Optimization of Matrix-Vector Multiplication using OpenMP in C

Introduction

In the realm of high-performance computing, efficient matrix operations are imperative to achieve high throughput and optimal system utilization. This study seeks to increase the performance of matrix-vector multiplication by leveraging the capabilities of OpenMP, a popular parallel programming model. The goal is to parallelize the matrix-vector multiplication operation with various OpenMP directives and measure their impact compared to an unoptimized version.

Methods Design Approach

The experiment is designed to compare the runtime and performance of multiple versions of matrix-vector multiplication: one without optimization (referred to as 'unoptimized') and others with various OpenMP implementations.

Experimental Setup

Optimization Level	**Description**
Unoptimized	Used as a control/benchmark
Opt1	`#pragma omp parallel num_threads(n)`
Opt2	`#pragma omp parallel for num_threads(8) private(k)`
Opt3	`#pragma omp parallel for schedule(dynamic, 8) private(k)`
Optimized	`#pragma omp parallel for private(k)`

The runtime, performance, and correctness of results are gauged for each version using the microtime tool.

Evaluation Strategy

The criteria for the evaluation are:

1. Average T_{serial} (microseconds).
2. $T_{parallel}$ (microseconds).
3. Speed Up (S).

4. Efficiency (E).
5. Correctness of results (C[N/2]).

Implementation Details

The implementation with OpenMP required introducing various pragma directives to enable parallel computation of the matrix-vector multiplication. My first challenge was found in `opt1.c`, when my results were incorrect due to unhandled race conditions. To ensure that the shared and private variables were correctly protected, I used the `private` pragma directive to protect the inner loop variable, ensuring that the algorithm's parallel sections did not lead to data races. After that, it was just a test of using different directives and seeing which had the most notable impact. Notably, the most performant version I used was:

```
#pragma omp parallel for private(k)
```

This version did not take a thread number at all yet yielded the most impressive results. Unfortunately I couldn't chart it because I could not control the number of processes, but the speed-up is significant.

Results

Charts can be found starting on page 3. All table data can be found here.

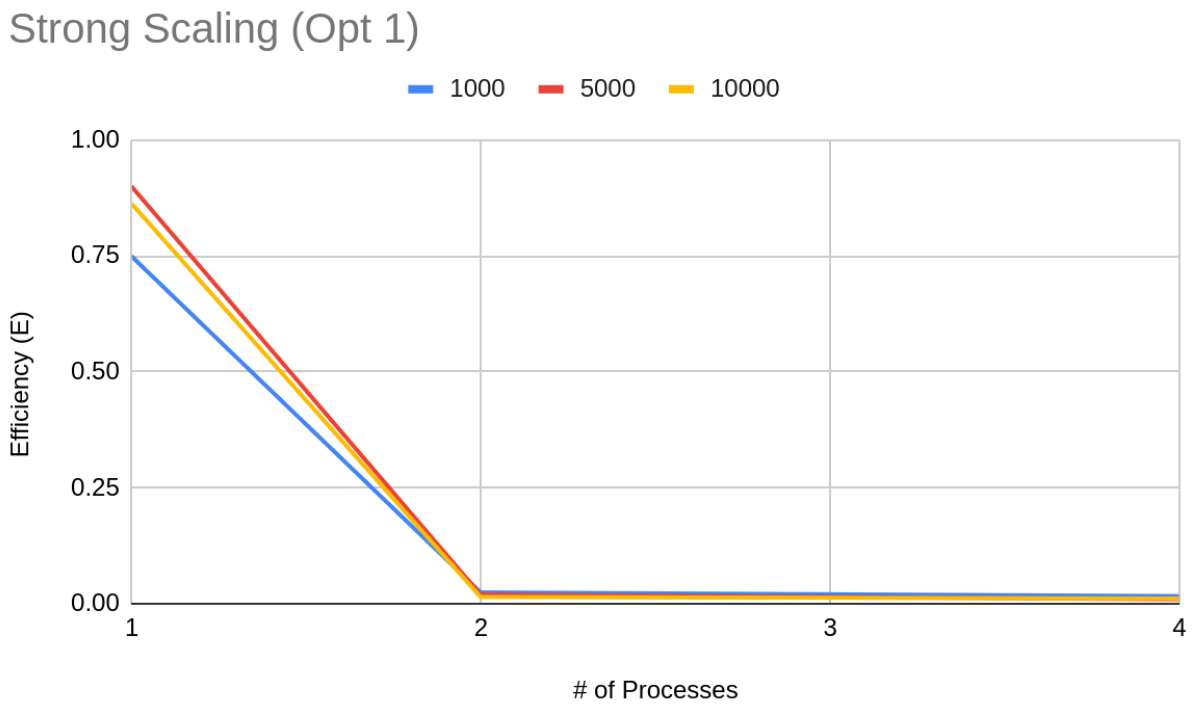
Note to the professor: I understand that speed-up and efficiency values greater than 1 should not be possible, but every test I ran returned the same results. Below are the charts containing my data. You can also find my data, charts, and pivotal tables here. This may be due to my hardware, or perhaps the pragma directive is improving things beyond just parallelization such as algorithmic improvements.

