CS426: Project #4 Report

Doruk Çakmakçı
21502293

Bilkent University — June 2, 2019

1 Introduction

Sparseness of a matrix is an important concern for matrix multiplication. Because, the zero-valued elements of a matrix does not contribute to the output of matrix multiplication. For improving space complexity of a program, 2D sparse matrices are stored using three arrays instead of a two dimensional array. One of the arrays store an index value(an implementation detail which has no spatial meaning) corresponding to the first non-zero element for each row. The second array has elements which are the indexed by the index value mentioned above which holds the column of the corresponding matrix element. The third array hold the values corresponding to the row and columns of the matrix and indexed in the like manner with the second array. In the following analysis, first array is referred as row_ptr, the second array is referenced as col_ind and the third array is referred as values array

Our goal in this project is to parallelize sparse matrix vector multiplication using GPU parallelism. NVIDIA CUDA is used as the parallelization tool.

2 Parallelization Strategy

The algorithm is parallelized by distributing the elements of the row_ptr array to the threads. To be concise, 1D block distribution(as input data decomposition technique) is used as the parallelization strategy. Each Thread computes the start(inclusive) and end(exclusive) indices for its own computation. In order to ensure load balancing, first the quotient of $number\_of\_rows/number\_of\_threads$ is distributed to all threads. The remainder of this division is distributed to threads with respect to the id's of the threads. Therefore the maximum difference in computation(in terms of rows) is for any two thread. However, since rows can have drastically different number of non-zero elements(some have no elements at all), load balancing can only be done by distributing the values array to the threads. But, using a parallelization strategy that uses values array distribution is very hard to implement and introduces more space requirements. If the this algorithm would be used in a real-time system, then values array distribution would be more feasible in terms of running time requirements.

CUDA uses grids of blocks and blocks of threads. For this implementation, grids consist of 1 block and blocks contain 1D thread blocks. Also there are number_of_threads thread blocks.This scheme is bounded by the architectural limits. In the CUDA documentation, it is stated that a block can have at most 1024 threads inside it. However, in my analysis, I have seen that 1024 threads is enough for a good parallelism.

3 Notes

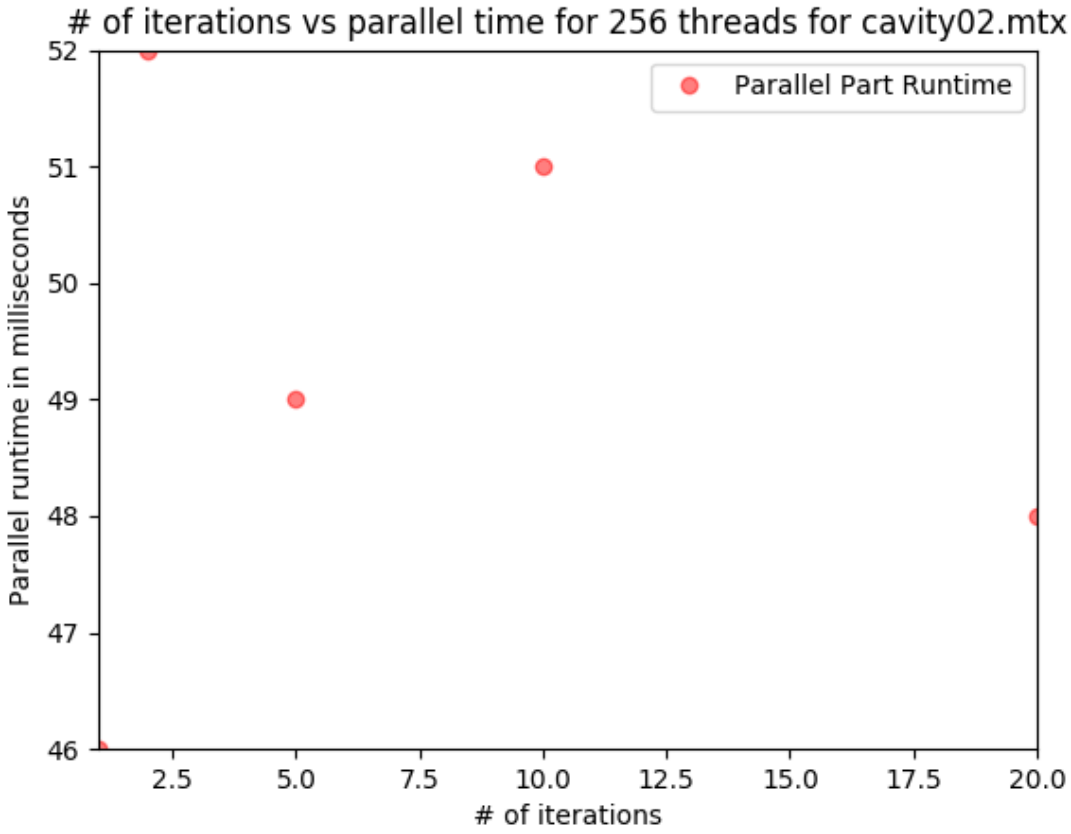
In the following analysis, Amdahl's law is used to compute the speedup and efficiency of the parallel algorithm. Instead of using Amdahl's law, one can implement the sequential version of the sparse matrix vector multiplication. However, to measure speedup, one must use the best algorithm(in terms of time complexity). Since the best implementation is a relatively contextual, Amdahl's law is used. The sequential part of the algorithm consists of reading the sparse matrix from file. The parallelizable part of the implementation is the sparse matrix vector multiplication. Since there is no sequential implementation of the algorithm for this project, an approximate analysis for the proportion of the parallelizable part of the implementation would be dependent to the iteration count. The sequential part of the algorithm contains

