Parallelization of $\mathbf{0^{th}}$ Order Generalised Mode Acceleration Method

Implementation of OpenMP, MPI and CUDA Parallelizing Algorithms

Abhinandan Kumbhar- 193109007 R. Nitin Iyer - 19310R001 Jalaj Gupta - 19310R002 Saurabh Pal - 193104001

***Abstract*—For large systems, it is a common practice to implement model order reduction to minimize computational effort. In this project a displacement response of the linear elastic cantilever beam subjected to harmonic point load at the point of application is estimated. Various parallelization technique has been used to expedite the calculation process. Model order reduction has been done using Guyan condensation on the 3782×3782 system to get reduced system of 500×500 . A 0^{th} order Generalized mode acceleration method proprosed by J. Rixen [1] was applied to the reduced system. First 10 modes were used to approximate the response. and GMAM was done using OpenMP, MPI and GPGPU programming, and the results were compared with the serial code and MATLAB codes to test for consistency and time reduction.**

***Index Terms*—GMAM, OpenMP, MPI, GPGPU, CUDA etc.**

I. Introduction

DESCRIBE: As shown in adjacent figure. A cantilever beam is rigidly held in a rigid wall at one end and at free end at one edge a harmonic force of frequency comparable to first natural frequency of the beam is being applied longitudinally and transversally. This problem has been modelled as 2D problem. So in figure above one plane section is being depicted which has been considered for further analysis. The plane has been discretized using quad elements by ABAQUS software, which resulted in 3782x3782 Mass and stiffness matrix. As per modal analysis it has been found that modal participation factors for first ten modes are significant enough for this analysis. So instead of carrying out our computation on such a large mass and stiffness matrices, these matrices has been condensed to smaller 500x500 order using Guyan condensation. This model order reduction has been carried out accurately on MATLAB. Then these reduced matrices has been used for further computation, on C language, for estimating response of the system. Zeroth order generalized mode acceleration method has been used in the code for response calculation. This serial code written for zeroth order mode acceleration method is further parallelized using open MP, MPI and CUDA programming language to reduce execution time. In this report comparative timing study has been done to check code performance and results are also compared with MATLAB results to check for data consistency and coherency.

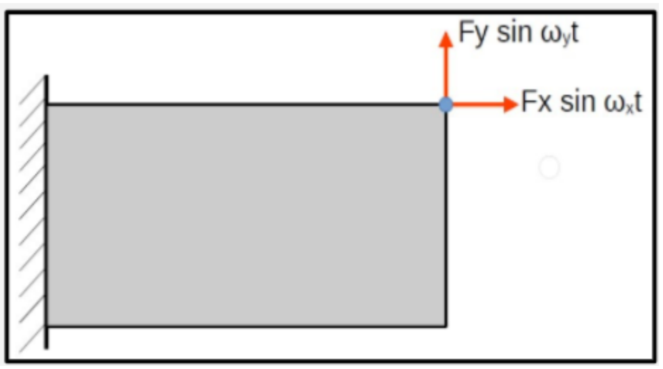


Fig. 1: Cantilever beam subjected to sinusoidal loading at the specified node

II. Theory for Generalized Eigenvalue Problem

Consider following generalized eigenvalue

$$\mathbf{K\Phi} = \lambda \mathbf{M\Phi}$$

To solve above eigenvalue problem we will use the method proposed by [2], consider first following standard eigenvalue problem for $\mathbf{M}$:

$$\mathbf{M\Phi_M} = \mathbf{\Phi_M}\lambda_\mathbf{M}$$

Where $\mathbf{\Phi_M}$, and $\lambda_\mathbf{M}$ are the eigenvector and eigenvalue matrices of $\mathbf{M}$, respectively. Then, we have:

$$\mathbf{M\Phi_M} = \mathbf{\Phi_M}\lambda_\mathbf{M} \implies \mathbf{\Phi_M^{-1}M\Phi_M} = \mathbf{\Phi_M^{-1}\Phi_M}\lambda_\mathbf{M}$$

$$\implies \lambda_\mathbf{M} = \mathbf{\Phi_M^T M \Phi_M}$$

Eigen-vector matrix has to be orthogonal because $\mathbf{M}$ is a symmetric matrix. If above equation is pre and post multiplied by $\lambda_\mathbf{M}^{-0.5}$. Then,

$$\lambda_\mathbf{M}^{-0.5}\mathbf{\Phi_M^T M \Phi_M}\lambda_\mathbf{M}^{-0.5} = \lambda_\mathbf{M}^{-0.5}\lambda_\mathbf{M}\lambda_\mathbf{M}^{-0.5} = \mathbf{I}$$

$$\hat{\mathbf{\Phi}}_\mathbf{M} = \mathbf{\Phi_M}\lambda_\mathbf{M}^{-0.5}$$

Let $\hat{K}$ be defined as

$$\hat{\mathbf{K}} = \hat{\mathbf{\Phi}}_\mathbf{M}^\mathbf{T}\mathbf{K}\hat{\mathbf{\Phi}}_\mathbf{M}$$

