CS 426 Parallel Computing Project 3 Report

Bartu Atabek - 21602229 30/04/2020

1 Parallelization Strategy

1.1 Parallelization strategy used in the two parts. (You can use any parallelization strategy that scales up.)

This project concerns multiplication of a sparse matrix with a vector. I implemented the operation xi+1 = Axi, where A is a sparse matrix and x is the vector. We were instructed to store result of the product of A and x into x at the end of ith iteration and use it in the i+1th iteration).

From the given test matrices we were instructed to represent the sparse matrix data using the Compressed row storage (CRS) format where we are storing the nonzero elements row-wise. Three vectors are needed:

- **values array:** float vector storing the matrix elements
- **col_ind array:** int vector storing the column indices of the matrix elements
- **row_ptr array:** int vector pointing to the first elements of each row

With the given structures for my parallelization strategy I computed the resulting vector x element-by-element:

$$y(k) = \sum_{i=row_ptr(k)}^{row_ptr(k+1)-1} a(i) \cdot x * (col_ind(i)) \tag{1}$$

This is a 'sparse' dot product. One vector is stored contiguously (stride one), the other vector is scattered in memory. In this equation we assume that row ptr(N) points to the memory location right after the last element of a, i.e. right after the last element of the last row of A.

$$col.ind(i_1) \neq col.ind(i_2) for \tag{2}$$

$$row.ptr(k) \leq i_1 < i_2 < row.ptr(k+1) \tag{3}$$

With CRS parallelization over rows is straightforward in which for each iteration I used the '#pragma omp for' for-loop to iterate through the values and used '#pragma omp atomic' to protect a single update to a shared variable to store the multiplication results. Afterwards I used '#pragma omp barrier' so that no thread can proceed past a barrier until all the other threads have arrived. By using it I was able to synchronize the threads in order to gather all the multiplication results into the 'x' vector.

2 Figures for Each Test Matrix

2.1 The parallel running time, speedup, and efficiency of my OpenMP implementation for test-matrix "fidapm08.mtx"

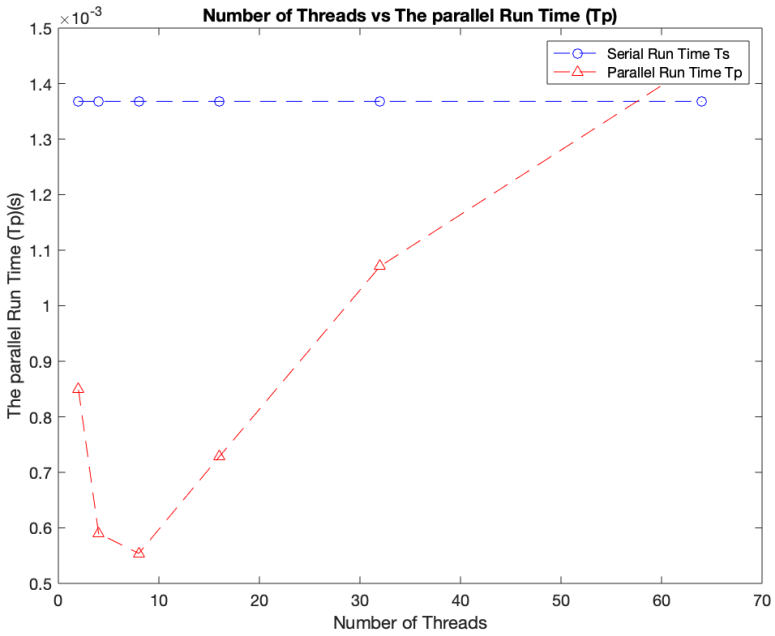

Figure 1: Serial vs Parallel Run-time comparison of sparse matrix vector multiplication on test-matrix "fidapm08.mtx" where number of threads for parallel execution is between 2 and 64.

