CSE 402 - Project Report
Finding eigenvalues of a symmetric matrix using QR iteration. (OpenMP and MPI)

Name - Pavan Ravi, UIN - 664834734, NetID - pavanr2

15 May 2022

Contents

List of Figures

1 Abstract

Eigenvalues are invariants of a matrix and sometimes have physical relevance. Eigenvalues of a symmetric matrix are used to study various physical systems. An efficient way to determine eigenvalues can significantly reduce computation time. In this project, the QR iteration method is explained. QR iteration is a 2 step process. First a matrix is transformed to a simpler matrix and then the QR iteration is performed. The OpenMP and MPI algorithms for QR iteration are discussed. The performance and scaling of these versions are also explored. The MPI version presented here has a lot of room to be improved. This will be discussed in the final section.

2 Introduction

In this project, the QR iteration method for finding the eigenvalues of a dense symmetric matrix is explored. Symmetric matrices (both sparse and dense) are regularly encountered in various real world physics problems. One example is finding natural frequencies of a multi DOF body/structure.
Mathematically, symmetric matrices have all real eigenvalues. To determine the eigenvalues analytically, we need to solve the characteristic polynomial equation of degree n for a square matrix of dimensions n x n. This is computationally very expensive and very rarely used. There are many different numerical algorithms to find the eigenvalues and eigenvectors of a matrix. Usually, a given matrix is transformed to a similar matrix using similarity transformations. Then an iterative algorithm is used on the transformed matrix to extract the eigenvalues and eigenvectors. In this project, householder transformation is applied on a symmetric matrix to obtain a tridiagonal matrix. This tridiagonal matrix is reduced to a diagonal matrix using QR iteration. The diagonal entries of the matrix are the required eigenvalues.

3 QR iteration method

The QR iteration has two steps. First, we perform a householder transformation to obtain a tridiagonal matrix which has the same eigenvalues as the original matrix. The householder transformation is shown below. Taken from (https://en.wikipedia.org/wiki/Householder_transformation), the procedure is as follows.

for k = 0 : num_cols - 2

$$\alpha = -sgn(A^k_{k+1,k})\sqrt{\sum_{j=k+1}^{n}(A^k_{jk})^2}$$

$$r = \sqrt{0.5 * (\alpha^2 - A^k_{k+1,k}(\alpha))}$$

$$v^k_1 = v^k_2 = ...v^k_k = 0$$

$$v^k_{k+1} = \frac{A^k_{k+1,k} - \alpha}{2r}$$

$$v^k_j = \frac{A^k_{jk}}{2r}, j = k + 2...n$$

$$P^k = I - 2(v^{(k)})(v^{(k)})^T$$

$$A^{k+1} = P^{(k)}A^{(k)}P^{(k)}$$

This will yield a tridiagonal matrix. Note that the outermost loop is a sequential loop. So only the inner operations can be parallelized.
Let the tridiagonal matrix be called H. Now a QR factorization (many ways to do this) is performed.

for i = 1 : num_iter
Find QR factorization of H^i (it is a n-step process for a tridiagonal matrix)
$H^{i+1} = R^{(k)}Q^{(k)}$

At the end of the iterations, a diagonal matrix is obtained.

4 Implementation

As seen in the previous section, the outermost loops of both Householder tridiagonalization and the QR factorization are dependent on the values obtained in the previous iteration. So the outermost loops cannot be parallelized. However, it is possible to parallelize the inner operations and extract performance. The full code can be found here.

4.1 General data structures and other technical details

The code is written in C++ with two classes **matrix** and **vector** which are created out of 1D arrays. The choice of 1D array for matrix representation is due the fact that memory is contiguous and it simplifies creating MPI buffers. No external linear algebra packages were used.

Compiler used : gcc-7.2.0
MPI Implementation : OpenMPI v4.1.0
Compiled using : cmake-3.18.4

The serial code was designed using the algorithm mentioned in the previous section. The serial code is used as a baseline to compare parallel code performance.

4.2 OpenMP

The OpenMP parallelization involved mainly in parallelizing the inner loop operations of the Householder and QR iteration. A Vtune profiling was done on the serial code to figure out the hotspots which were consuming large number of memory cycles. Based on this, the repeated matrix multiplications consumed the most time and parallelizing this operation did help to acheive speedup. The parallelization of other operations also helped but the most significant is the matrix multiplication.
The matrix multiplication was implemented by swapping the loops to (i,k,j) ordering. The outermost loop was parallelized with a pragma directive. The (i,k,j) and (k,i,j) versions are cache friendly have have significantly lesser cache misses than the naive (i,j,k) order.
In the QR iteration, however, there isn't scope for much parallelization (there is a matrix multiplication, but now we are dealing with matrices not as dense as the initial symmetric matrix.