Since $\mathbf{K}$ is symmetric this matrix would also be symmetric. The eigenvalue problem for $\hat{K}$ is

$$\hat{\mathbf{K}}\mathbf{\Phi_K} = \lambda_\mathbf{K}\mathbf{\Phi_K}$$

where, $\mathbf{\Phi_K}$ and $\lambda_\mathbf{K}$ are the eigenvector and eigenvalue matrices of $\hat{\mathbf{K}}$. Pre-multiply above matrix by $\mathbf{\Phi_K^T}$

$$\mathbf{\Phi_K}^\mathbf{T}\hat{\mathbf{K}}\mathbf{\Phi_K} = \lambda_\mathbf{K}$$

Now plugging back expression for $\hat{\mathbf{K}}$ in above expression

$$\mathbf{\Phi_K}^{\mathbf{T}}\hat{\mathbf{\Phi}}_{\mathbf{M}}^{\mathbf{T}}\mathbf{K}\hat{\mathbf{\Phi}}_{\mathbf{M}}^{\mathbf{T}}\mathbf{\Phi_K} = \lambda_{\mathbf{K}}$$

$$\mathbf{\Phi_K}^{\mathbf{T}}(\mathbf{\Phi_M}\lambda_{\mathbf{M}}^{-0.5})^{\mathbf{T}}\mathbf{K}\mathbf{\Phi_M}\lambda_{\mathbf{M}}^{-0.5}\mathbf{\Phi_K} = \lambda_{\mathbf{K}}$$

$$\implies \mathbf{\Phi^T K \Phi} = \lambda_{\mathbf{K}}$$

where,

$$\mathbf{\Phi} = \hat{\mathbf{\Phi}}_{\mathbf{M}}\mathbf{\Phi_K} = \mathbf{\Phi_M}\lambda_{\mathbf{M}}^{-0.5}\mathbf{\Phi_K}$$

This expression also implies that $\mathbf{\Phi}$ also diagonalizes $\mathbf{B}$ and gives identity matrix. So,

$$\mathbf{\Phi^T M \Phi} = \mathbf{I} \implies \mathbf{\Phi^T M \Phi}\lambda_{\mathbf{K}} = \lambda_{\mathbf{K}}$$

$$\mathbf{\Phi^T M \Phi}\lambda_{\mathbf{K}} = \mathbf{\Phi^T K \Phi}$$

$$\mathbf{M\Phi}\lambda_{\mathbf{K}} = \mathbf{K\Phi}$$

This is expression for generalized eigenvalue problems. So, the following algorithm has been devised for calculation of the eigenvalues and eigenvectors-

1) $\mathbf{\Phi_M}, \lambda_{\mathbf{M}} \leftarrow \mathbf{M\Phi_M} = \lambda_{\mathbf{M}}\mathbf{\Phi_M}$
2) $\hat{\mathbf{\Phi}}_{\mathbf{M}} \leftarrow \hat{\mathbf{\Phi}}_{\mathbf{M}} = \mathbf{\Phi_M}\lambda_{\mathbf{M}}^{-0.5} \approx \mathbf{\Phi_M}(\lambda_{\mathbf{M}}^{-0.5} + \epsilon\mathbf{I})^{-1}$
3) $\hat{\mathbf{K}} \leftarrow \hat{\mathbf{K}} = \hat{\mathbf{\Phi}}_{\mathbf{M}}^{\mathbf{T}}\mathbf{K}\hat{\mathbf{\Phi}}_{\mathbf{M}}$
4) $\mathbf{\Phi_K}\lambda_{\mathbf{K}} \leftarrow \hat{\mathbf{K}}\mathbf{\Phi_K} = \lambda_{\mathbf{K}}\mathbf{\Phi_K}$
5) $\lambda \leftarrow \lambda = \lambda_{\mathbf{K}}$
6) $\mathbf{\Phi} \leftarrow \mathbf{\Phi} = \hat{\mathbf{\Phi}}_{\mathbf{M}}\mathbf{\Phi_K}$
7) $\mathbf{\Phi}$ and λ

