CMSC416: Assignment 1 Report

Krisnajit Rajeshkhanna

September 2024

1 Problem 1: Vector Minimum Distance

1.1 Solution Approach

In order to parallelize this program, we see that the value we would like to aggregate is `min_dist` across all the parallel regions. We can use the OpenMP directive `reduction(min:min_dist)` as this will essentially aggregate the minimum of all minimum distances after all parallel regions complete.

1.2 Performance Analysis & Testing

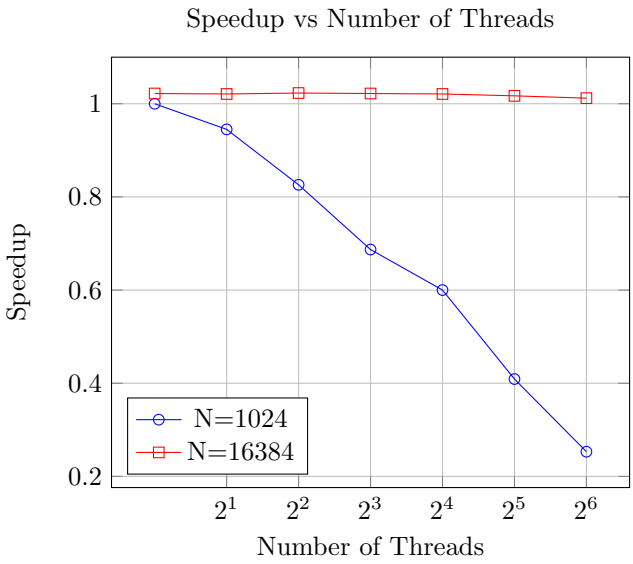

Figure 1: Speedup vs Number of Threads for Small and Large Inputs

With a small input, the serial program had an average run time of 0.00174 s. On the other hand, the parallel program started at an average run time of 0.00171 s

with 1 thread and ended at 0.00675 s with 64 threads. As represented by the graph, the run time of the parallel program decreased in run time as the number of threads increased due to the overhead of having too many threads executing the same task.

With a large input, the serial program had an average run time of 0.439 s. On the other hand, the parallel program stayed pretty consistent at around 0.43 s for $1, 2, 4, 8, 16$ threads, but started to increase to around 0.434 for 64 threads. As represented by the graph, the run time of the parallel program was consistently faster than the serial program, but slightly started to dip in speedup from 32 and more threads.

2 Problem 2: Number of Edges in DG

2.1 Solution Approach

In order to parallelize this program, we see that the value we would like to aggregate is `count` across all the parallel regions. We can use the OpenMP directive `reduction(+:count)` as this will essentially aggregate the sum of all edge counts after all parallel regions complete.

2.2 Performance Analysis & Testing

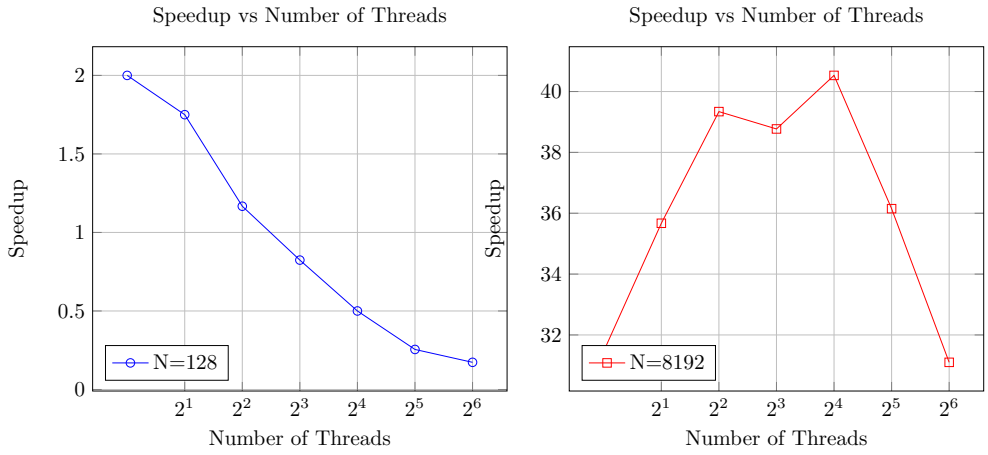

Figure 2: Speedup vs Number of Threads for Small and Large Inputs

For these graphs, we observe something similar yet very different when tackling the problem of counting edges.

With a small input, the serial program had an average run time of 0.00014 s. On

the other hand, the parallel program started at an average run time of 0.00007 s with 1 thread and ended at 0.00081 s with 64 threads. As represented by the graph, the run time of the parallel program decreased in run time as the number of threads increased due to the overhead of having too many threads executing the same task.

With a large input, the serial program had an average run time of 0.267 s. On the other hand, the parallel program run time decreased pretty consistently from 0.085 s to 0.066 s for $1, 2, 4, 8, 16$ threads, but started to increase to around 0.08 for $32, 64$ threads. As represented by the graph, the run time of the parallel program was consistently faster than the serial program, but peaked in speedup at around 16 threads.

3 Problem 3: Unique Product of Elements

3.1 Solution Approach

In order to parallelize this program, we see that the value we would like to aggregate is `count` across all the parallel regions. We can use the OpenMP directive `reduction(*:count)` as this will essentially aggregate the unique product of the vector elements after all parallel regions complete.