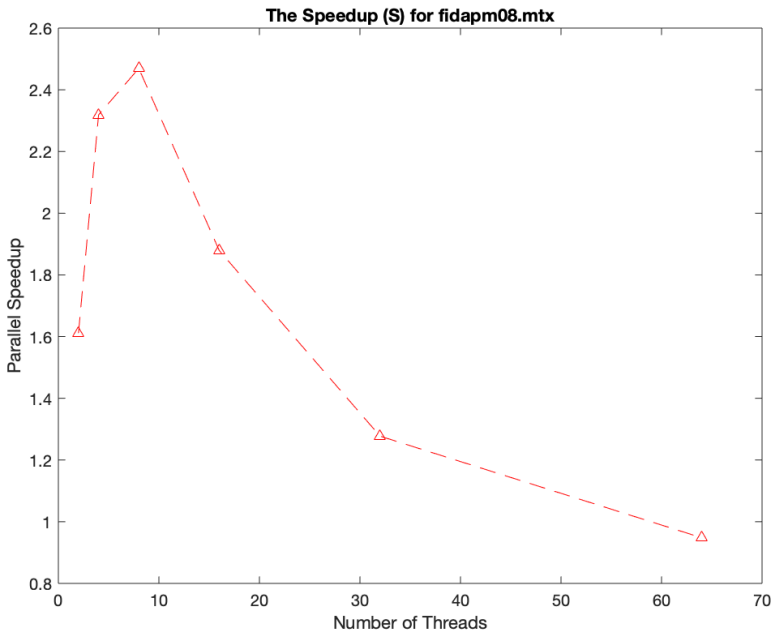

Figure 2: The speedup S on test-matrix "fidapm08.mtx" where number of threads for parallel execution is between 2 and 64.

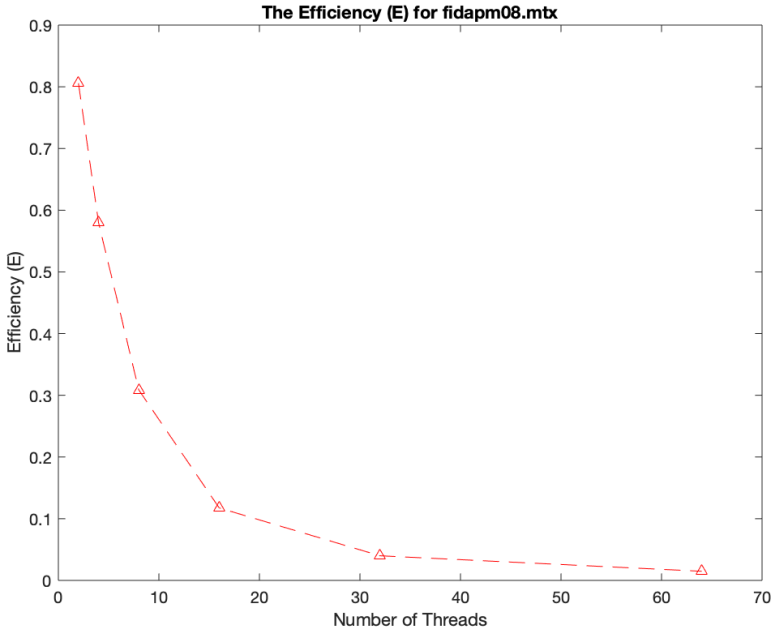

Figure 3: The efficiency E on test-matrix "fidapm08.mtx" where number of threads for parallel execution is between 2 and 64.

