Various Parallelization Implementations for Jacobi Method

Jack Rowe (jbr2826, jackrowe)

March 2024

1 Overview

This algorithm is programmed in C++ using OpenMP. The goal is to determine the execution time for various numbers of threads and matrix sizes. The project features one executable (`jacobi`), however there is an additional `outer_jacobi.cxx` file that parallelizes the outer for loop. Additionally, a shell script was used to collect and output the data, and a Python script using Matplotlib was used to create organized visualizations. Output text files used for the visualizations have been saved in this repository.

2 Exercise 8

2.1 Initial Parallelization

On the initial pass, the code was parallelized using `#pragma omp parallel for` on each for loop inside the outer for loop. Notably, the memory assigning loop was not parallelized in this step (see first-touch subsection). Additionally, Figure 2 and Figure 3 show that the precision and iteration values are not affected by thread count. All graphs have a log-scaled x-axis, as samples were taken every 3rd power of 2.

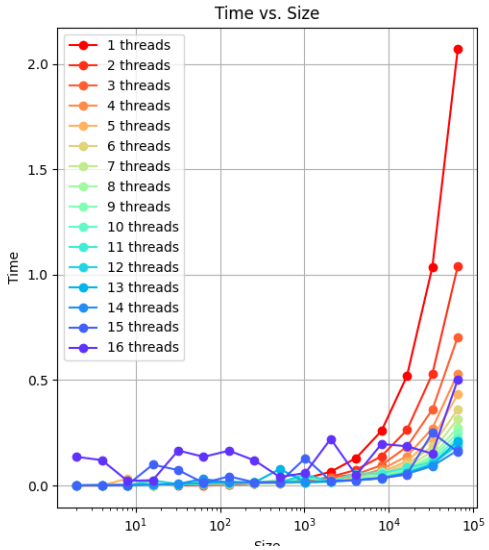

Figure 1: Thread Counts on Execution Time vs. Matrix Size

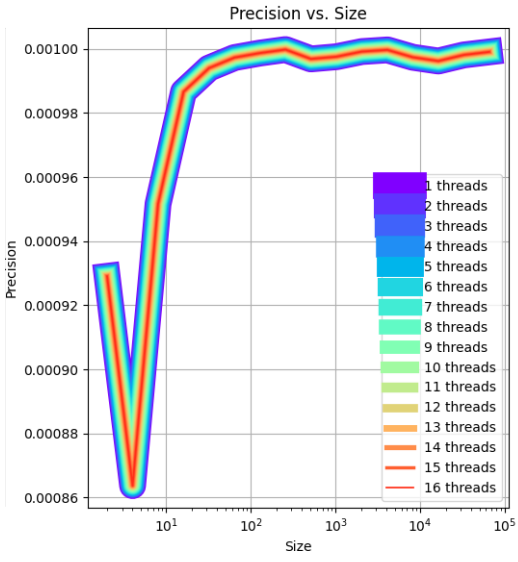

Figure 2: Thread Counts on Precision vs. Matrix Size

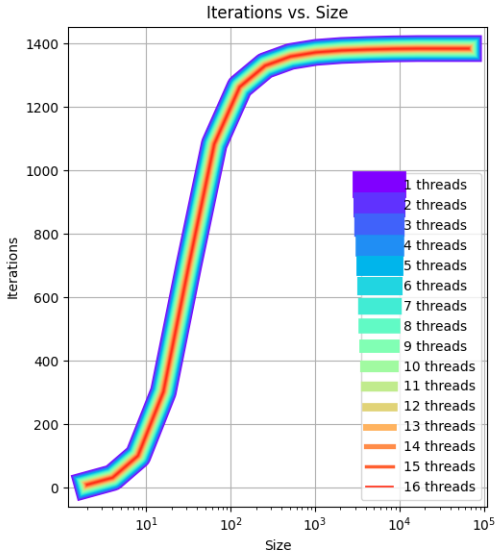

Figure 3: Thread Counts on Iterations vs. Matrix Size

3 Exercise 9

3.1 Implicit Barrier Removal

By default, OpenMP places an implied barrier after `#pragma omp for` sections. However, this can be removed by appending `nowait` at the end of the statement, as long as the threads do not need to synchronize after the loop. This did not provide a noticeable decrease in execution times.

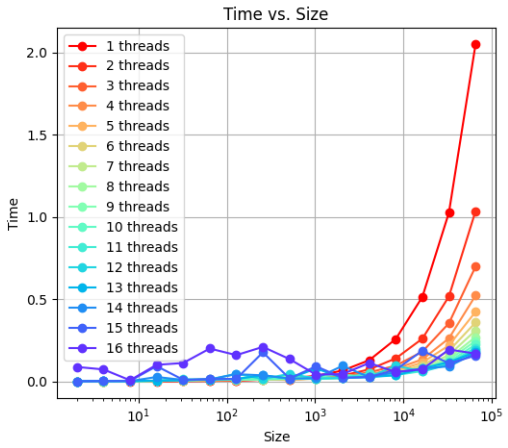

Figure 4: Thread Counts on Execution Time vs. Matrix Size (nowait)

3.2 Affinity

Next, the shell script was modified to use `OMP_PROC_BIND=True` in the run command. This sets the proper affinity, and shows a noticeable decrease in execution time. (I did not realize that I had a completely incorrect understanding of affinity on the last assignment - oops).