While solving generalized eigenvalue problem $(\mathbf{K}, \mathbf{M})$, we have to solve two simple eigenvalue problems of $\mathbf{M}$ and $\hat{\mathbf{K}}$. We have used QR decomposition for solving this problem. We will now study QR decomposition in details, followed by GMAM method

III. Theory of QR decomposition with Gram-Schmidt

We now perform QR decomposition of a matrix $\mathbf{A}$ as $\mathbf{QR}$, where $\mathbf{Q}$ is orthogonal matrix and $\mathbf{R}$ is upper triangular matrix. The computation of $\mathbf{Q}$ is iterative process in which each iteration deals with calculation of k^{th} column of Q by using below formula

$$\mathbf{Q_k} = \mathbf{A_k} - \sum_{n=0}^{k-1}(\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$$

The $\mathbf{Q_k}$ is then used to rotate $\mathbf{A_k}$ and proceed to next iteration.

IV. Theory of GMAM for 0^{th} order

After obtaining the eigenvalues and eigenvectors from the algorithm explained in the above section, the response can be obtained as-

$$q_n = \sum_{r=1}^{k}\frac{x_r x_r^T F}{\omega_r^2 - \omega^2}[\,\sin\omega t - \frac{\omega}{\omega_r}\sin\omega_r t]$$

where, x_r is the rth eigenvector , k is the number of modes considered & being 10 in our case. ω is forcing frequency, ω_r is square root of r^{th} eigenvalue. The F is vector of force amplitudes.

V. Potential regions for Parallelization

After studing the algorithm of solving generalized eigenvalue problem and GMAM, we found out that major portion of code deals with matrix multiplication. So, we decided to perform parallelization of matrix multiplication.

While solving generalized eigenvalue problem $(\mathbf{K}, \mathbf{M})$, we have to solve two simple eigenvalue problems of $\mathbf{M}$ and $\hat{\mathbf{K}}$. We have used QR decomposition for solving this problem, Which is iterative process. Suppose we perform L iterations of QR decomposition for solving single simple eigenvalue problem.The QR decomposition in itself needs N i.e size of matrix, iterations. Then it means we need to do $2 * L * N$ iterations in total for solving two simple eigenvalue problems. We can parallelize this QR decomposition algorithm and save a lot of time.

Depending upon above study, we have decided to parallelize QR decomposition and matrix multiplication using OpenMP, OpenMPI and Cuda.

VI. Implementation of OpenMP, MPI and GPGPU algorithms

OpenMP :
The OpenMp parallelization is done by observing the 'for' loops and making the appropriate variables as private to each thread.

MPI :
Parallelization of QR decomposition

$$\mathbf{Q_k} = \mathbf{A_k} - \sum_{n=0}^{k-1}(\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$$

Formation of Q is parallelized by distribution of

$$\sum_{n=0}^{k-1}(\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$$

among the processors. Each process has to perform computation for k/cores columns. Root process will normalize and assemble the columns of Q. The Q will then be used to rotate the mass of transformed matrix, and then next iteration will be performed.

Using matrix multiplication, we'll transform the K matrix, and perform QR decomposition on transformed K matrix.

Parallelization of Matrix Multiplication ($C = AB$) **:-** Share B matrix with all the processors. Send sets of rows to different processors , and perform multiplication of those rows with the B matrix. Receive the computed rows at root processor, and assemble all the rows at appropriate positions.

$$\mathbf{Setsofrows} = \frac{\mathbf{No.ofrowsofA}}{\mathbf{No.ofProcessors}}$$

CUDA parallelization
As explained in the previous section, Two major areas are identified where we can apply parallelization. Kernel functions

are written for below two codes and used them wherever they are required.

1) Matrix multiplication
2) QR decomposition

CUDA kernel for matrix multiplication (AB=C)

Suppose we have matrix C of size $N \times N$, We have launched a kernel having grid of $N \times N$ size, each grid block having N threads.

$matmult <<< grid(N, N), N >>> (A, B, C)$

Each block corresponds to one element of C matrix and calculates one entry of C matrix C[i][j]. Within each block, N threads will help compute dot product i^{th} row of A and j^{th} column of B.