2.2 The parallel running time, speedup, and efficiency of my OpenMP implementation for test-matrix "fidapm11.mtx"

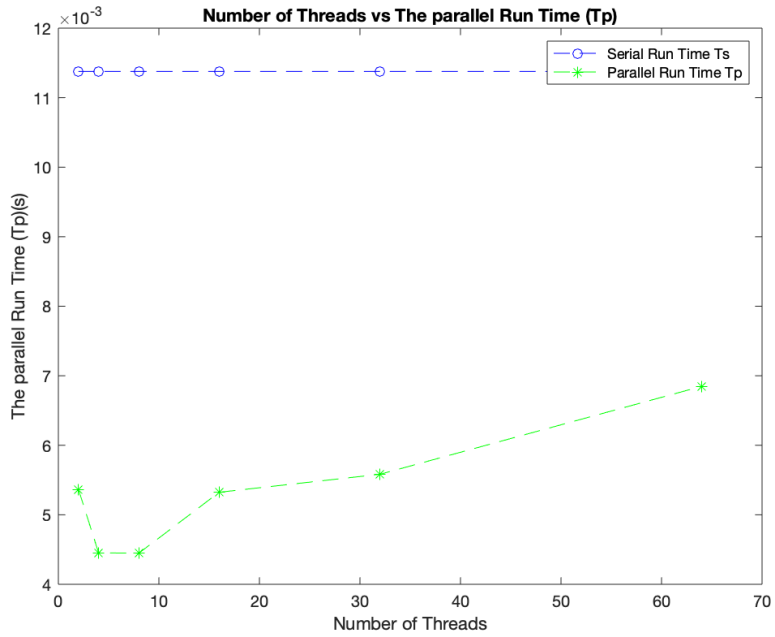

Figure 4: Serial vs Parallel Run-time comparison of sparse matrix vector multiplication on test-matrix "fidapm11.mtx" where number of threads for parallel execution is between 2 and 64.

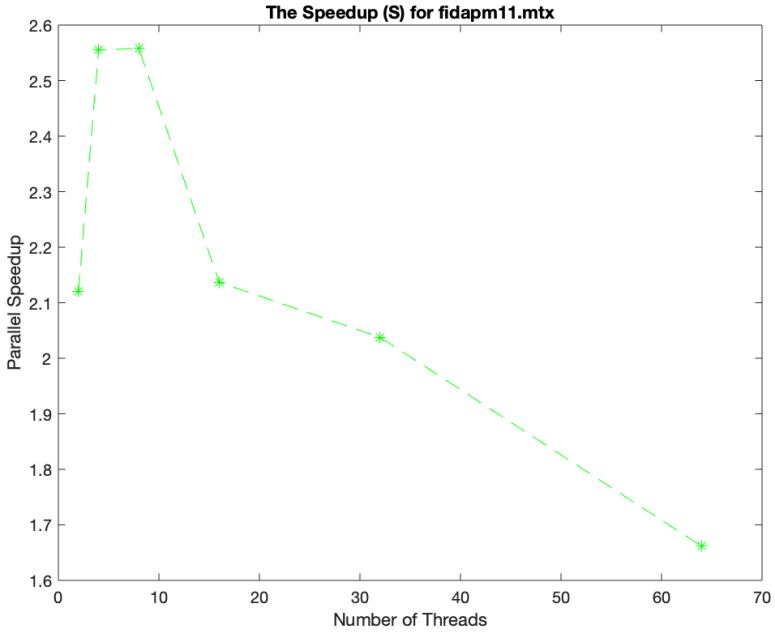

Figure 5: The speedup S on test-matrix "fidapm11.mtx" where number of threads for parallel execution is between 2 and 64.

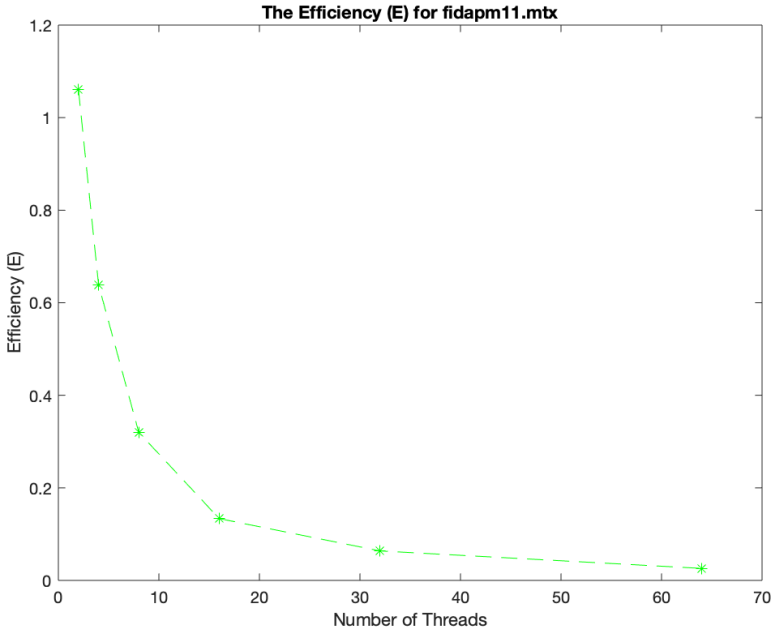

Figure 6: The efficiency E on test-matrix "fidapm11.mtx" where number of threads for parallel execution is between 2 and 64.