$3 * k$(k is just a proportionality constant 3 is for creating the three arrays mentioned previously). The parallelizable part of the algorithm contains $1.5 * (iteration\_count) * k$(1.5 is for copying from and to device memory and the computation on the elements. this factor is multiplied with iteration_count to scale up).

4 Figures

4.1 cavity02.mtx

4.1.1 Parallel Running Time vs of Iterations

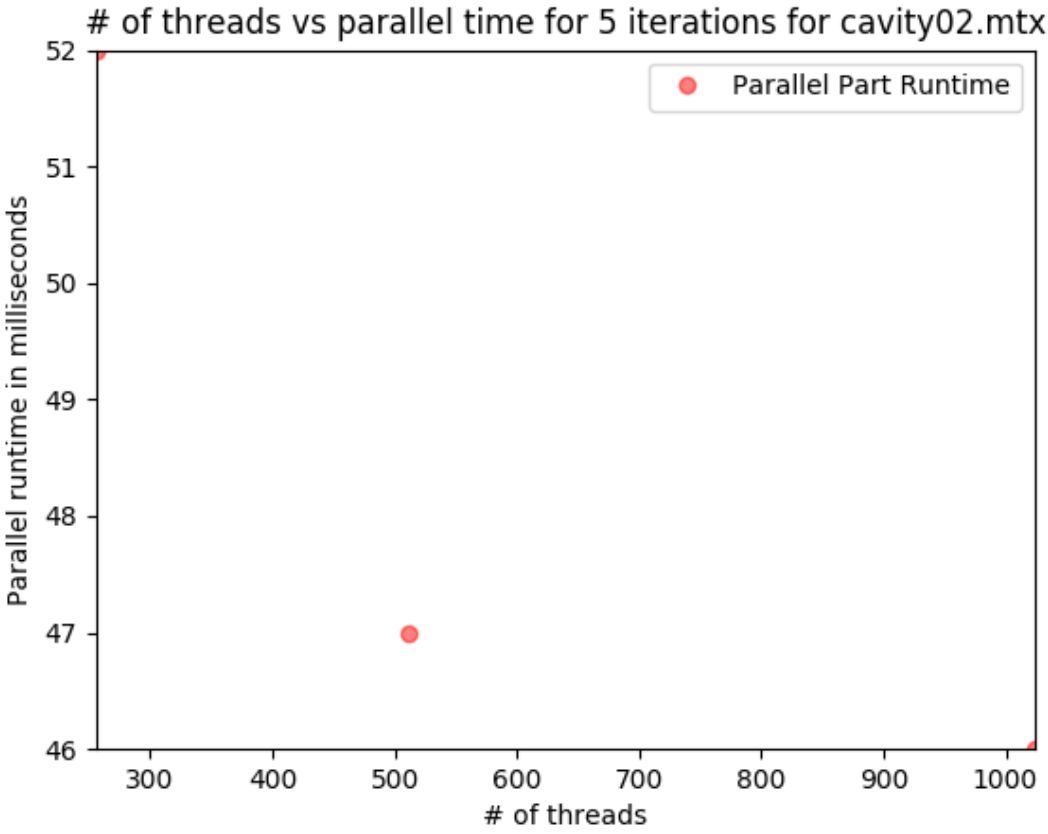


From the figure we can see that the number of iterations do not monotonically increase the running part of the parallel algorithm. I did not foresee this result since increasing the number of iterations should increase the running time of the program drastically. For this case, it may be the case that the size of the cavity02 matrix is small and does not introduce a lot of overhead while increasing iteration. As a last minute note I have found out that the selected iteration counts for this plot are not representative of the effect of number of iterations on the parallel running time. The following results (tested using the same parameters) are more representative:

- **100 iterations:** parallel part running time is 60ms
- **1000 iterations:** parallel part running time is 102ms
- **10000 iterations:** parallel part running time is 570ms

