Parallel Pi Calculation Analysis

Carmen Park

November 1, 2025

1 Introduction

This first assignment for CS431 involved computing π using three distinct computational approaches. These were serial numerical integration, parallel numerical integration, and parallel Monte Carlo estimation. Both were parameterized by a number of steps, which meant number of subdivisions for integration and number of random points for the Monte Carlo method. Our implementation of these methods extended the foundational code provided by Jee Choi, we implemented the core π calculation algorithms and OpenMP parallelization while leveraging the existing framework for performance testing and execution on Talapas[1].

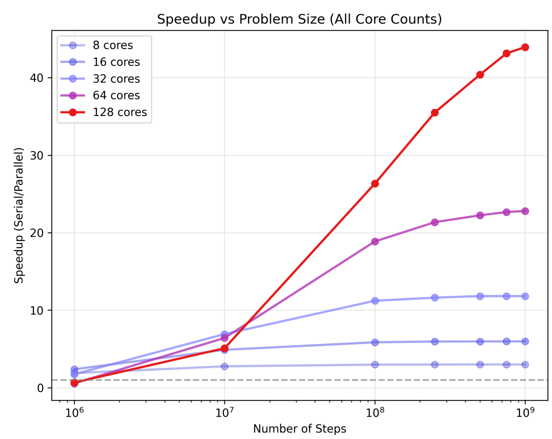
Sending batch requests to Talapas was very effective for large-scale testing. The original `pi.srun` configuration utilized 128 cores with 10^8 steps. For a breadth of information, I explored a range of steps from 1 million to 1 billion [2] and 8 to 128 cores [3]. This range emphasized the endpoints where parallel efficiency trends become more apparent, particularly when examining both computational limits and baseline performance. Logarithmic scaling was also used for all axes in diagrams to linearize power-law relationships and to be able to compare across such a wide range.

[1] Talapas is the University of Oregon's high-performance computing cluster.

[2] Step counts: 10^6, 10^7, 10^8, 2.5×10^8, 5×10^8, 7.5×10^8, 10^9

[3] Core configurations tested: $8, 16, 32, 64, 128$

2 Findings



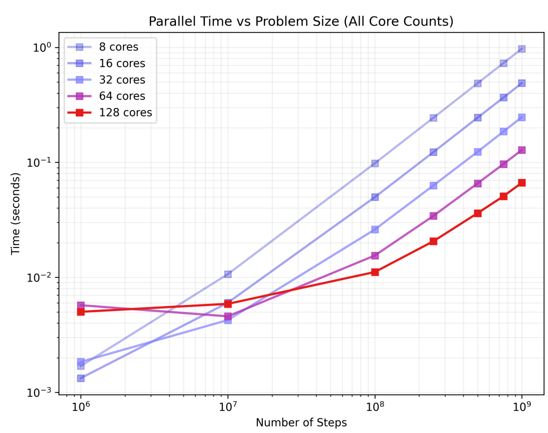
2.1 Speed Up over Cores

The comparison of serial integration to parallel integration revealed distinct scaling patterns across core configurations. Using Ahmdal's law, where

$$SpeedUp = \frac{T_{serial}}{T_{parallel}}$$

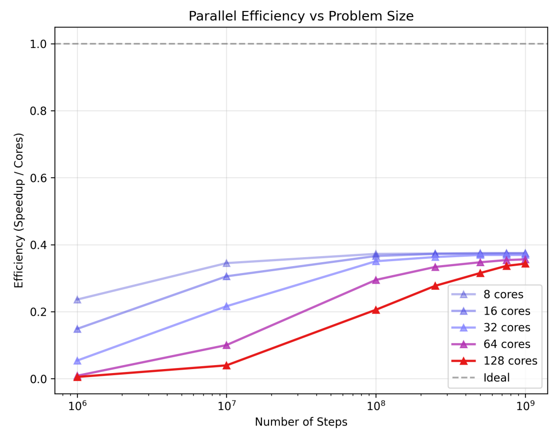
I was able to compare both methods. At 8 cores, parallel integration reached approximately $4X$ speedup regardless of problem size, while higher core counts had more complex behavior. 128 cores only began to outperform 64 cores in the 10^7 to 10^8 step range, similarly 64 cores only began to outperform 32 in the 10^7 range. This suggests that exists some threshold where increased parallelism only becomes advantageous after surpassing some number of steps. Similarly, once this number of cores out performs the former number exhibits diminishing returns only after nearly doubling the previous speedup. These are only observations, but I found this very compelling and would like to learn more about the physical limits of Amdhal's law.

2.2 Timing vs to Problem Size



In my opinion, the overall execution time on multiple cores showed the most predictable trends. The higher core counts had dramatic improvements for large problems but minimal benefit for smaller computations which was most likely due to costly overhead. The crossover points between core configurations highlight problem-size thresholds where additional parallelism becomes advantageous. I felt this was important in illustrating the balance between computational workload and parallel overhead.

2.3 Parallel Efficiency



This last observation on efficiency further emphasizes my overall findings. Efficiency reveals how effectively additional cores are utilized and is calculated as $E = SpeedUp/n$ where n is core count. The 8-core configuration maintains relatively stable efficiency across problem sizes, but higher core counts require substantially larger problems to achieve comparable efficiency. The 128-core configuration barely approaches peak efficiency at 10^8 steps, indicating significant parallel overhead for smaller problem sizes.

3 Limitations

Given more space, I would analyze accuracy-timing trade-offs between Monte Carlo and integration methods, rather than focusing primarily on core count scaling.