4.3 MPI

Similar to OpenMP parallelization, a few functions of the inner loop were parallelized. A 1D tiled matrix multiplication using (MPI_Scatterv(), MPI_Bcast() and MPI_Gatherv()) was implemented. Also, the norms of vectors were calculated using MPI_Scatterv() and MPI_Reduce(). These changes did bring an improvement in the performance.
If we have a matrix multiplication operation C = AB, then the rows of the matrix A are split using MPI_Scatterv(). Using Scatterv makes sure that we can handle arbitrary number of rows and processes. Logic is written to specify displacements for the Scatterv call. Example - If we have 10 rows and 3 processes, then rank 0 gets 4 rows, ranks 1 and 2 get 3 rows each. Matrix B is broadcasted to all processes (this can be improved) using MPI_Bcast(). Now each process computes a part of matrix C. To gather all the rows, we use MPI_Gatherv() call.
The L2-norm of the vector was evaluated using MPI_Scatterv () and MPI_Reduce(). If we have a vector v of arbitrary size, we need the sum of squares of the components. The vector is scattered to multiple processes using MPI_Scatterv() and each process computes the sum of squares. These local sums are reduced in rank 0 and square root is taken to get the norm.

However, there is still room to improve the computation-communication overlap. This will be discussed in 'Possible improvements' section.

5 Results and discussion

The OpenMP and MPI results are compared against the serial version of the code. The plot is give below. The number of threads here is 20 for the OpenMP version. This is half the number of cores in the allocated node on Campus Cluster (no hyperthreading). As expected, we see very bad speedup for very small matrices (like 16x16, 32x32). This is because the overhead of thread creation is much more compared to the serial execution time of the program. However, as we increase the matrix size, we see very good speedup (around 10x).
The MPI version also behaves in a similar manner to the OpenMP version when run on a single node. The communication overhead for small matrices is high but for larger size matrices, the observed speedup is much higher.

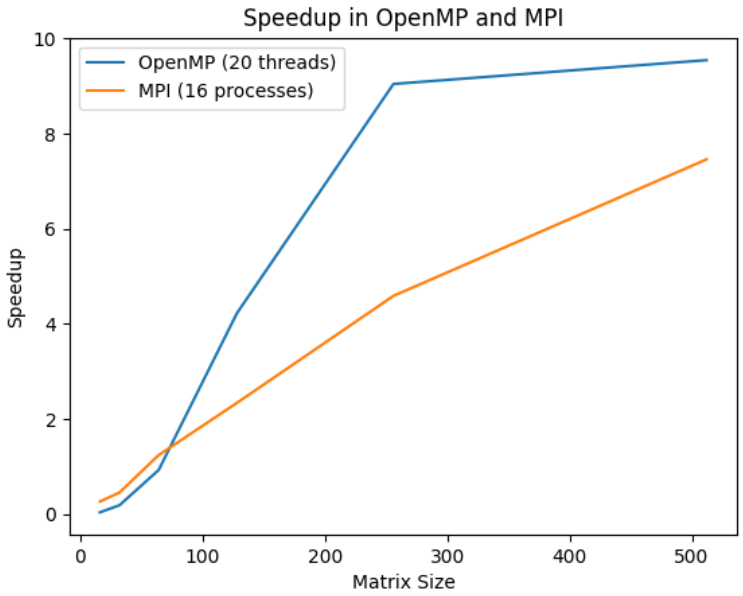

Figure 1: *Speedup vs. Matrix Size*

5.1 Scaling results

Strong scaling studies were conducted on the Campus Cluster for both OpenMP and MPI versions. The results are presented below.

5.1.1 OpenMP

Here, we observe how the execution time changes when number of threads are increased and keeping the matrix size to be a constant.

