122.24414
13	12304310	23587.21039
14	12386987	60088.54359
15	12518889	49426.52532
16	12656321	25622.24559
17	15414941	45157.11637
18	15529999	19957.36273
19	15683981	82184.29659
20	15847386	31057.96562
21	18538799	23787.72494
22	18648851	69868.38355
23	18814268	144361.9437
24	19029441	38312.77038
25	21666992	24190.32003
26	21752103	56034.21592
27	21871788	83323.33667
28	22200467	70156.12139
29	24759091	34450.0588
30	24833058	31220.91046
31	25020079	34763.92058
32	25292877	42957.37054

Poobah - Java

Number of Threads	Average Runtime	Standard Deviation
1	3821669.6	1737209.329
2	3482450.4	1868962.723
3	2742531	76023.76286
4	3208534.2	1104172.278
5	5333329.4	103126.5885
6	6229468	1703053.546
7	6861214.6	1728738.323
8	6942095	1165499.999
9	8268458.4	267257.8546
10	9424654	1135461.538
11	9150691.8	1233394.708
12	11235422	1167412.952
13	10952436.8	1061554.245
14	10899153.4	749808.5959
15	11828687	1991154.812
16	11276397.8	1087565.03
17	13225401.8	376561.3132
18	15366630.4	1895762.847
19	14792973.8	1324500.426
20	15147171	1681002.147
21	16203820.8	715527.493
22	17754606	1399451.989
23	18873653.8	2439024.523
24	17542111.8	1630108.4
25	19044555.2	976775.2566
26	21991724.4	1524413.4
27	20911865.6	2492774.682
28	23059592	1541886.704
29	21364069.2	738090.4578
30	23549764.8	854857.7269

31	23129668.6	1201278.727
32	25865830.2	2281094.59

Poobah - Python

Number of Threads	Average Runtime	Standard Deviation
1	79154848	2082499.961
2	147221542	3884453.214
3	222227804.6	3336585.722
4	293679179.2	1016181.945
5	372050351.2	6065385.419
6	443438126.4	5354934.591
7	514362242.4	3287512.302
8	592821215	3544892.363
9	657916983.4	7611398.379
10	740020408.8	7169105.637
11	811203064.8	7824278.384
12	883486716.4	11126417.06
13	959382305.2	7180514.457
14	1034612584	9896931.871
15	1119837929	11986908.75
16	1182902030	15296696.49
17	1266467944	10666293.15
18	1345786525	14233744.01
19	1412879986	8874925.493
20	1485196179	13473013.34
21	1567391464	6563566.05
22	1638365596	10360073.55
23	1703677403	4034754.455
24	1786297466	6114555.414
25	1874759141	17221943.2
26	1936563520	9612700.12
27	2011671836	10817864.79

28	2080284978	13860052.04
29	2168544728	18840348.42
30	2225219215	19127618.79
31	2298896776	13782012.75
32	2364307147	7786744.727

Macbook - C

Number of Threads	Average Runtime	Standard Deviation
1	3136171	170853.9997
2	3063011	27830.97872
3	3121849	24846.43061
4	3177872	24807.54413
5	6129589	79345.49834
6	6206800	39713.38692
7	6248456	29369.3521
8	7012624	845629.7967
9	9064931	385458.2172
10	9311140	13433.0968
11	9420675	57190.06382
12	9548087	40122.24414
13	12304310	23587.21039
14	12386987	60088.54359
15	12518889	49426.52532
16	12656321	25622.24559
17	15414941	45157.11637
18	15529999	19957.36273
19	15683981	82184.29659
20	15847386	31057.96562
21	18538799	23787.72494
22	18648851	69868.38355
23	18814268	144361.9437
24	19029441	38312.77038

25	21666992	24190.32003
26	21752103	56034.21592
27	21871788	83323.33667
28	22200467	70156.12139
29	24759091	34450.0588
30	24833058	31220.91046
31	25020079	34763.92058
32	25292877	42957.37054