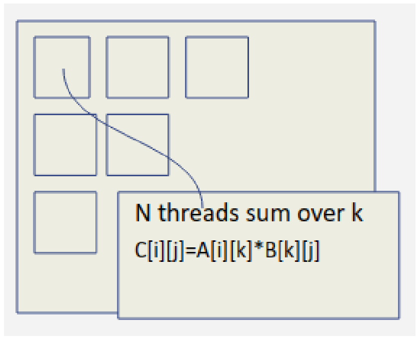


Fig. 2: Cuda Kernel for matrix multiplication

CUDA kernel for QR decomposition (A=QR)

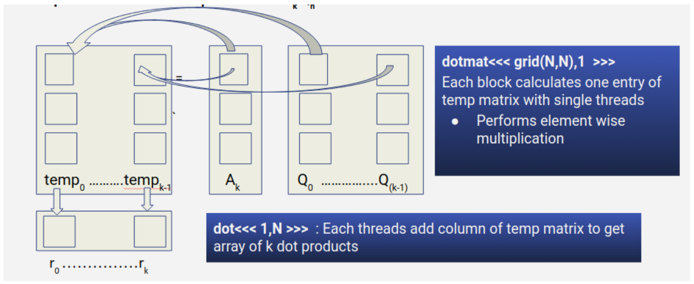
QR decomposition is performed in n iterations. k^{th} iteration corresponds to calculation of k^{th} column of Q matrix. So, below explained kernels are launched for n times for computing each column of Q. Formula for calculating k^{th} column of Q is as follows:

$$\mathbf{Temp_k} = \mathbf{A_k} - \sum_{n=0}^{k-1} (\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$$

$$\mathbf{Q_k} = \mathbf{Temp_k}/SQRT(\mathbf{Temp_k}.\mathbf{Temp_k})$$

As we can see, k dot products have to be performed for k^{th} iteration. Dot product of $\mathbf{A_k}$ and $\mathbf{Q_n}$ are then multiplied with $\mathbf{Q_n}$ i.e $(\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$ and sum them all. Then, substracted the summation from $\mathbf{A_k}$ to get vector $\mathbf{Temp_k}$. Unit vector of $\mathbf{Temp_k}$ is stored in k^{th} column of $\mathbf{Q}$. All this is done in three steps.

1) Step 1:Calculation of k dot products i.e. $\mathbf{A_k Q_n}$.

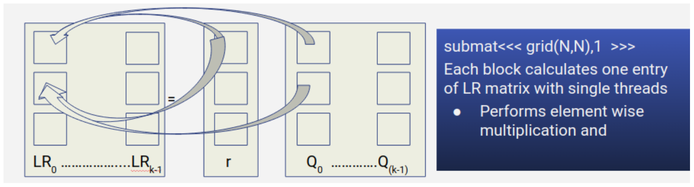

temp$_0$ ....temp$_{k-1}$
=
A$_k$
Q$_0$ ....Q$_{(k-1)}$
r$_0$...............r$_k$

dotmat<<< grid(N,N),1 >>>
Each block calculates one entry of temp matrix with single threads
• Performs element wise multiplication

dot<<< 1,N >>> : Each threads add column of temp matrix to get array of k dot products

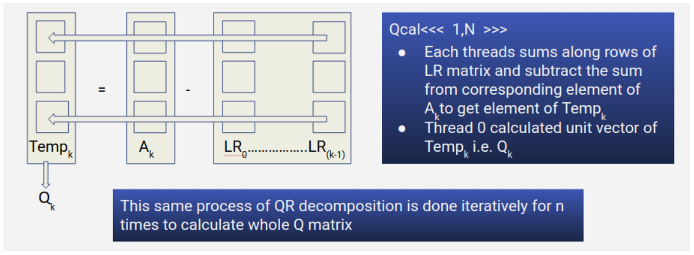
Fig. 3: Cuda Kernels for step 1

2) Step 2: Calculate matrix $(\mathbf{A_k}.\mathbf{Q_n})\mathbf{Q_n}$.
3) Step 3: Calculate $\mathbf{Temp_k}$ and $\mathbf{Q_k}$.


=
LR$_0$ ....LR$_{k-1}$
r
Q$_0$ ....Q$_{(k-1)}$

submat<<< grid(N,N),1 >>>
Each block calculates one entry of LR matrix with single threads
• Performs element wise multiplication and

Fig. 4: Cuda Kernel for step 2



Qcal<<< 1,N >>>
• Each threads sums along rows of LR matrix and subtract the sum from corresponding element of A$_k$ to get element of Temp$_k$
• Thread 0 calculated unit vector of Temp$_k$ i.e. Q$_k$

This same process of QR decomposition is done iteratively for n times to calculate whole Q matrix

Fig. 5: Cuda Kernel for step 3

VII. Results and Discussion

System specifications : The code were run on the CDAC Param Sanganak. For OpenMp, number of threads used were equal to number of cores allocated. Similarly, for OpenMPI, number of cores assigned were same as number of processes.
Time study : The serial code took 48 mins to run and below is the time study for different techniques.

