Parallelizing the Calculation of Pi

Oliver Boorstein
University of Oregon

October 29, 2025

1 Implementation

Coming up with a working algorithm was not a difficult part of this assignment. Neither was implementing OpenMP. However, determining how best to set up my job for Talapas was more interesting. I played around with a few different options, e.g. array jobs and varying CPUs per task. This allowed me more control over the whole process. I also used a separate Python helper file to perform some averaging on the CSV files produced by my pi.c program.

2 Estimations

Across the board, the esitmations of pi at 100 million steps were accurate to 9-10 digits, except for the Monte Carlo Method, which varied by slightly more for atomic writes compared to a critical section, 0.0006% and 0.01%, respectively (likely just noise from the randomness of the method). For integrations, estimations were also consistent across different numbers of threads.

3 Parallelization and Speed

Most interestingly, time in seconds and number of threads turned out to be positively correlated. As can be seen in Figure 1, for all parallelized calculations, the time to complete generally climbed in response to an increase in the number of threads. This result was far more exaggerated for the Integral + critical computations. Afterwards, I verified my results using individual by running indiviudal jobs 20 times each on 14 and 28 threads. These results are summarized in Table 1. The time taken on 14 and 28 threads line up well with the results seen on the bulk iteration through thread counts. We also see stark contrast between time for atomic and critical methods for inegration, but not for Monte Carlo.

3.1 1 Billion Steps

I attempted to run a job executing on 1 billion steps, iterating through different thread counts. However, for fear of clogging Talapas' pipeline, I avoided increasing the time limit on the jobs. These jobs were only able to complete on 1-4 threads, within the time limit, but these showed similar trends as the jobs with 100 million steps.

3.2 Potential Explanations

3.2.1 Correlation of Thread Count and Time

As for understanding this connection, the steady growth says to me that calculating pi is not enough work for each thread to be worth the overhead. Threads spend more time being initialized and cleaned up (or waiting to write to their shared variables) to make the parallel processing have a meaningful impact. Additionally, the compiler optimizations, with flag -O3, likely explain some of Serial's dominance.

3.2.2 Difference in Atomic vs Critical

Atomic and critical primarily differ in what they deem protected. Since atomic protects memory locations only, it leaves room for other threads to load the instructions necessary to perform the write (once it is their turn). Critical, on the other hand, protects an entire code region, preventing other threads from even beginning to act on the instructions gaurded by it. This clearly explains the vast difference between Integral + atomic and Integral + critcal. However, both Monte Carlo methods are still so similar. I imagine this is due to the only occasional (when $x^2 + y^2 < 1$) access that threads executing the Monte Carlo method need to shared memory. Comparitively, with integration every thread needs access to shared memory each iteration.

4 Conclusions

In summary, the calculation of π is too elementary to be worth parallelizing. The overhead necessary to manage the threads only increases the time taken to complete the steps. The atomic and critical directives also demonstrated their strengths and weaknesses, with atomic prevailing in the context of integration, and inconclusive results with Monte Carlo.

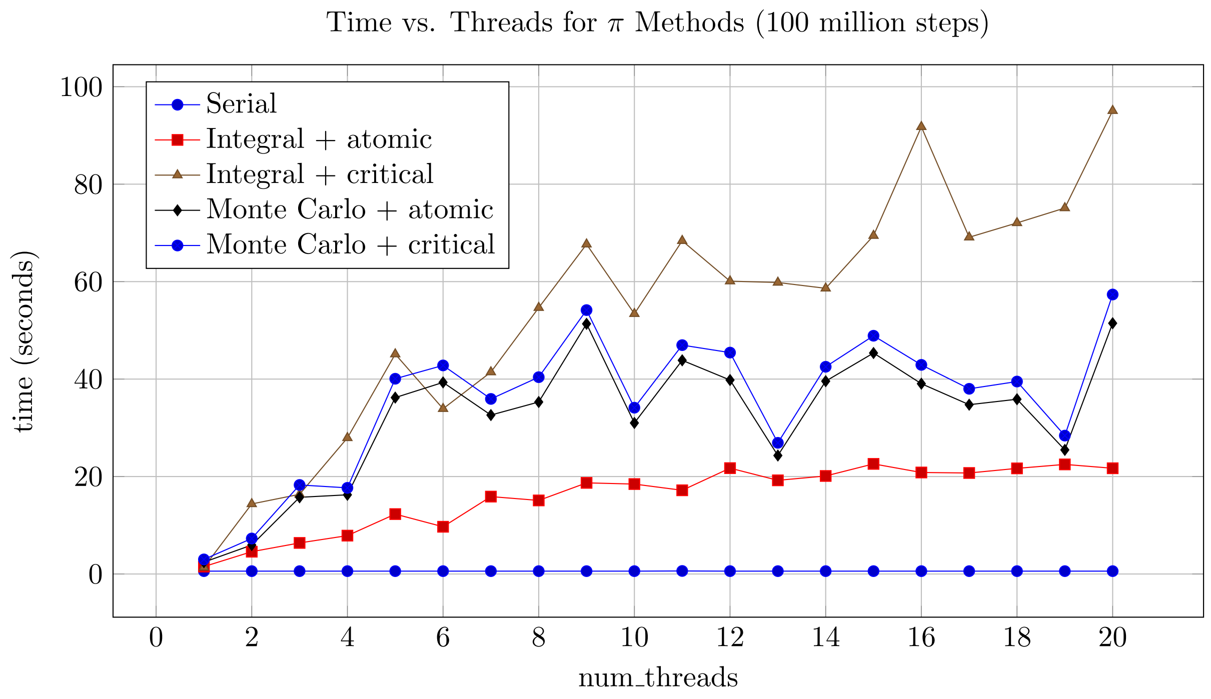

Figure 1: Runtime vs thread count across π implementations. Averaged over 3 runs for each thread count.

Table 1: Averaged timing results for π estimation at 14 and 28 threads (across 20 runs each).

Threads	Serial	Integral + atomic	Integral + critical	MC atomic	MC critical
14	0.5850	21.4103	61.3180	35.7606	38.8002
28	0.5863	21.7390	100.6184	39.1163	41.7586