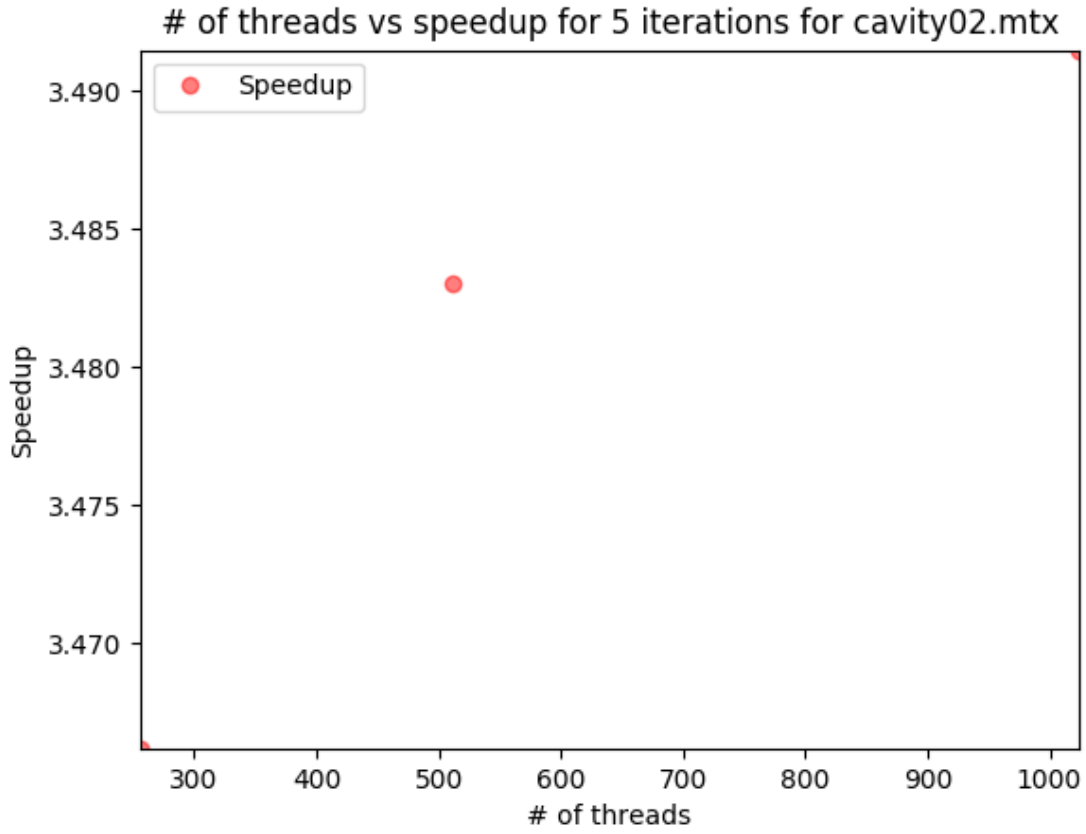
We can see that 1 iteration does not introduce a significant overhead but testing the program using the powers of 10 gives better results.

4.1.2 Parallel Running Time vs of Threads



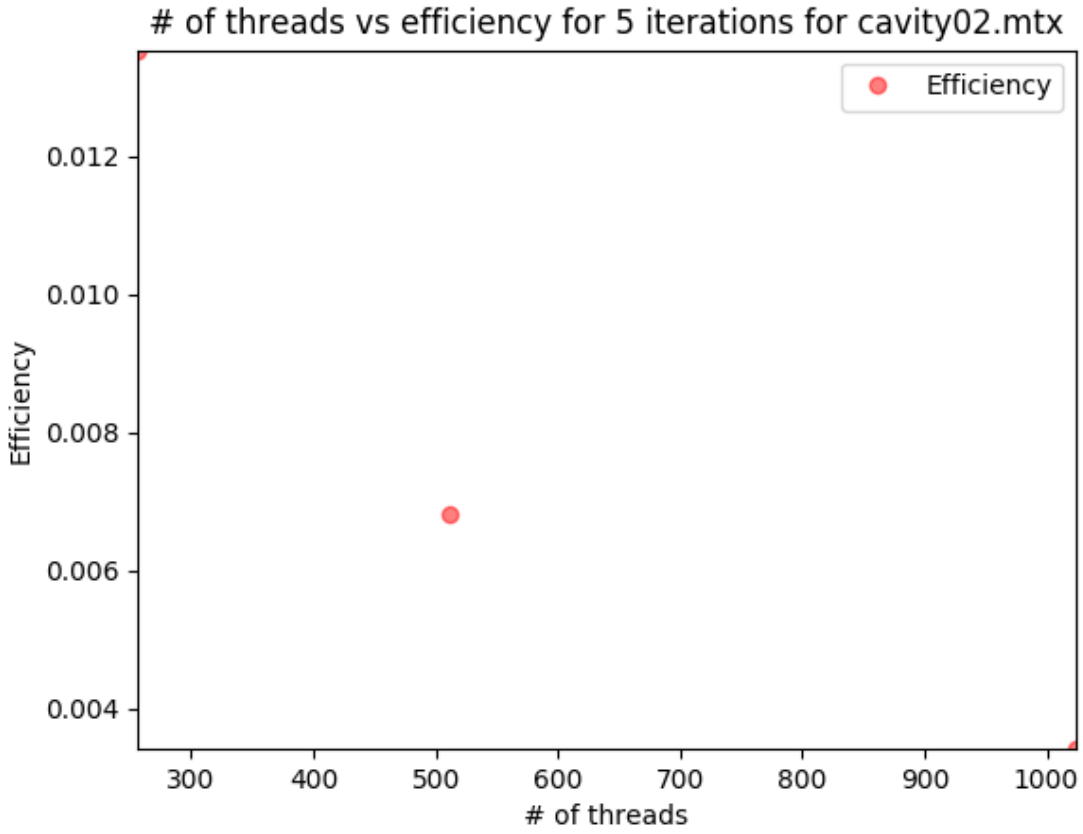
As expected, we see that the number of threads used is significant in terms of running time(the outcome is dependent to the size of the input and over-decomposing the inputs can also generate an overhead higher than the benefit of the parallelization). As the number of threads is increased, the running time of the program decreases.