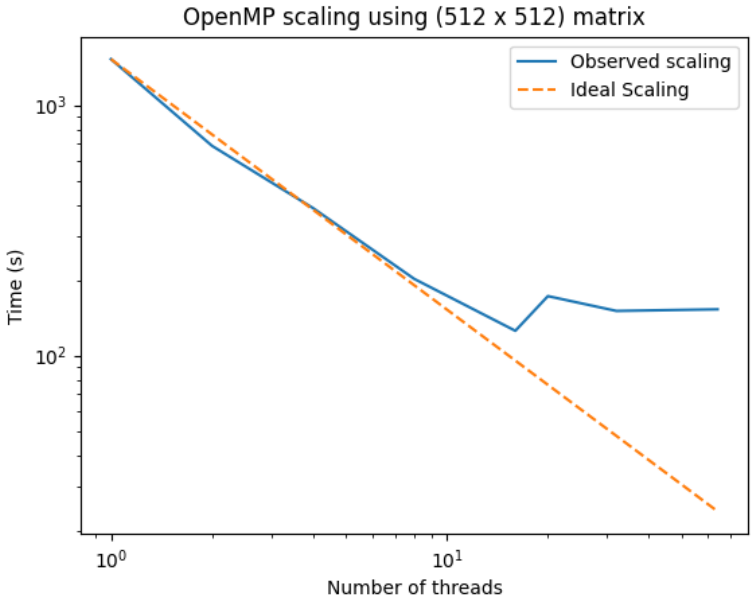

Figure 2: *Time (s) vs. Number of threads*

The dashed line shows the ideal strong scaling. Ideal strong scaling can be thought of as $T_n = T_1/n$ where T_n the time taken by the program to execute using n threads and T_1 is one thread execution time. The strong scaling efficiency is given by,

$$\eta = T_n/T'_n$$

where T'_n is the observed time when using n threads. The table below shows the scaling efficiency for a fixed matrix of size 512 x 512.

Number of Threads	Strong scaling efficiency(%)
1	100
2	110
4	97.95
8	94.35
16	76.03
20	44.21
32	31.67
64	15.6

We can see for 2 threads, we have achieved an efficiency greater than 100, this may be due to the arrangement of data in cache. We see that good scaling is observed from 1 to 8 threads (above 90%), after which the scaling efficiency goes down.

5.1.2 MPI

The scaling studies for MPI is done in a similar fashion. Instead of threads, we will be talking about number of processes. The plot below shows the scaling of the MPI version.

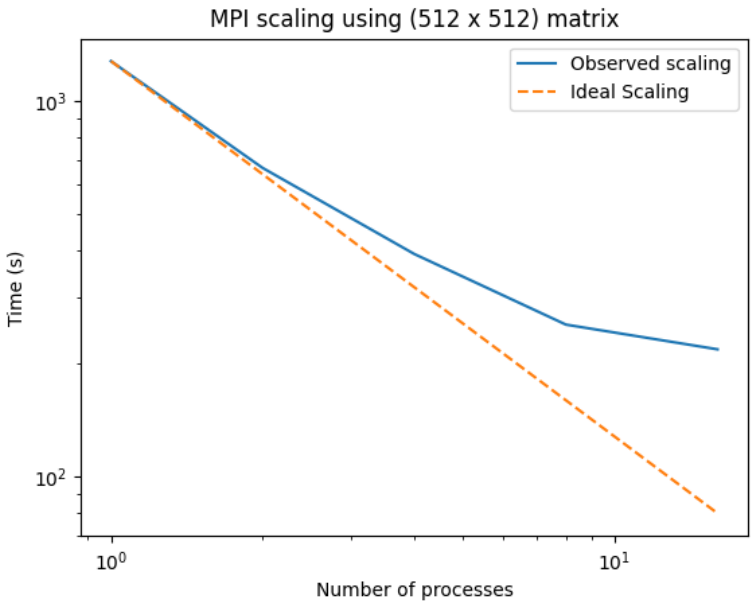

Figure 3: *Time (s) vs. Number of Processes*

Similar to OpenMP results, we calculate the scaling efficiency.

Number of Processes	Strong scaling efficiency(%)
1	100
2	96.22
4	81.6
8	62.8
16	36.6

The results above are for 1 node case. A multi node run was also performed but the results were not optimal due to some configuration issues with the fabric. The node to node communication was not optimal and results form those runs will be skipped for now.

6 Possible Improvements

As evident from the results, the MPI version can further be optimized. In this project, only a few functions were made to use MPI calls. For problems like these collective calls are more appropriate because most of the algorithm is about splitting the matrix in a row or a column fashion as needed. In the present implementation the most time consuming functions (matrix multiplication) were parallelized. Many other operations were done by single processes and collected at rank 0 to combine the results. This does give good speedup but the program doesn't scale that well.
Running a Vtune profiling on all three version of the code, it can be seen that around 80% of the cycles are spent in the matrix multiplication in all three versions. Since QR algorithm and Householder algorithm both have repeated matrix multiplications, using OpenMP+MPI for matrix multiplication should improve performance. Also, changing the 1D MPI row tiling and using a 2D tiling should also help in boosting the performance.
More overlap of communication and computation would also help in speeding up the program. Since QR is an iterative algorithm, we can think of initiating communication for the next iterations in the present iteration.