Macbook - Java

Number of Threads	Average Runtime	Standard Deviation
1	3821669.6	1737209.329
2	3482450.4	1868962.723
3	2742531	76023.76286
4	3208534.2	1104172.278
5	5333329.4	103126.5885
6	6229468	1703053.546
7	6861214.6	1728738.323
8	6942095	1165499.999
9	8268458.4	267257.8546
10	9424654	1135461.538
11	9150691.8	1233394.708
12	11235422	1167412.952
13	10952436.8	1061554.245
14	10899153.4	749808.5959
15	11828687	1991154.812
16	11276397.8	1087565.03
17	13225401.8	376561.3132
18	15366630.4	1895762.847
19	14792973.8	1324500.426
20	15147171	1681002.147
21	16203820.8	715527.493

22	17754606	1399451.989
23	18873653.8	2439024.523
24	17542111.8	1630108.4
25	19044555.2	976775.2566
26	21991724.4	1524413.4
27	20911865.6	2492774.682
28	23059592	1541886.704
29	21364069.2	738090.4578
30	23549764.8	854857.7269
31	23129668.6	1201278.727
32	25865830.2	2281094.59

Macbook - Python

Number of Threads	Average Runtime	Standard Deviation
1	79154848	2082499.961
2	147221542	3884453.214
3	222227804.6	3336585.722
4	293679179.2	1016181.945
5	372050351.2	6065385.419
6	443438126.4	5354934.591
7	514362242.4	3287512.302
8	592821215	3544892.363
9	657916983.4	7611398.379
10	740020408.8	7169105.637
11	811203064.8	7824278.384
12	883486716.4	11126417.06
13	959382305.2	7180514.457
14	1034612584	9896931.871
15	1119837929	11986908.75
16	1182902030	15296696.49
17	1266467944	10666293.15
18	1345786525	14233744.01

19	1412879986	8874925.493
20	1485196179	13473013.34
21	1567391464	6563566.05
22	1638365596	10360073.55
23	1703677403	4034754.455
24	1786297466	6114555.414
25	1874759141	17221943.2
26	1936563520	9612700.12
27	2011671836	10817864.79
28	2080284978	13860052.04
29	2168544728	18840348.42
30	2225219215	19127618.79
31	2298896776	13782012.75
32	2364307147	7786744.727

Windows 11 - C

Number of Threads	Average Runtime	Standard Deviation
1	2002880	209520.851
2	1963020	253065.9539
3	1886980	74210.25536
4	1949820	75598.92195
5	2124120	275192.8451
6	2014640	103861.7013
7	2022640	226492.6334
8	2171960	375219.9528
9	3479380	132264.1183
10	3644340	341659.7328
11	3537600	91553.56356
12	3581320	159649.2624
13	3581280	81636.67681
14	3648340	136375.0087
15	3616300	134287.8252

16	3747560	190976.0011
17	4850240	97952.86111
18	5065880	225281.5172
19	5546300	653414.7726
20	6003260	641333.0515
21	5766400	628746.9801
22	6237680	567530.1684
23	6568700	107833.135
24	6739500	215257.4505
25	6811960	125922.6469
26	6922780	251212.444
27	6864580	132569.6873
28	6941520	261343.2073
29	6994480	95831.92057
30	7110840	215590.8231
31	7092900	45223.16884
32	7040480	152504.4655
33	7663960	312481.2042
34	8270440	214229.2767
35	8762480	572671.0155
36	9417520	363650.5493
37	9451340	487362.8556
38	9716540	450696.775
39	9646300	660719.6531
40	9550540	373440.5749
41	9933940	104692.1821
42	10197920	229496.4313
43	10173720	194169.4415
44	10246060	274662.7077
45	10415580	236405.6091
46	11075020	349591.4716
47	10929860	162009.1139

48	11076760	410649.5014
49	10969220	170212.432
50	11495080	283366.3653
51	11975300	427644.7357
52	12600340	497336.177
53	12455900	387604.1279
54	12976140	419957.0966
55	12810700	365445.7621
56	13057700	347921.9381
57	13117640	218439.1174
58	13164400	423955.0271
59	13672980	296574.8674
60	13680100	177299.0835
61	13744600	184899.8242
62	13834460	304148.8501
63	14310940	517887.061
64	14278500	310497.7214