4.1.3 Speedup



Keeping in mind that the speedup is bounded by the number of processors and the plotted speedup is theoretical and approximate, the results are not good. However a more pervasive analysis of the implementation(using the best sequential implementation of sparse matrix vector product and calculating the speedup as $sequential\_time/parallel\_time$) may yield a better result. From another perspective, using cuda cores as the number of processors in the Amdahl's law is may not be feasible since some CUDA cores may be using the physical processor.

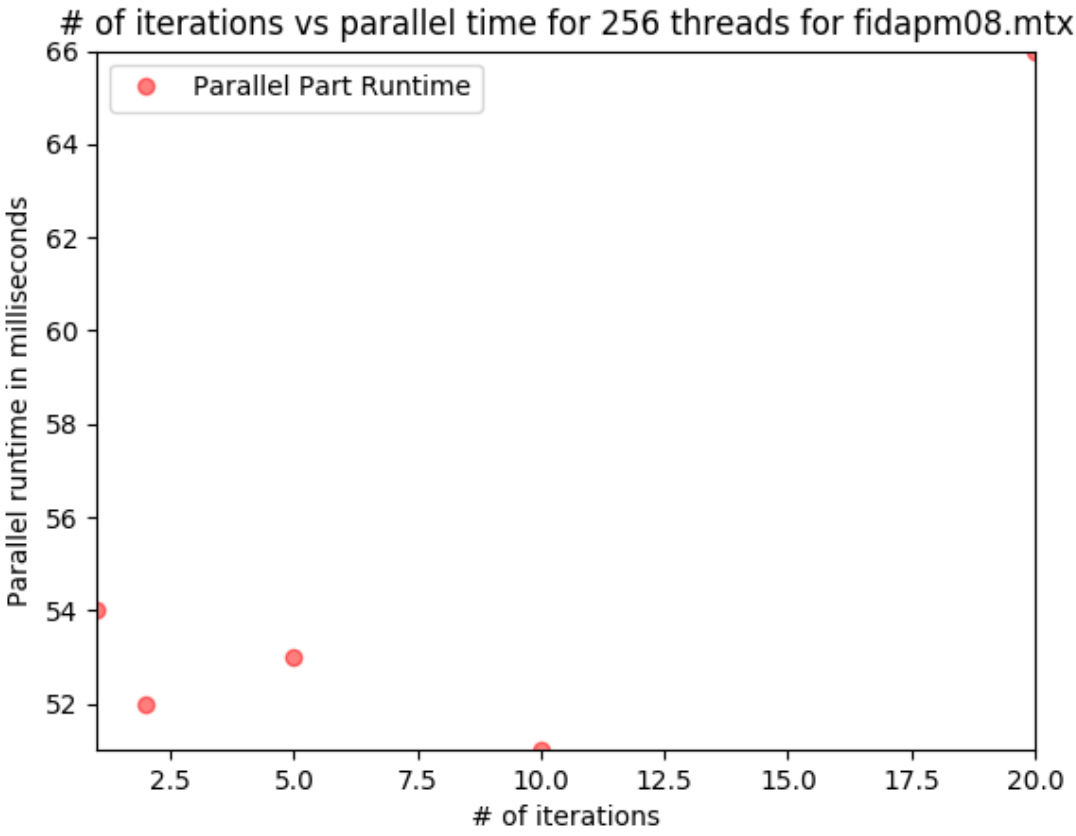
4.1.4 Efficiency



Since efficiency is the speedup per processor, we can say that the speedup per processor drops with respect to the number of threads used. Speedup tends to saturate with respect to the number of threads(if more threads are used it is speculated that speedup would converge(saturate)) and efficiency drops as a consequence. We can see this phenomena in this plot. Also this system is not cost-optimal since the efficiency is not constant with respect to number of processors and speedup per processor changes.

