CS 431 Homework 01: Parallel Pi on OpenMP

David Haddad

October 30, 2025

1 Introduction

This report details the implementation and performance analysis of two parallel algorithms for calculating π using OpenMP. The first was the classic numerical integration method in which we calculated $4 \times (Area\ of\ \frac{1}{4}th\ circle)$. The second (embarrassingly parallel) method was the Monte Carlo method in which we calculate the proportion of randomly generated points that lie within/without the circle. The purpose of this assignment is to analyze the scalability and performance of these methods on the UO Talapas cluster. Throughout testing, the findings demonstrate key understandings of how synchronization strategies effect performance. The project itself illuminates the downsides of high-contention algorithms using a critical section and `atomic` and how that can actually cause negative speedup, while a `reduction`-based algorithm can achieve the expected linear speedup we want when scaling.

2 Methodology and Implementation

All code for the project was written in C and parallelized using OpenMP. The initial serialized approach which was never taking longer than 4s regardless of input size (up to $1B$). The theory was that the compiler was optimizing this code far past the current scope of the project. The `Makefile` compiler flags were altered from `-O3` (heavy optimizations) to the `-O0` flag (no optimization) to better demonstrate the true timing of the serialized approach as written in `pi.c`. However, the issue persisted. This "fake" baseline caused confusion and led to increasing the problem size to $100B$. Even this astronomically large input showed an unbelievable 6s for the serial approach. This meant that a $100x$ increase in size led to almost *no change* in execution time. Since this could not be true, the final data set reduced the input size to $1B$ and we ignored the "serial baseline", instead focusing results on the correlations between the parallelized methods.

2.1 P1: Parallel Integration with `atomic`

This algorithm is the same as the serial method, with added OpenMP declarations. The use of the `#pragma omp atomic` directive ensured that we avoided the race condition present without careful handling of the sum variable. This leads to serialization of this section of code, which creates a severe synchronization bottleneck.

2.2 P2: Parallel Monte Carlo with `reduction`

A common metaphor for this method is "dart throwing" in a loop and calculating the ratio of darts that "hit" and are `in_circle` as π. This algorithm parallelizes that "dart-throwing" loop. Since each iteration of the loop is simply selection of a random point and checking if it's in the circle, they could all be run in parallel at the same time. However, this leads to a race condition and a similar bottleneck on the `in_circle` variable if solved in the same way as **P1**. It was resolved using the OpenMP `reduction(+:in_circle)` clause, giving each thread it's own copy and eliminating the bottleneck/unpredictable output.

3 Results and Analysis

The data collected illustrates how different synchronization strategies for the same underlying problem can have drastic differences in performance.

3.1 Experimental Setup

The data collected and used for the following figures was run using a series of `.srun` files and the following loop for data collection:

```
for i in {1..20} do ./pi 1000000000 done
```

Afterwards, the averages of the "20" runs were calculated. Unfortunately, even with this input size, the Talapas cluster would cancel the jobs on the multi-threaded test scripts due to the **P1 Integration** method presenting negative performance scaling at a factor that caused the 4th or 5th iteration to reach the job `TIME LIMIT`. Thus the averages calculated in the figures do not truly represent 20 runs for each method as that would require higher permissions/usage limit on Talapas.

3.2 Analysis of P1 (Integration)

The P1 algorithm demonstrated severe **negative scaling**. The baseline 1-thread time (T_1) was **6.817 seconds**. As shown in Table 1, adding more threads ended up counterproductively and drastically *increasing* the runtime.

Table 1: P1 (*atomic*) Performance vs. Thread Count

Threads (p)	Avg. Time (s)	Speedup (T_1/T_p)
1 (Baseline)	6.817	1.00x
2	45.863	0.15x
4	94.485	0.07x
8	128.712	0.05x
16	121.172	0.06x

These counterintuitive results are an example of *high contention*. Instead of 16 threads running peacefully in parallel, the 16 threads end up waiting for access to the same `atomic` variable *1 billion* times! The cost to synchronize and waiting completely surpass any benefit that came from increasing the number of threads. For some reason, the 16 threads ran faster than 8, but this was actually unique to the final data set as compared to the failed attempts (see output directory). Also, due to jobs being canceled before completing 20 iterations, random scheduling variance may account for this minor outlier.

3.3 Analysis of P2 (Monte Carlo)

To demonstrate the contrast, the project suggested the use of a `reduction` clause for this *embarrassingly parallel* computation. In return, we see a almost perfect **linear speedup**. The baseline time (T_1) was **12.466 seconds**. This result shows how using the correct OpenMP directives for the structure of your problem (and having an easy problem to parallelize) can successfully, and nearly completely, eliminate performance bottlenecks.

Table 2: P2 (*reduction*) Performance vs. Thread Count

Threads (p)	Avg. Time (s)	Speedup (S_p)
1 (Baseline)	12.466	1.00x
2	6.238	2.00x
4	3.123	3.99x
8	1.563	7.97x
16	0.783	15.92x

Table 2 shows that as we doubled the threads, we consistently halved the runtime. These are the kind of consistent results that can be seen from successful and thoughtful parallelization of your code.

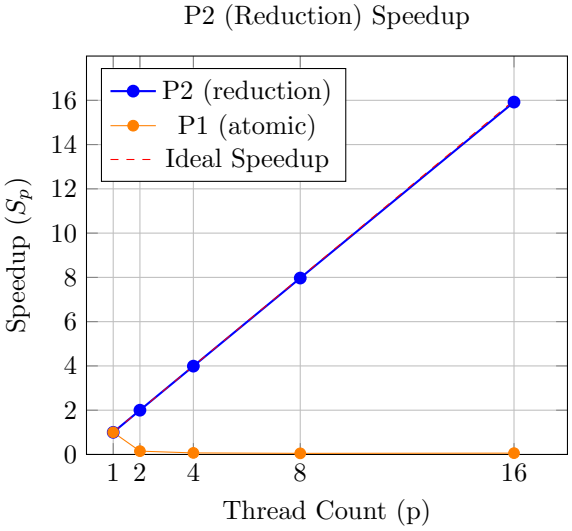

Figure 1: Linear Speedup (red, dashed) vs Thread Count for P1 (atomic) and P2 (reduction). The P2's actual speedup (blue) almost perfectly the ideal linear speedup. P1's performance, however, decreased with increased thread count.