Alper Şahıstan
21501207

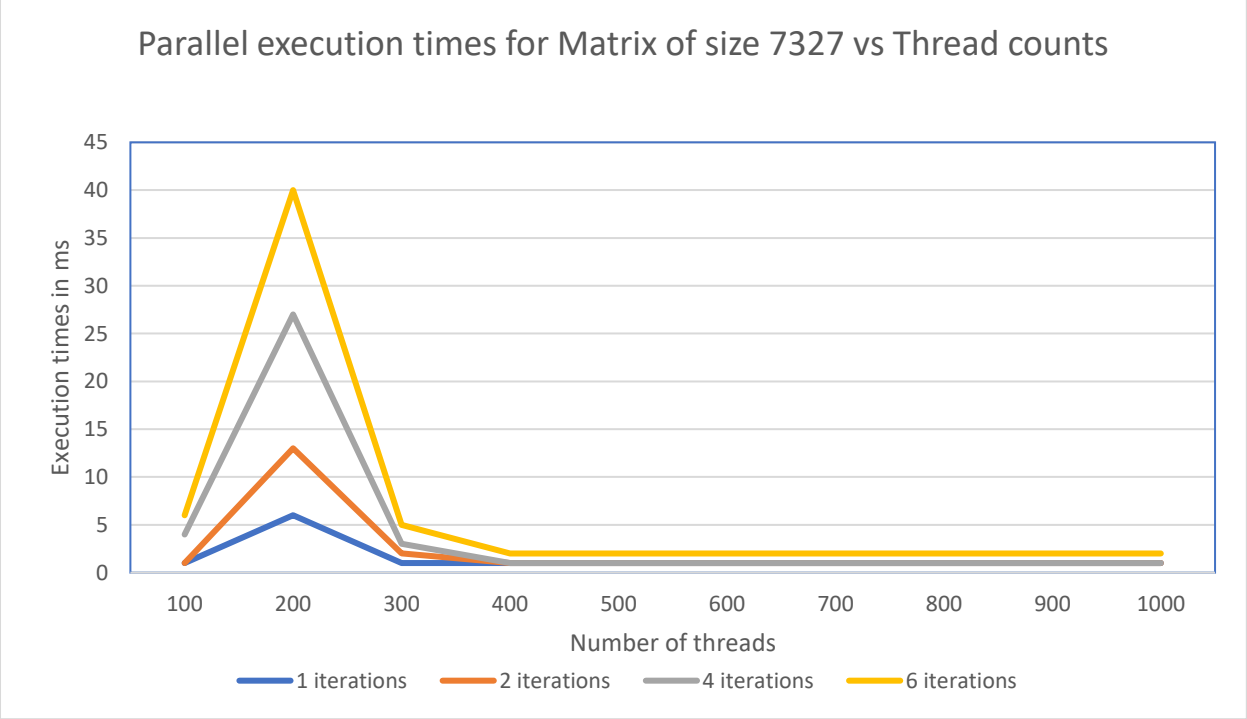
CS426-Parallel Computing Project 4 Report