Discussion

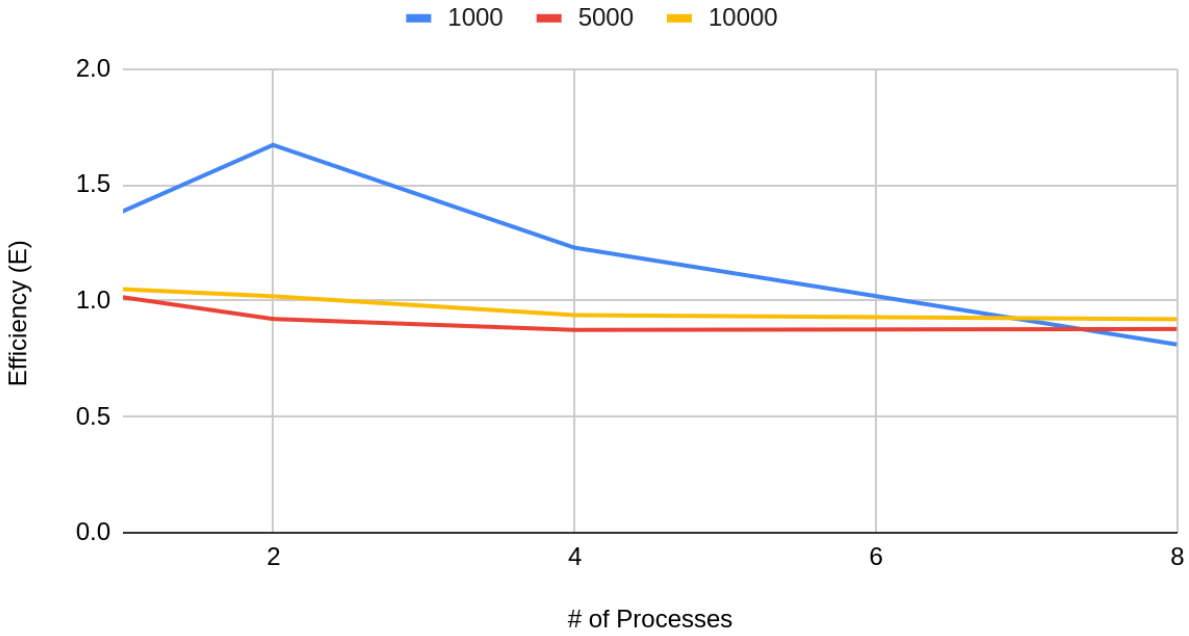
1. By leveraging OpenMP, the matrix-vector multiplication showcased a substantial improvement in performance with nearly every type of directive used. This demonstrates the strength of parallelism in such computations.
2. The most optimized version of the code used OpenMP without choosing a number of cores. This may be due to my computer having 32 cores and the default OpenMP might be taking advantage of all 32.
3. My data consistently resulted in Efficiency values greater than 1. This suggests that there is some kind of beneficial hardware or software my computer has that aids during certain problem sizes.
4. Pragma directives do not provide performance freely - sometimes the overhead is greater than the gains from parallelization, and unprotected variables may lead to race conditions and incorrect results.