2.3 The parallel running time, speedup, and efficiency of my OpenMP implementation for test-matrix "cavity02.mtx"

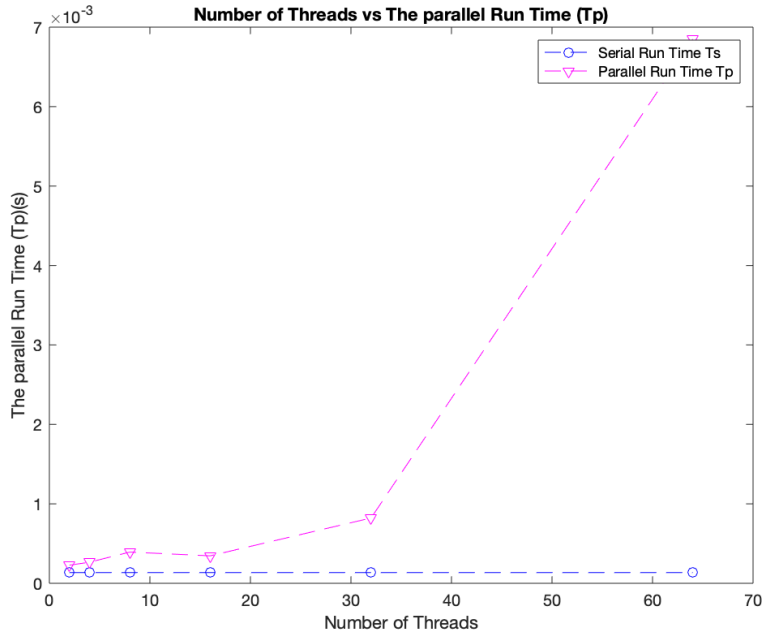

Figure 7: Serial vs Parallel Run-time comparison of sparse matrix vector multiplication on test-matrix "cavity02.mtx" where number of threads for parallel execution is between 2 and 64.

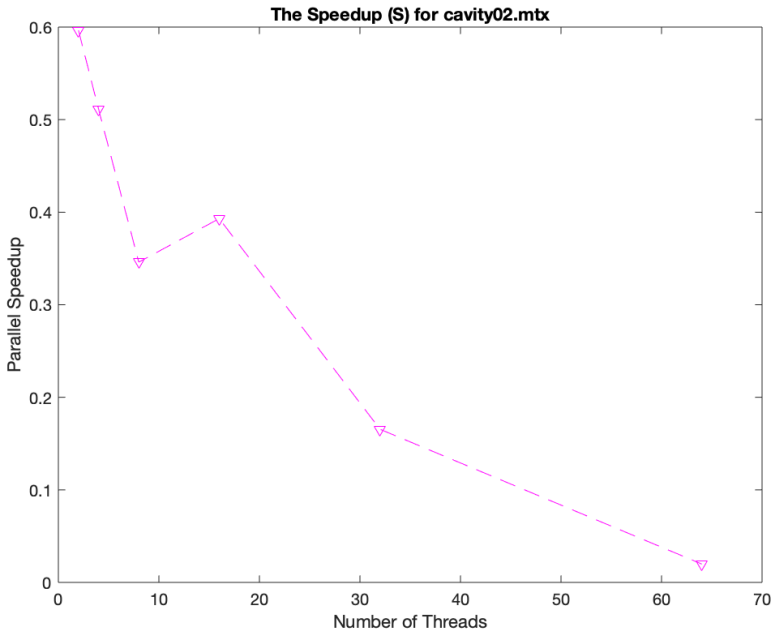

Figure 8: The speedup S on test-matrix "cavity02.mtx" where number of threads for parallel execution is between 2 and 64.