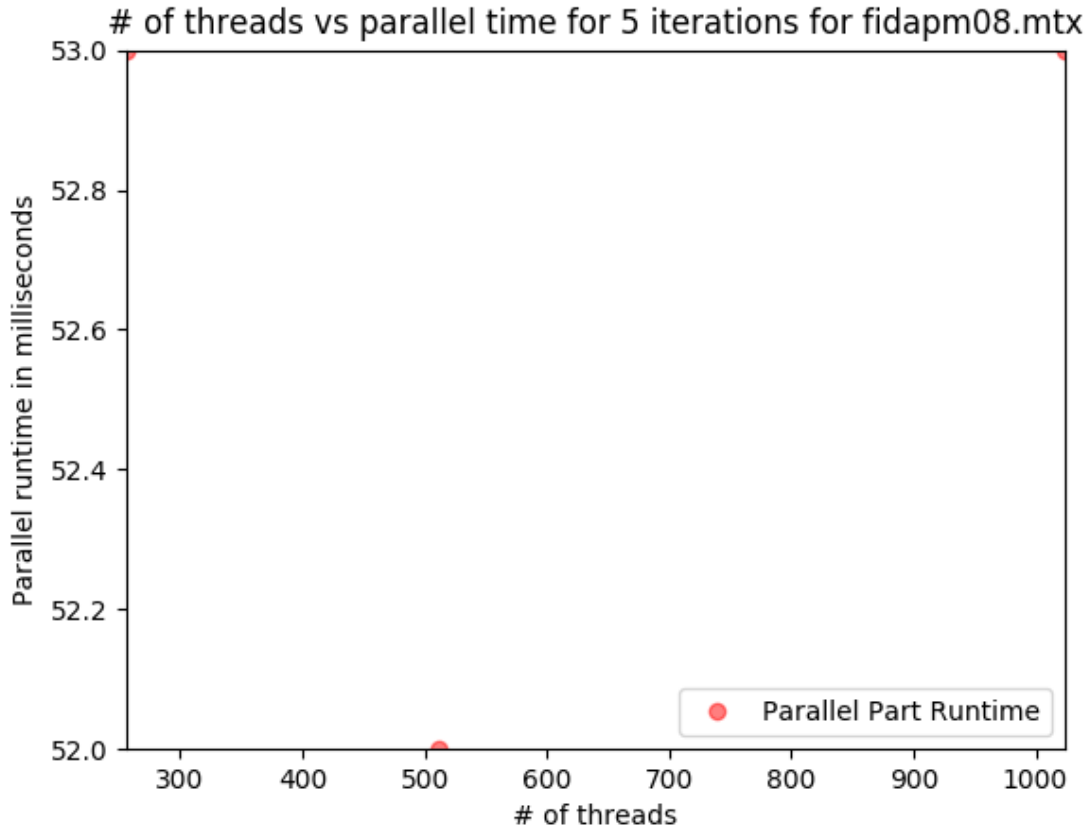
4.2 fidapm08.mtx

4.2.1 Parallel Running Time vs of Iterations



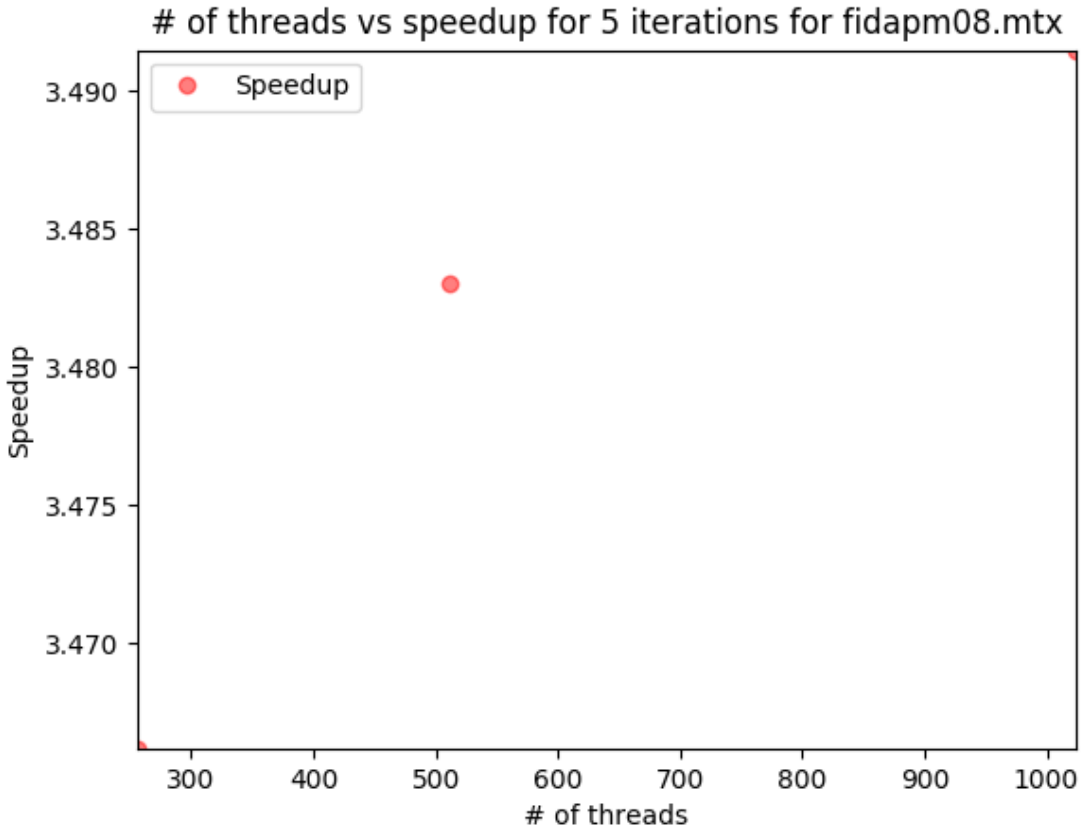
We can see that the parallel running times are increased slightly with respect to the size of the input.