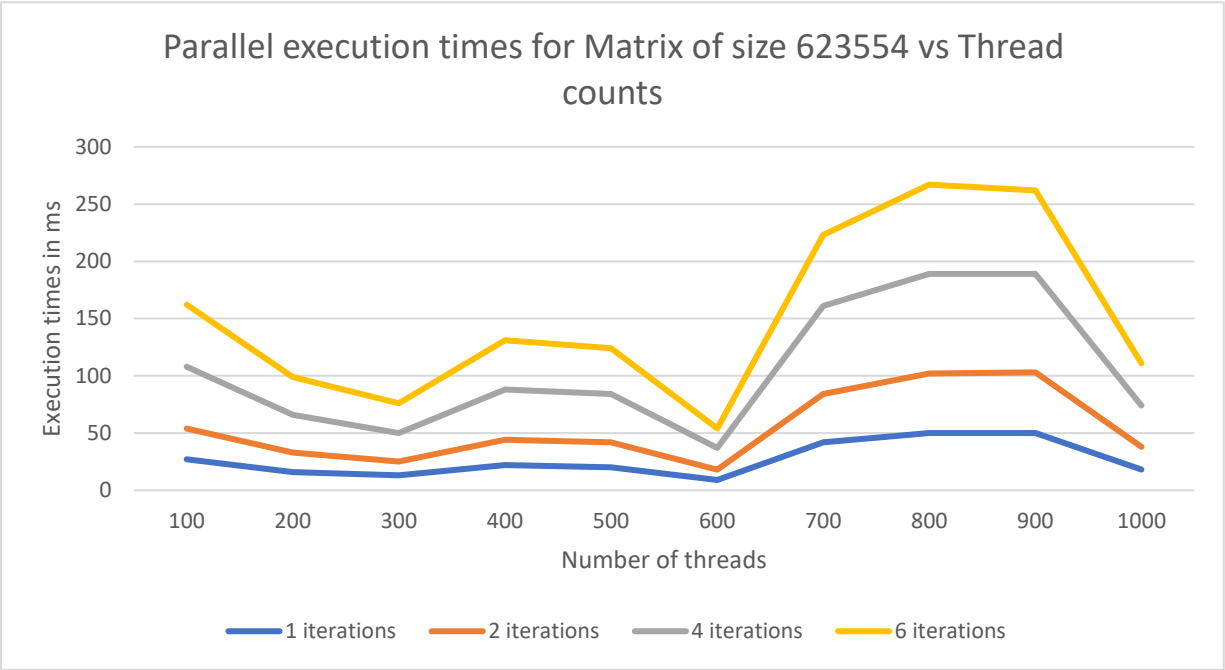
1. Parallelization Strategies Used

The program uses row-wise distribution of the matrix to the threads. For each thread kernel computes how much of the rows it should calculate. This is done by dividing the total number of to the total number of threads. If the number of rows is not divisible by number of threads remainder of the division is assigned to the last thread.

In between iterations program does not copy device arrays to host and back to device again. Instead 2 device arrays are allocated. For each iteration these arrays switch roles of input and output. For example let d_vector1 be input and d_vector2 be output vector for iteration 1. In iteration 2 d_vector1 will be output and d_vector2 will be input. After all iterations are complete the vector that contains the final result is copied back to host. By doing so the bottle neck of iteration count - 1 copies are avoided.

2. Results





Speed up for test matrices:

For **size 7327** the **serial run time is 19 ms** and best parallel time is achieved with **100 threads** and it is **1 ms**. **Speed up: 19 / 1 = 19.00, Efficiency: 19.00 / 100 = 0.19**

For **size 103076** the **serial run time is 237 ms** and best parallel time is achieved with **200 threads** and it is **7 ms**. **Speed up: 237 / 7 = 8.78, Efficiency: 8.78 / 200 = 0.09**

For **size 623554** the **serial run time is 1310 ms** and best parallel time is achieved with **600 threads** and it is **9 ms**. **Speed up: 1310 / 9 = 145.56, Efficiency: 145.56 / 600 = 0.24**

3. Discussion

For different sizes of matrices there are different low spots in the graph curves. However these points are consistent among iterations since implementation uses parallelized matrix multiplication. We achieve better speed ups in larger matrix sizes. Even efficiency goes up with larger matrices. When we look at the first graph the results indicate that after 300 threads execution time flat-lines to 1ms. This is due to small matrix size. However when we look at the larger matrices we see that the situation is not the same after the fastest execution time increasing the thread count increases the execution time unlike the first case. This is due to last thread handling the remainder part of the jobs. Since the row count is small in smaller matrices the remainder cannot grow to affect the execution time much. But in the case of 2nd and 3rd matrices having the row counts of 3876 and 22294 respectfully does increase the possibility of significant overhead for the last thread. This can be solved with load-balancing the work yet doing this will also add additional overhead and wont be cost effective for smaller matrices.