3.2 Performance Analysis & Testing

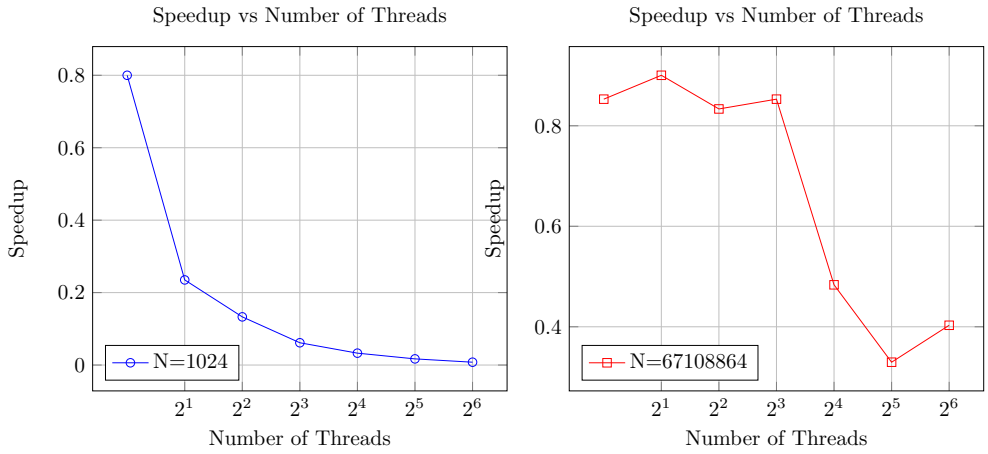

Figure 3: Speedup vs Number of Threads for Small and Large Inputs

For these graphs, we observe something quite different when tackling the problem of multiplication and division.

With a small input, the serial program had an average run time of 0.00004 s. On the other hand, the parallel program started at an average run time of 0.00005 s with 1 thread and ended at 0.00512 s with 64 threads. As represented by the graph, the run time of the parallel program decreased in run time as the number of threads increased due to the overhead of having too many threads executing the same task. Note that the speedups were exponentially approaching an asymptote at 0 indicating that with $N \to \infty$, there would essentially be no difference between using the serial code and the parallel code for a smaller dataset.

With a large input, the serial program had an average run time of 0.0712 s. On the other hand, the parallel program run time hovered between 0.08 s to 0.09 s for $1, 2, 4, 8$ threads, but started to increase to around $0.15 - 0.2$ s for $16, 32, 64$ threads. As represented by the graph, the run time of the parallel program was consistently faster than the serial program, but tanked sharply in speedup at around 16 threads and higher.

4 Problem 4: Discrete Fourier Transform

4.1 Solution Approach

In order to parallelize this program, we see that the value we would like to consider is `theta` across all the parallel regions. This is not an aggregation like the previous problems, but rather we are focused on avoiding data races when reading and writing to theta as it serves as an intermediate value when calculating the transformation. We can use the OpenMP directive `private(theta)` as this will essentially allow for each thread to read and write values to its own `theta` to avoid data races.

4.2 Performance Analysis & Testing

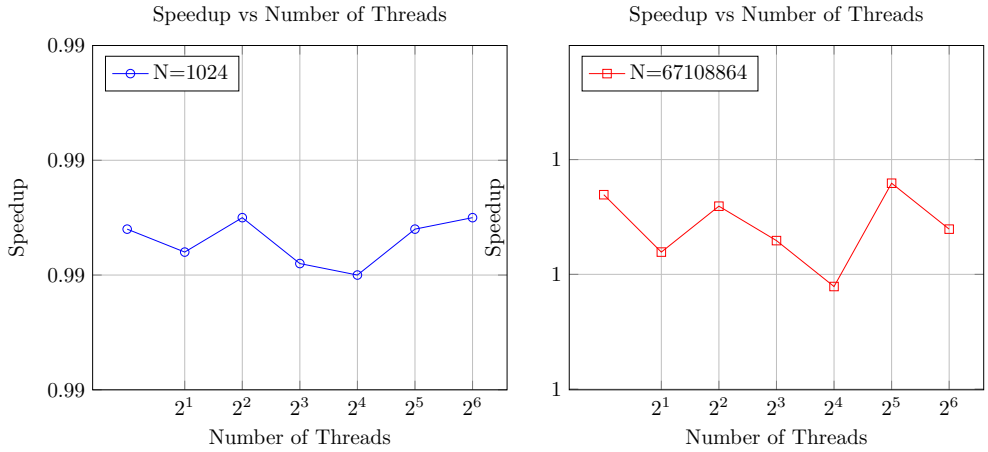

Figure 4: Speedup vs Number of Threads for Small and Large Inputs

For these graphs, we observe something completely different as when observed at close they look very staggered. However, we note that both graphs roughly have a range of 0.001. Therefore, the speedups are very similar to one another yet up close vary ever so slightly.

With a small input, the serial program had an average run time of 0.0415 s. On the other hand, the parallel program started at an average run time of 0.418 s with 1 thread and ended at 0.417 s with 64 threads. We need to see that the graphs show very small fluctuations in speedup as they are all pretty consistent with one another in value.

With a large input, the serial program had an average run time of 2.61 s. Once again, the parallel program run time hovered between 2.59 s to 2.6 s showing not much difference in time. Seeing as if we are seeing absolutely no difference in run time or speedup after changing the size of the dataset, we notice that the main difference between this and the other problems is the OpenMP directive used. Since we are using `private()` and not a `reduction(:)`, we are just trying to avoid a data race in a simple calculation, so there may not be much improvements in time if the program is trying to run more threads on a simpler task than necessary.