Sequential vs. Parallel Vector Summation

Jack Rowe (jbr2826, jackrowe)

March 2024

1 Introduction

This algorithm is programmed in C++ using OpenMP. The goal is to determine the execution time for various numbers of threads and vector sizes. The project features two executables: `vectorsum` and `affinity_vectorsum`. They both output execution times, however the latter uses an interpolation function to determine when to switch from sequential to parallel given the number of threads. Additionally, a shell script was used to collect and output the data, and a Python script using Matplotlib was used to create organized visualizations.

2 Comparing Methods

2.1 Parameters

There two parameters to vary are the number of threads and the vector size. Executing the shell script will run `vectorsum` with thread sizes (8, 16, 32) and vector sizes [1, $10,000$) in increments of 25. Running the python script will then generate the visualizations as shown in Figure 1.

2.2 Summary

For 8 and 16 threads, sequential computation is generally faster at low values up until a certain threshold (yellow dot). This is probably due to the extra time needed to assign tasks to threads. Afterwards, the parallelized code performs much better. Occasionally, the parallelized code will vastly underperform, likely due to a single thread receiving multiple of the larger tasks. 32 threads is larger than the `vm-small` core count of 16, which was intentionally chosen to show the extra time needed to manage extra threads.

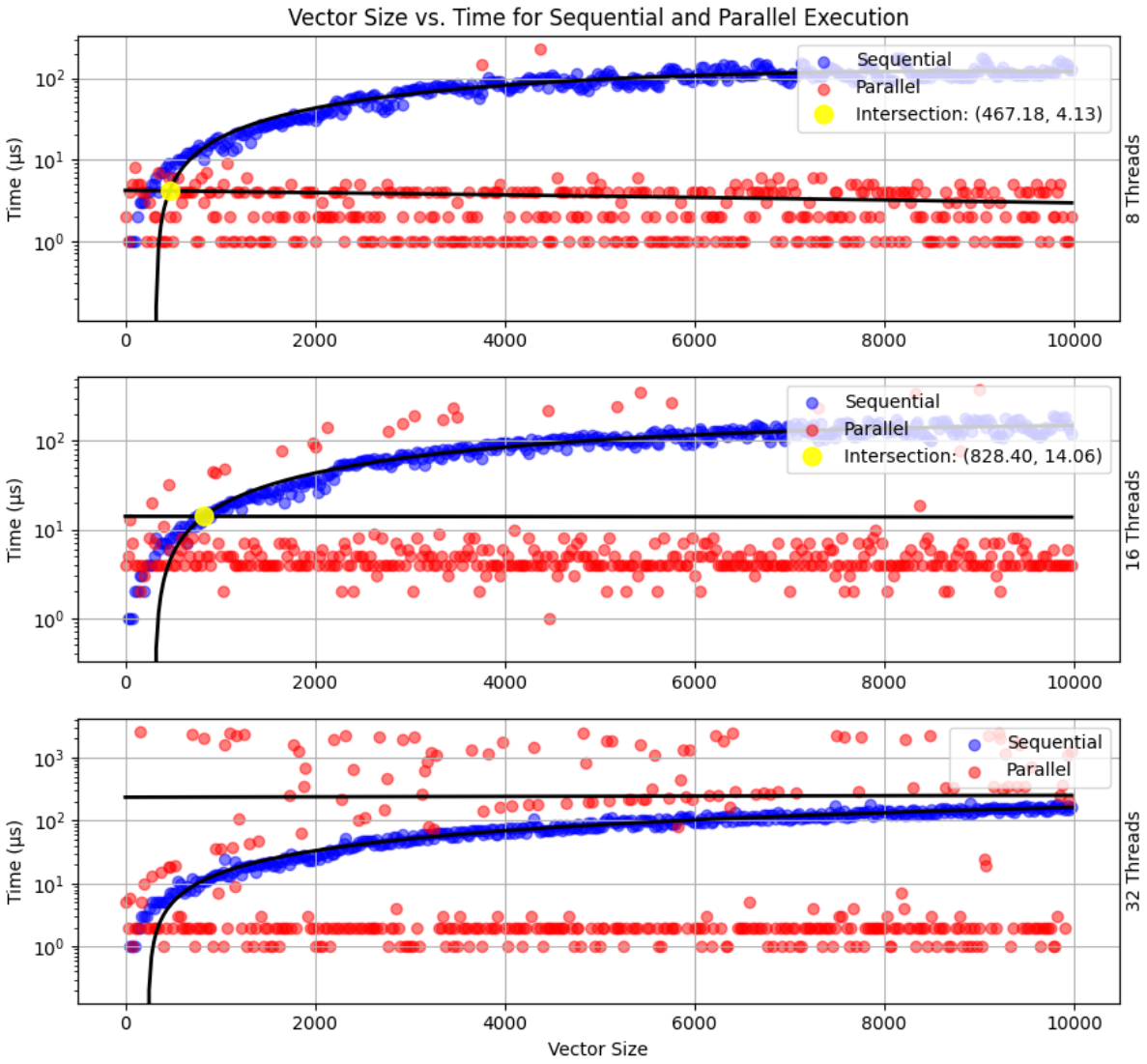