4.2.2 Parallel Running Time vs of Threads



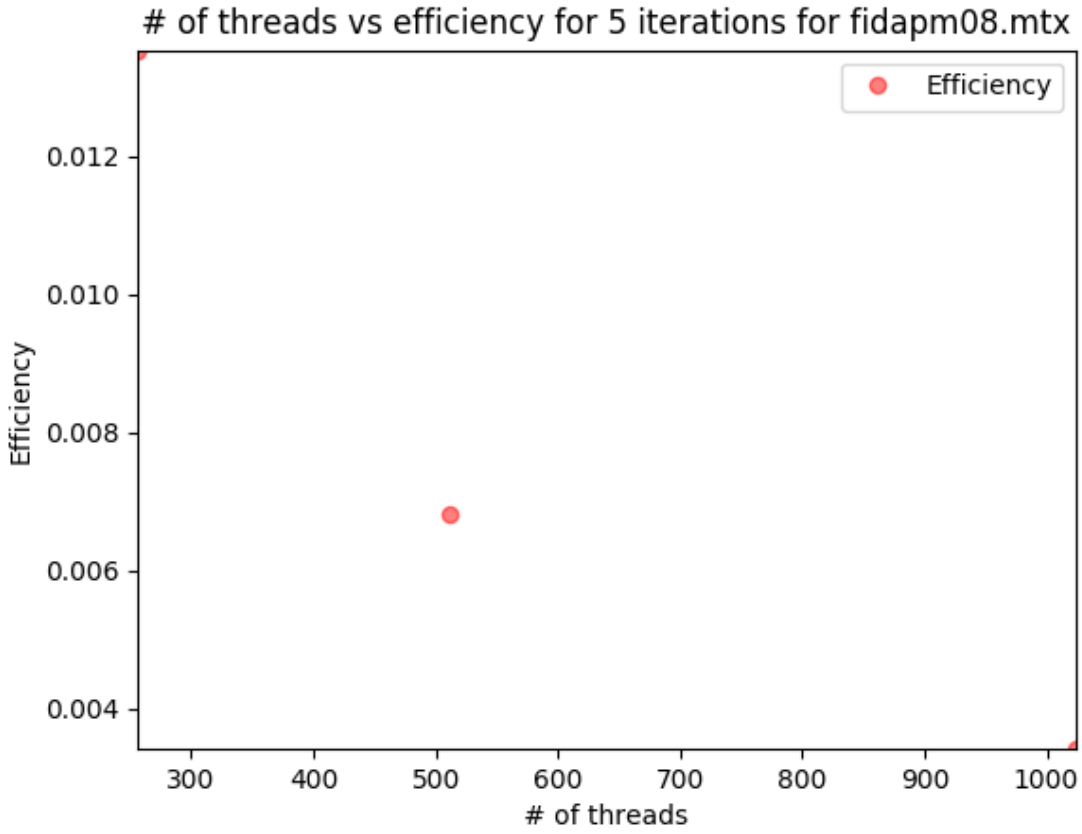
The parallel running time is almost constant with respect to these number of threads but since this only a small part of a big spectrum no influential analysis can be made from this graph.

4.2.3 Speedup



A similar analysis as the cavity02 matrix can be made.

