**PART 1**

1. Used thread count of 10 for this table

|  |  |  |  |
| --- | --- | --- | --- |
| **ARRAY\_SIZE (# of <int>elements)** | **Single Thread Time, t1 (ms)** | **Multi Thread Time, t10 (ms)** | **Speed up Factor, S (S = t1/t10)** |
| 10 | 0 | 56 | 0 |
| 100 | 0 | 112 | 0 |
| 1000 | 0 | 102 | 0 |
| 10000 | 2 | 94 | 0.021 |
| 100000 | 21 | 45 | 0.467 |
| 1000000 | 218 | 168 | 1.298 |
| 10000000 | 2467 | 1562 | 1.579 |

2. I was expecting quite a lot of speed up. Even using just 10 threads I expected speed up factor to be a factor of around 7. I thought sorting independently per thread and then combining into a merge would not take too long. I thought the sort algorithm would take the longest time and merging would not take too much time, but evidently, merging the threads took a lot of run-time.

3. I did not meet the speed up I expected. I postulate that the reason was because the algorithm to merge the multithread sorted arrays was very run-time heavy, and waiting for the threads to join took a lot of run-time. When using a single thread, the merge algorithm took O(2n) run-time, whereas the multithread merge algorithm took O(n^2) time.

4.

|  |  |
| --- | --- |
| **Thread Count (<int> # of threads)** | **Speed-Up Factor (S = T1/Tn)** |
| 2 | 1.514 |
| 3 | 1.532 |
| 4 | 1.532 |
| 5 | 1.6 |
| 6 | 1.5 |
| 7 | 1.453 |
| 8 | 1.363 |
| 9 | 1.25 |
| 10 | 1.208 |
| 11 | 1.034 |
| 12 | 1.053 |
| 13 | 0.986 |

It appears that to some extent adding more threads made run-time faster but eventually adding more threads are actually detrimental to run-time. Graphing this function has a maximum at thread count = 5.

**PART 2**

To get a PI value accurate to 3 decimal places (π ~ 1.314…), I need a SAMPLE\_SIZE between 7,000,000 and 10,000,000. In this same sample size range, with a thread count of 20, multi-threading is actually faster than single-threading. At 7,000,000 samples the speed up factor is greater than for 10,000,000 samples.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sample Size** | **Single-Thread Time (t1)** | **Multi-Thread (20 threads) Time (t20)** | **Speed up factor (S = t1/t20)** |
| 1,000 | 1 | 124 | 0.008 |
| 10,000 | 2 | 167 | 0.012 |
| 100,000 | 11 | 305 | 0.036 |
| 1,000,000 | 146 | 525 | 0.278 |
| 10,000,000 | 1393 | 1222 | 1.140 |
| 100,000,000 | 13279 | 6995 | 1.898 |

1. Splitting up work between threads is not nearly as simple as I thought it to be. It seems to only be effective when working with a lot of information, but otherwise for relatively smaller samples multithreading seems to be more detrimental than helpful. This makes sense as multi-threading is used in machine learning a lot and lots of data processing must be done to create models and predictions, and the datasets of these models are usually quite large.
2. There is a lot of implications when designing multi-threading solutions. First, there is a lot of room for error and debugging is very difficult. Second, unless shared variables are locked, they are open to getting incorrect values. Finally, when multi-threading you must think about how the algorithm will be implemented. For example, in the mergesort (PART 1) code, we had to think about the most efficient way to divide up array slices into the different threads, get their output, and then merge them together. In comparison, for single-threading, I did not have to think about how to divide up the array as the mergesort method takes care of that already.
3. I think I will probably need over 10^12 samples, given the run-time of just 10^8 samples, the run-time would tank at numbers as high as 10^12 or GREATER. I think the monte carlo method is a very intuitive and smart method to calculate PI at LOW accuracies. However, more and more samples are needed for higher accuracy estimations which increase run-time by a lot. Multi-threading does help but not too many threads can be recruited or else it has a more detrimental effect on run-time.

Monte Carlo is a smart method but there are better, faster methods to calculate PI. The Chudnovsky Algorithm has a time complexity of O(nlog(n^3)), which is a better run-time than Monte Carlo’s method O(n^2).

Another method is the Bailey-Borwein-Plouffe Formula that has a time complexity of O(nlognM(logn), where M(m) = O(mlogm \* 2^(logm))[2] which has a better time complexity than both Monte Carlo and Chudnovsky.

[0] Monte Carlo Method time complexity <https://www.geeksforgeeks.org/estimating-value-pi-using-monte-carlo/>

[1] Chudnovsky time complexity <https://www.geeksforgeeks.org/estimating-value-pi-using-monte-carlo/>

[2] Bailey-Borwein-Plouffe Formula time complexity <http://www.davidhbailey.com/dhbpapers/digits.pdf>