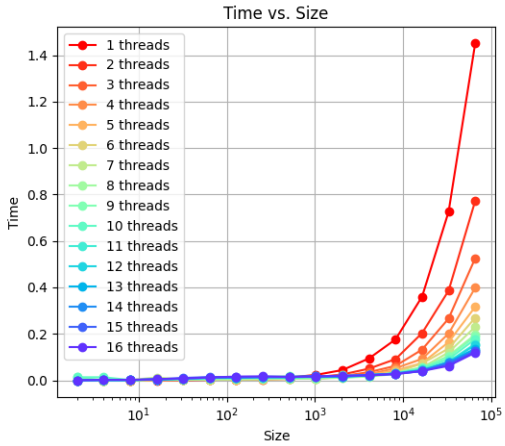

Figure 5: Thread Counts on Execution Time vs. Matrix Size (Affinity)

3.3 First-Touch

First-touch memory assignment was implemented next. This was done by assigning a `#pragma omp parallel for` statement above the loop. However, I am not entirely certain that this is proper implementation, given that `schedule()` isn't specified here. See next section.

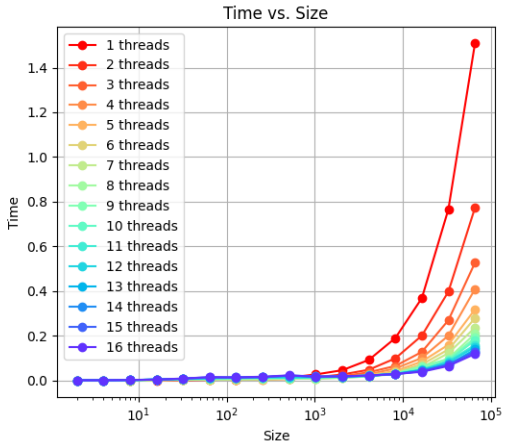

Figure 6: Thread Counts on Execution Time vs. Matrix Size (First-Touch)

4 Exercise 10

4.1 Scheduling

Since the work being done is fairly uniform in size (looking at each for loop shows `vectorsize` length iterations), I concluded that static scheduling would probably work best. Using the default small chunk size of 1 does not result in much noticeable decrease in execution time. Scheduling also helps to ensure that first-touch is implemented safely.

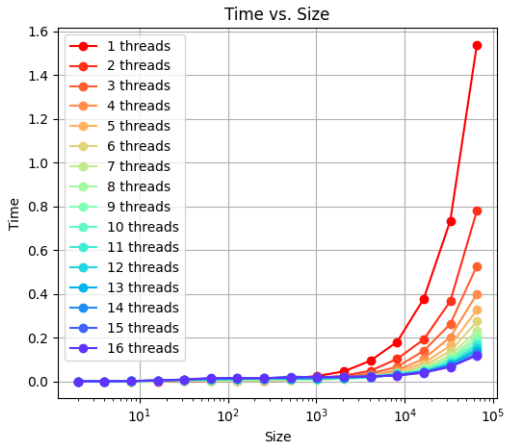

Figure 7: Thread Counts on Execution Time vs. Matrix Size (Scheduling)

5 Exercise 11

5.1 Outer Loop Parallelization

This part of the exercise was much more difficult to get working, and proved largely problematic. My approach involved changing the `break;` statement to a shared variable, and then inserting that variable as a condition in the for loop. Overall, the precision and execution times were all over the place. The iterations maintained a large constant value at 1597388920 regardless of the thread number or size.

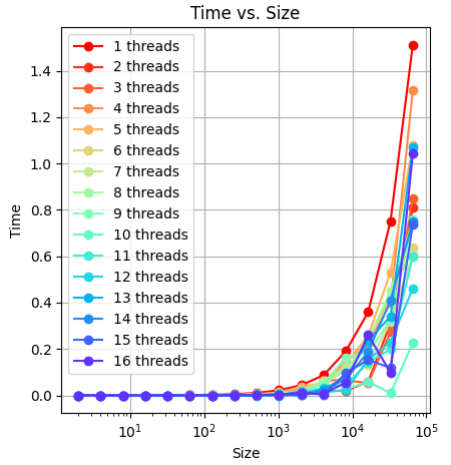

Figure 8: Thread Counts on Execution Time vs. Matrix Size (Outer)

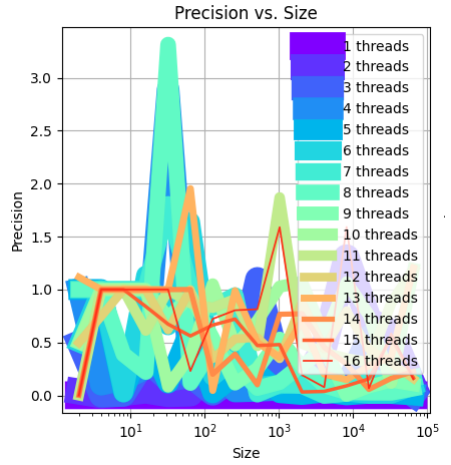

Figure 9: Thread Counts on Precision vs. Matrix Size (Outer)

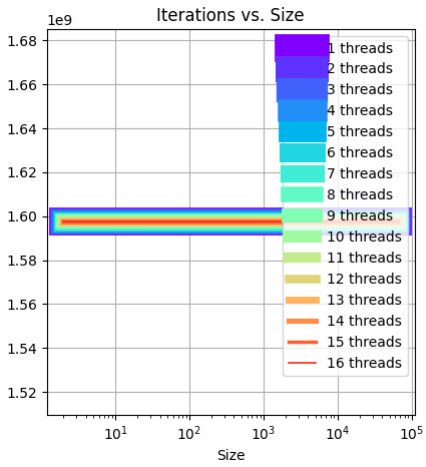

Figure 10: Thread Counts on Iteration vs. Matrix Size (Outer)