Windows 11 - Java

Number of Threads	Average Runtime	Standard Deviation
1	1604480	660895.3601
2	1421480	59568.29694
3	1392880	78875.64263
4	1475360	54967.24479
5	1525500	63560.91566
6	1564980	185011.0997
7	1633140	228146.2974
8	2092940	469885.2977
9	3086800	1267560.172
10	2562000	153147.1025
11	2565640	212404.3502

12	2538260	64131.7628
13	2758060	337182.0918
14	2761760	153137.7256
15	2657140	129549.8668
16	3044540	235471.5333
17	3204460	247614.1111
18	3490080	82870.51345
19	3864200	207242.5753
20	4301680	670112.88
21	4110140	182110.4418
22	3968220	381107.1201
23	4655920	161915.4008
24	4645600	561626.2191
25	4636180	133364.1893
26	4708540	166586.8482
27	5006360	375943.6434
28	5030000	171788.6783
29	4932560	129189.2526
30	4952940	157108.523
31	5245680	451021.9695
32	5414140	228654.637
33	6219160	401784.2618
34	5683180	149849.8815
35	6019940	254750.6879
36	6412680	676154.4069
37	6430480	331554.6441
38	6572980	327875.879
39	6722240	780172.0727
40	6788740	138501.0217
41	6728320	236825.3935
42	7295740	293932.2762
43	7061700	130684.3717

44	7172920	115320.3885
45	7341620	114630.4802
46	7788500	527228.6364
47	8227960	285554.0369
48	8076640	300209.5318
49	8217260	542078.5257
50	8271120	287426.5767
51	8512320	329529.7437
52	8613480	386759.1718
53	8432060	313566.6484
54	8968660	54070.90715
55	9254680	428387.8231
56	9282040	320086.8991
57	9490460	489752.0219
58	9266180	70350.92039
59	9432680	175583.2196
60	9729440	353752.5661
61	9387720	166380.9995
62	12427860	6806601.955
63	9519620	272235.5359
64	10125920	412456.1274

Windows 11 - Python

Number of Threads	Average Runtime	Standard Deviation
1	33614300	800146.2586
2	67194300	735298.3748
3	101191160	1716164.54
4	134455620	1587576.474
5	167026540	1015605.186
6	200078560	1312274.406
7	234271400	1381279.246

8	265325540	964633.44
9	301093180	3475135.469
10	334241160	1317250.913
11	366588700	2290910.561
12	402288120	6877302.471
13	434657120	1122544.541
14	465901260	1576370.488
15	499012960	3786478.517
16	531773080	3023558.372
17	565712180	1233100.205
18	597944140	2148634.82
19	631894960	1811418.434
20	665676420	4949067.881
21	768865720	37358598.38
22	765401320	17303310.54
23	807704400	14685785.69
24	834683860	24005672.49
25	891127220	40673697.49
26	957198920	83061893.39
27	951533960	15681453.96
28	1042777940	75836998.81
29	983356060	7868167.979
30	1015047160	10584731.22
31	1046368480	5022063.932
32	1106988300	33084109.01
33	1163926280	58545703.65
34	1197404600	35926904.31
35	1258604520	81462831.09
36	1215766760	6444193.189
37	1234323860	5273325.052
38	1381136280	147388807.1
39	1370438700	27172522.21

40	1348636880	21145883
41	1381629620	19046505.12
42	1499377580	130823559.4
43	1473469400	36695669.24
44	1495246360	24953118.6
45	1570016740	37722161.88
46	1664382920	78682080.18
47	1683382420	38791897.24
48	1679897280	46974637.87
49	1675320280	21734874.68
50	1745486900	32384293.8
51	1828897580	88658033.95
52	1801779100	21558408.07
53	1799171520	56532887.41
54	1797912260	2260047.596
55	1831852960	5087731.352
56	1868599940	12337276.52
57	1897973080	7199944.447
58	1930557680	12575205.03
59	1959473560	9412234.374
60	2004703140	13936936.98
61	2026123480	7420350.14
62	2065393420	14064390.28
63	2172311840	65845761.67
64	2146749680	23701477.62