Conclusion

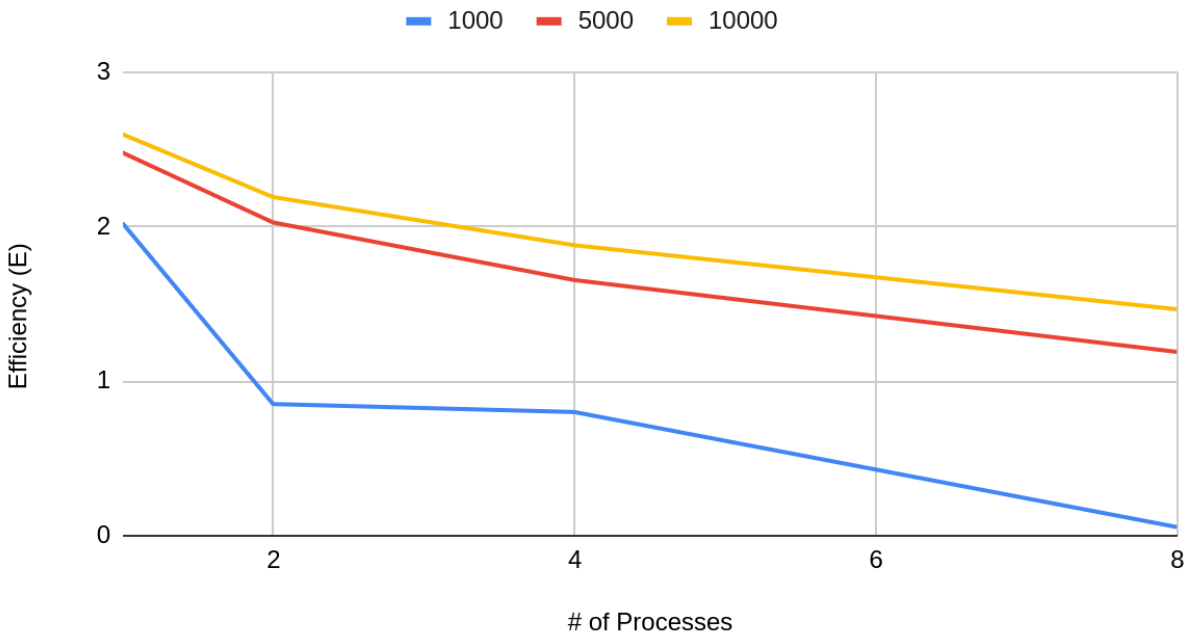
Parallelizing matrix-vector multiplication using OpenMP showcases the substantial potential gains in performance. The study reaffirmed the significance of parallel computation, especially in scenarios like matrix operations, where data can be processed concurrently. Although my efficiency numbers were unrealistic (with values greater than 1), this still demonstrates how impressive the performance gains are from pragma directives.