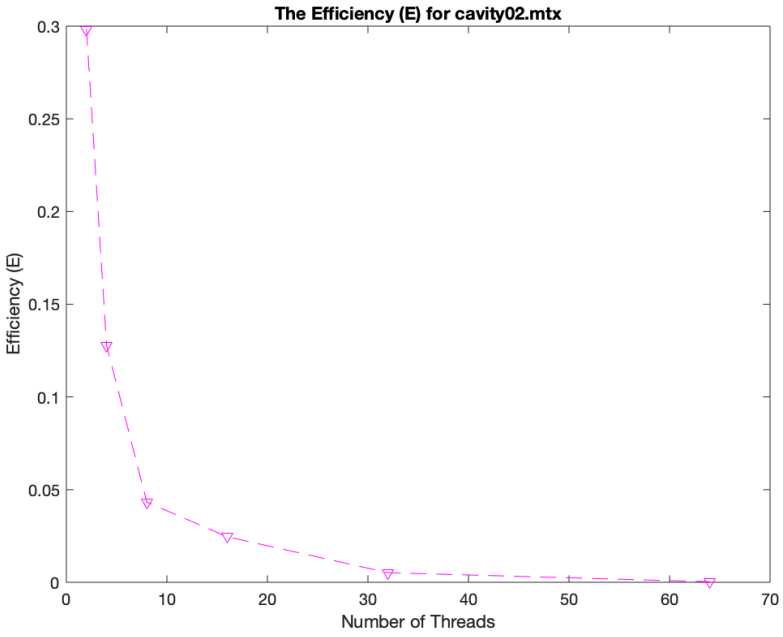

Figure 9: The efficiency E on test-matrix "cavity02.mtx" where number of threads for parallel execution is between 2 and 64.

3 Results & Discussion

With respect to the parallel time, speedup and efficiency results shown in the figures above for three distinct test matrices I will discuss the performance and scalability of the algorithm. Note that the algorithm may have different performance on different parallel architecture.

First of the the data shown in the figures was obtained by running the program with the following arguments;

./exe 'no_of_threads' '1' '0' 'filename'

where no_of_threads was valued between 2-64 and the number of iterations were fixed as one.

The parallel run time is defined as the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution which was measured from the start of the multiplication to the end discarding the file reading. As it can be seen from figures 1,4,7 almost in every execution the parallel run time was faster than the serial execution for each test matrix. Only during the execution with 64 threads the execution in the first and third matrix exceeded the serial execution which can be explained by the fact of communication overheads. Since the number of iteration was fixed as one separating the process into 64 threads increased this overhead. Also from another perspective the size of the first and the third matrix compared to the second one which is almost 70x bigger than the third matrix is also another factor. As size increases the values differ as it can be seen from the figures.

The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

$$S = \frac{T_s}{T_p} \tag{4}$$

According to figures 2,5,8 the speedup initially increases in the first two figures and then starts decreasing. In a typical algorithm as the number of processors increase the speedup could be negative, sub-linear, linear, super-linear. As it can be seen for the initial executions the speedup follows a super-linear path which was caused by the optimization of the CRS format to store the sparse matrix which indicates that the parallel version does less work than corresponding serial algorithm. Afterwards for a typical success scenario it was expected that the curve to stabilize and take a constant value however, in this case it starts to decrease. This again could be related to the fact that number of iterations is one.

The efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized.

$$E = \frac{S}{p} \tag{5}$$

For the first and the second matrices the efficiency starts as approximately to one and decreases in a linear fashion with respect to the speedup values. In all three test matrices this is common however, in the third matrix efficiency starts from 0.3. Since the number of iterations is set as one this effects the efficiency of the program as well.

In terms of the scalability of the solution the fact that

- **Increase number of processors –> decrease efficiency**
- **Increase problem size –> increase efficiency**

This can be clearly seen from the figures since the matrix with the smallest size performed the worst whereas the matrix with the largest size performed the most optimal. Also increasing the number of the threads decreased efficiency which could also seen from the figures. Overall however, a scalable parallel system can always be made cost-optimal by adjusting the number of processors and the problem size.