OpenMP (min)	OpenMPI (min)	Cuda (min)
8 thds - 9.1	8 PEs - 9.4	8 cores - 11.02
12 thds - 10.4	12 PEs - 9.3	12 cores - 10.9
16 thds - 12.5	16 PEs -10.7	16 cores - 9.7
20 thds - 15.03	20 PEs - 11.1	20 cores - 10.1

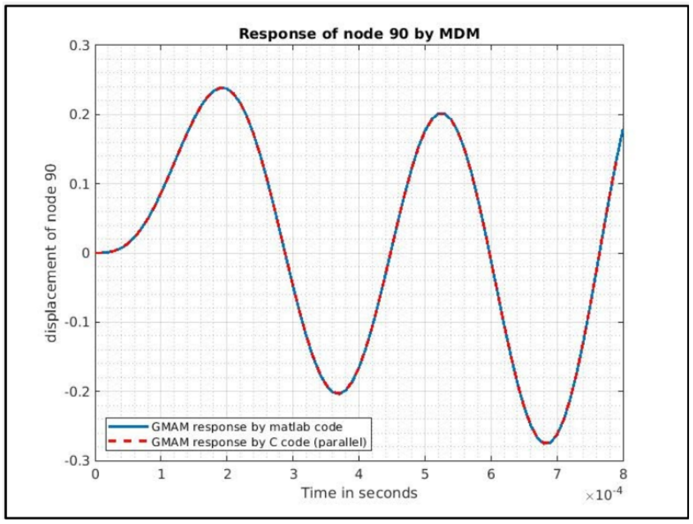


Fig. 6: Response obtained from code plotted in Matlab

The response of node 3781 (90 in reduced matrix) is plotted and its error value is also plotted. we can see that the error is of the order 10-4 (0.1%)
The same problem was solved using Matlab code to confirm that the parallelized code are having data consistency and coherency.

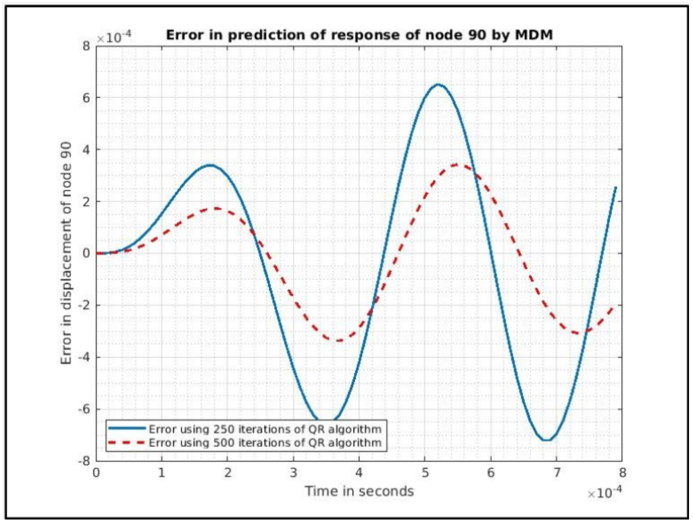


Fig. 7: Error variation with changing number of iterations used for predicting response

In case of MPI programming for parallelization, the increase in time as number of processes goes on increasing is counter intuitive , but it may have happened because decrease in time of computation by each process is not significant as compared to time increase due to increase in data sharing. Also, for OpenMPI, the time taken by root process is bottle-neck. Since after distributing the workload equally amongst all processes the extra remainder work has to be done by root process.

We can observe that for CUDA program, time remains almost same even if we increase the number of cores. It may have happened because we are launching the same kernels in every case. We are launching same threads and same numbers of blocks for all various numbers of cores.

As explained by Amdalh's law, We do not get speed as expected. Since the every iteration of QR decomposition is dependent on previous iteration, we have limitation on speed up, still we can see speed-up upto 4-5 times.

References

[1] D. Rixen, "Generalized mode acceleration methods and modal truncation augmentation," in *19th AIAA Applied Aerodynamics Conference*, p. 1300, 2001.

[2] B. Ghojogh, F. Karray, and M. Crowley, "Eigenvalue and generalized eigenvalue problems: Tutorial," *arXiv preprint arXiv:1903.11240*, 2019.