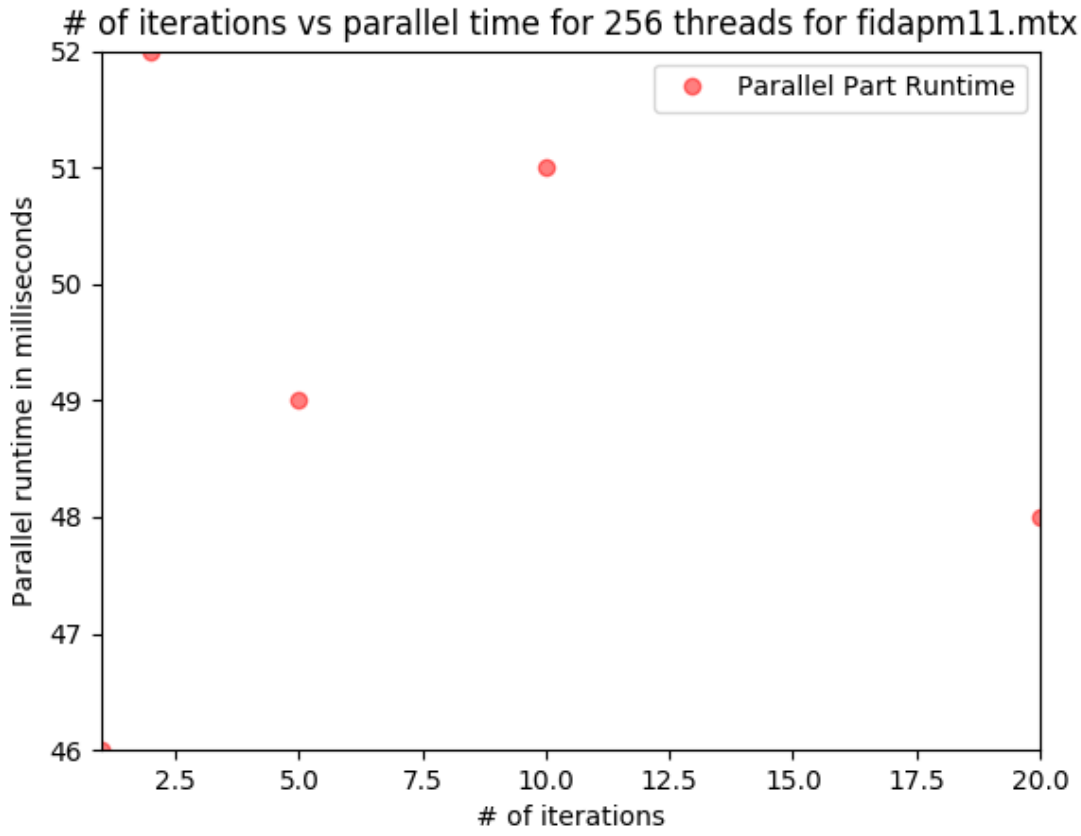
4.2.4 Efficiency



A similar analysis as the cavity02 matrix can be made.