Figure 1: Sequential vs. Parallel Time Graphs

3 Affinity Implementation

Analyzing the data shows two intersection points at (467.18, 4.13) and (828.40, 14.06). Using their associated thread counts of 8 and 16, it is possible to dynamically determine which trendline is smaller given a certain number of threads. In this case `if(vectorsize > affinity_cutoff)` serves as a toggle in the `#pragma` line. Figure 2 shows the parallel times matching the sequential times up until a precomputed intersection point. A snippet containing the interpolation function is also below.

```
int interpolate_affinity_cutoff(int nthreads) {
  int x1 = 8, y1 = 467;
  int x2 = 16, y2 = 828;

  return y1 + ((y2 - y1) / (x2 - x1)) * (nthreads - x1);
}
```

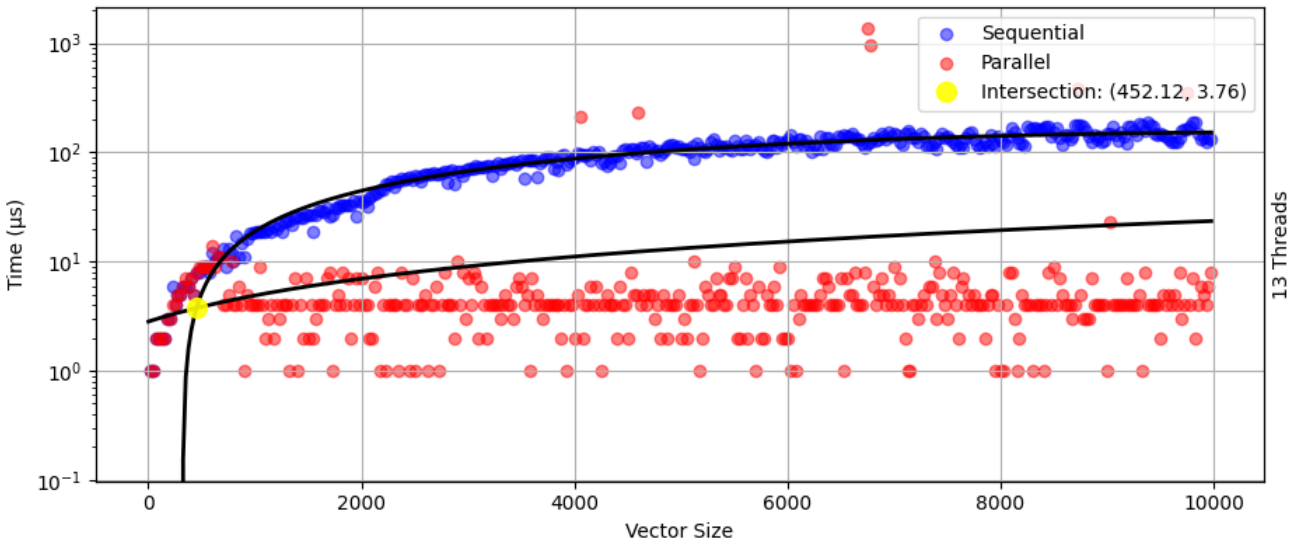

Figure 2: Sequential vs. Affinity Parallel Time Graph

4 Bandwidth

Multithreading bandwidth is an estimate of the amount of work done per thread, and can be calculated according to the following equation.

$$Bandwidth = \frac{Work}{NumThreads \cdot ExecTime}$$

We know that at the intersection point in Figure 2, A vector size of 452 on 13

threads results in an execution time of roughly $3.76\mu s$. The work done can be totaled to $500 \cdot VectorSize$ according to the following 2D loop.

```
for (int iloop = 0; iloop < 500; ++iloop) {
      for (int i = 0; i < vectorsize; ++i) {
        outvec[i] += invec[i] * factor;
      }
      factor /= 1.1;
    }
```

As such, the bandwidth roughly comes out to $4.62 \cdot 10^3 \frac{Operations}{Thread \cdot \mu s}$