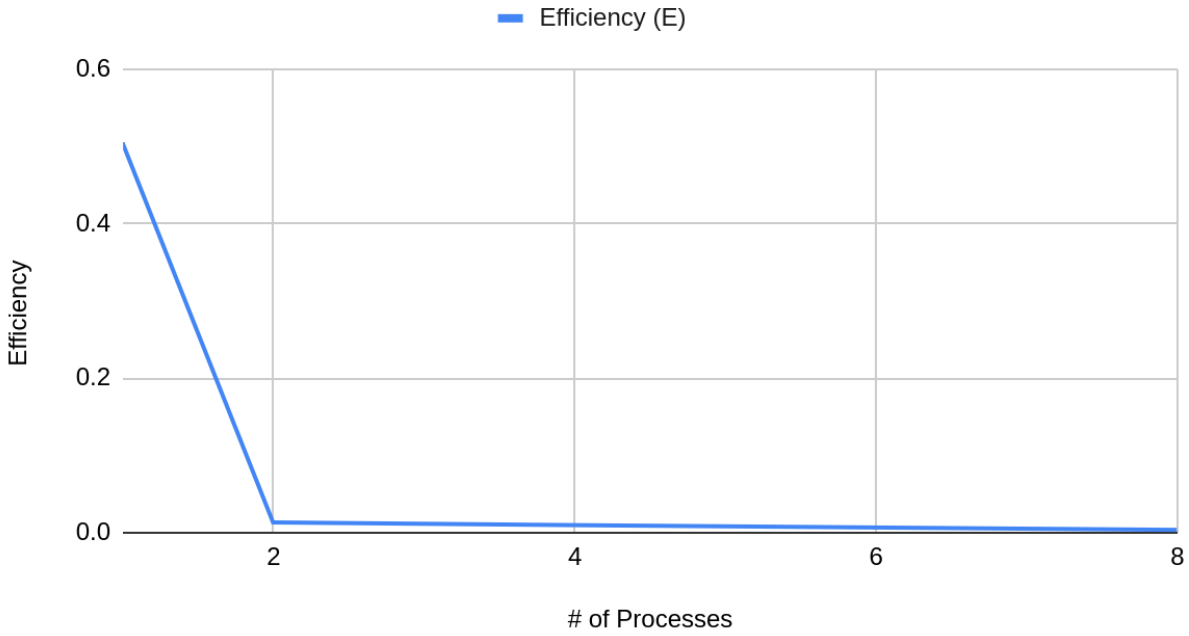
Strong Scaling (Opt 2)



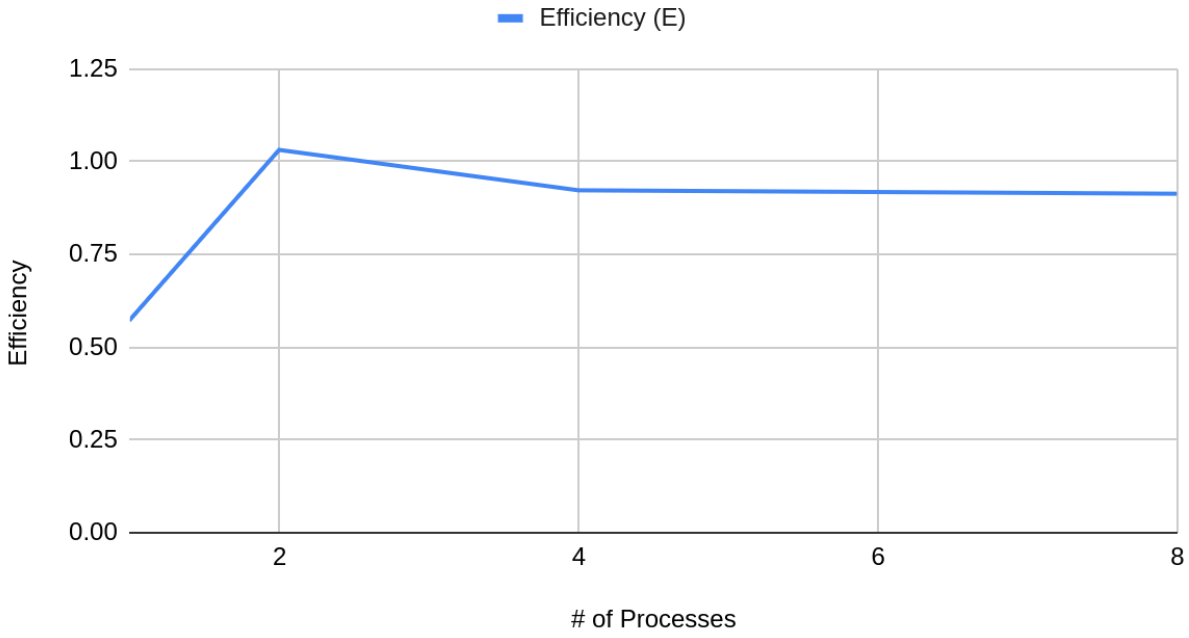
Strong Scaling (Opt 3)



Opt 1 - Weak Scaling



Opt 2 - Weak Scaling



Opt 3 - Weak Scaling