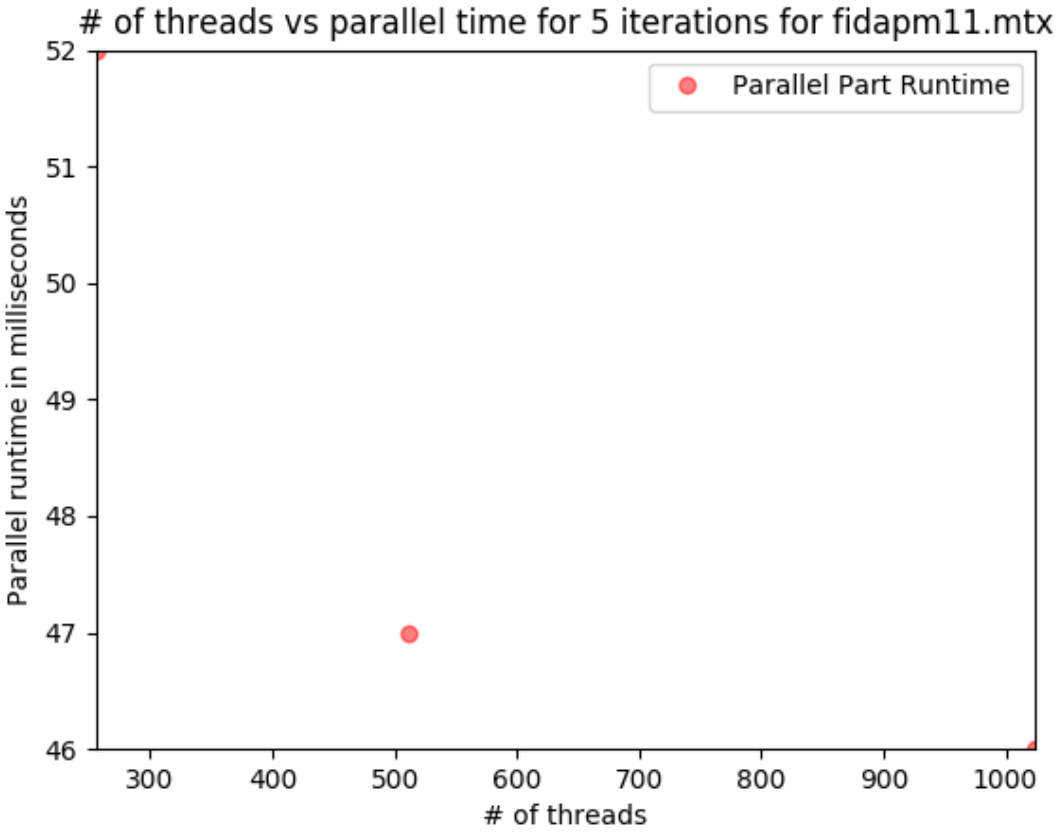
4.3 fidapm11.mtx

4.3.1 Parallel Running Time vs of Iterations



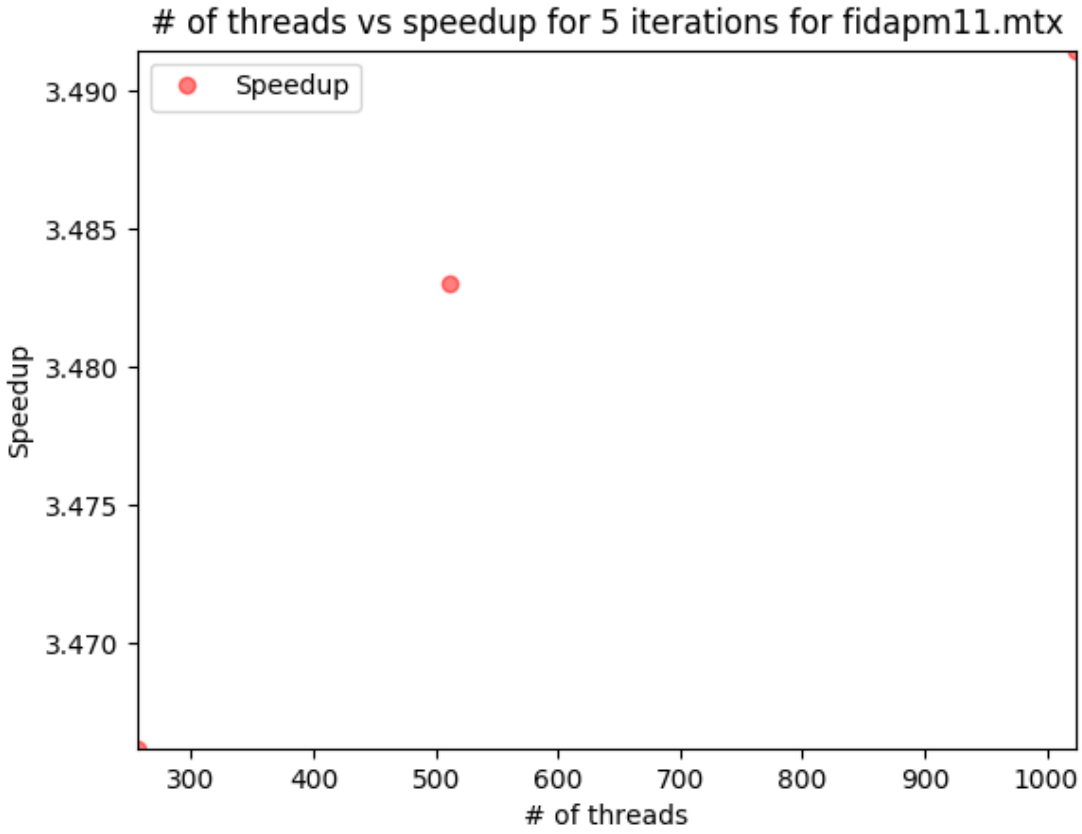
A similar analysis as the cavity02 matrix can be made.

4.3.2 Parallel Running Time vs of Threads



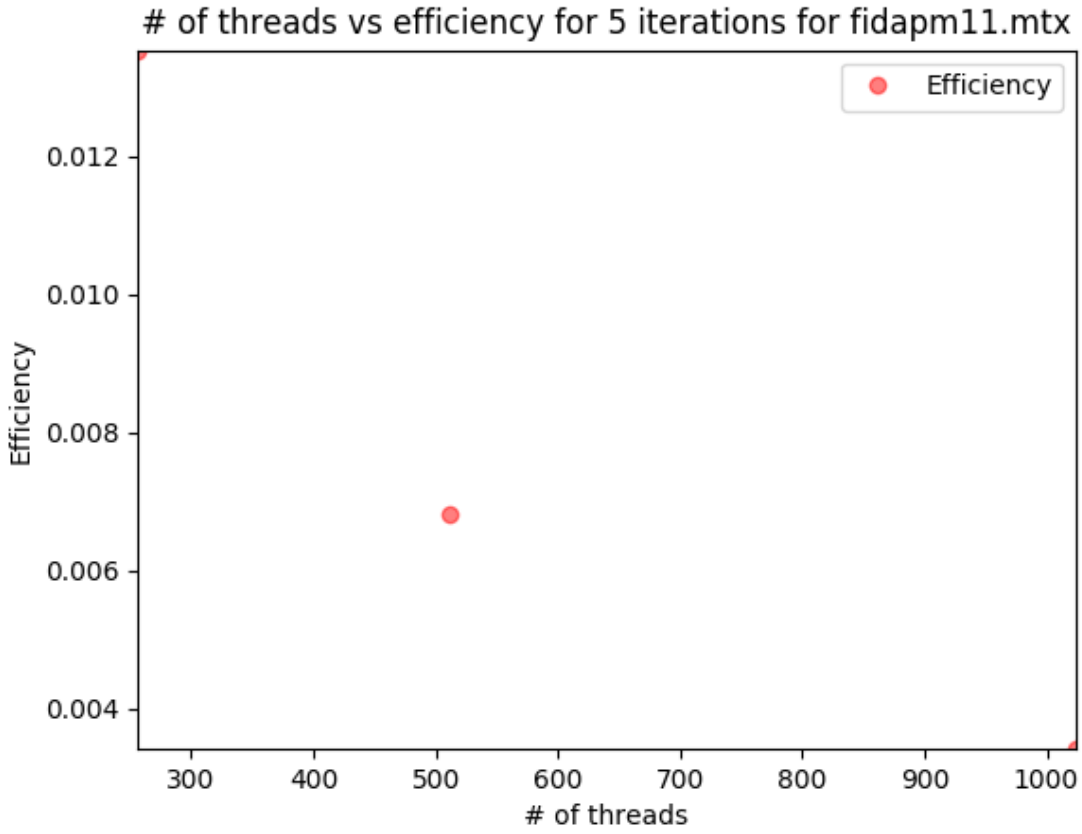
A similar analysis as the cavity02 matrix can be made.

4.3.3 Speedup



A similar analysis as the cavity02 matrix can be made.

4.3.4 Efficiency



A similar analysis as the cavity02 matrix can be made.