Nonlinear Finite Element Application Optimization

Tongge Wu[1], Debbie Liang[2], Zhuoming Zhang[2]

[1] *Department of Mechanical Engineering, University of California, Berkeley*
[2] *Department of EECS, University of California, Berkeley*
May 13th, 2020

1 Introduction

Finite Element Method (FEM), as a numerical method to solve partial differential equation (PDE), is widely used in numerical analysis and engineering analysis (e.g. mechanical, civil, and material engineering field). Starting with Galerkin method, FE method discretizes the problem domain into *elements*, each element has multiple *nodes* (e.g. triangle or quadrilateral element). Based on the properties of the PDE, FE method then constructs local matrix (e.g. mass matrix, stiffness matrix) for each element. After this, all local matrices are *assembled* into a global matrix using the connectivity between element (i.e. the way elements share nodes). The key step (math core) of FE method boiled into solving a linear system or a group of linear systems. Moreover, if the property of PDE involves nonlinearity, then the math core of FE method becomes solving a nonlinear system or a group of nonlinear system.

Within the scope of this project, we aimed to parallelize a mechanical FE simulation using OpenMP and GPU, where a nonlinear material (rubber) is under large-deformation (i.e. nonlinear), and the transient (i.e. time-dependent) mechanical simulation involves two loading stages - pulling and vibrating.

Following this introduction, in Section 2 we will elaborate on the numerical framework and general code scheme of the FE simulation. Section 3 lists various techniques to accelerate the FE simulation, along with the improvements in performance. Following that, Section 4 shows the scalability of our code. And Section 5 shows the conclusion.

2 Math Framework and Code Scheme

In this project, we used FE method to simulate the vibration of a two-dimensional solid rubber-like rectangular block ($X \times Y = 2m \times 10m$). The bottom surface of the block was fixed on the flat ground, and the top surface of the block was pulled into a prescribed displacement ($u_Y = 1m$) , then released to introduced the free vibration. The block was discretized into a regular mesh consisting of $EX \times EY$ elements. Each linear quadrilateral element has 4 nodes, resulted in $NX \times NY = (EX + 1) \times (EY + 1)$ nodes totally. Since the bottom nodes were boundary conditions (fixed in both directions), and the top nodes were boundary condition in the pulling stage, the total degree-of-freedom (DOF) was roughly $NX \times NY \times 2$.

Briefly, the FE method resulted in a group of nonlinear ordinary differential equations (ODEs) for this particular simulation,

$$[\boldsymbol{M}][\hat{\boldsymbol{a}}_{n+1}] + [\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})] = \boldsymbol{0}. \tag{1}$$

where the $\boldsymbol{M}$ was the mass matrix, it was sparse, symmetric and positive definite. $\hat{\boldsymbol{a}}_{n+1}$ was the acceleration of each DOF. $\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})$ was the stress divergence of each DOF, which was a nonlinear function of displacement ($\hat{\boldsymbol{u}}_{n+1}$) of each DOF. Eq.1 was essentially Newton's second law applied at a continuum level. Newmark integrator was then

used to convert the group of nonlinear ODEs (Eq.1) into a group of nonlinear algebraic equations, plus some update operations (not shown in the text),

$$\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}) = \frac{1}{\beta\Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1} + \boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1}) - \boldsymbol{M}[(\hat{\boldsymbol{u}}_n + \hat{\boldsymbol{v}}_n\Delta t)\frac{1}{\beta\Delta t^2} + \frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n] = \boldsymbol{0}. \tag{2}$$

where β was the Newmark parameter. Δt was the time step. $\hat{\boldsymbol{v}}_n$ was the velocity of each DOF. Subscripts represented solution at different time steps (old n versus new $n+1$). On a deeper level, we used Newton-Raphson nonlinear solver to solve the group of nonlinear algebraic equation (Eq. 2), which required us to form a tangent matrix (gradient) in each Newton-Raphson iteration,

$$\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)} = D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})\backslash(-\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})). \tag{3}$$

where $\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)}$ was the update part of the solution. $D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})$ was the tangent matrix, which was sparse, symmetric and indefinite. Backslash represented a linear solver.

All intermediate math and other FE method details were in Section 6 (Appendix), including the discretization, updating operations of Newmark integrator, and how the tangent matrix was formed, etc.

Algorithm 1: Code Scheme (potential optimization colored in blue)

```
Construct mesh - node, element, boundary condition, material constant;
Construct Mass matrix (S.P.D. SpM, OpenMP);
% Pull Stage;
while time < T_1 do
    Prescribe boundary condition;
    % Call Newmark-integrator function;
    Save old u, v, a;
    Form residual based on initial guess u;
    % Call Newton-Rasphson nonlinear solver;
    while norm(residual) > tol AND iter < maxiter do
        Form tangent matrix (Sym SpM, OpenMP);
        A few SpMM, SpMV operations;
        δu = PARDISO(tangent matrix, residual) (MKL);
        u = u + δu;
        Form residual based on u;
    end
    %Newton-Rasphson solver converges;
    v, a update;
end
% Vibration Stage;
while time < T_2 do
    % Call Newmark-integrator function;
end
```

To code such a numerical framework, the following scheme was developed (see Algo.1). The initialization part included setting the mesh, node coordinates, the connectivity array and the boundary condition. This part was designed to be generic such that it could also accept other FE input files. Since the mass matrix was a constant matrix, it was created only once at the beginning of the scheme. In the design space of the mass matrix, it was a symmetric sparse matrix, which could reduce the memory allocation. The assembling process of the mass matrix was independent between each element, thus, it could be accelerated using threaded programming such as OpenMP.

Then, we called the Newmark integrator for time stepping. Before the while loop of Newton-Rasphson solver, we guessed a solution and formed an initial residual (RHS of Eq.2). Within the while loop of Newton-Rasphson solver, the solver would form the tangent matrix and solve for a solution update (RHS and LHS of Eq.3). In the design space of the tangent matrix, it was symmetric and sparse, which could also reduce the memory allocation. The assembling

process of the tangent matrix was also independent between each element, hence a threading programming could be used. In the design space of the linear solver, for simplicity, a direct sparse solver from Intel Math Kernel Library (MKL) was adopted and could be tuned. The dimension of $Ax = b$ linear system (number of rows of A) was $NX \times NY \times 2 - NX \times 3$ in pulling stage, and $NX \times NY \times 2 - NX \times 2$ in vibrating stage.

If the Newton-Raphson solver converged, hence the inner while loop broke, the Newmark integrator would update related terms and step forward. In this scheme of code, the vibration stage was almost the same as the pull stage, but with a slightly different boundary condition.

3 Parallel Computing Optimizations

DOFs	Applied Optimization	Threads	Wall-time (s)
2,560	Matlab	1	1515.89
2,560	Naive C++	1	97.78
2,560	Sparse Matrix	1	21.94
2,560	Symmetric Matrix	1	21.26
16,000	Sequential	1	148.04
16,000	OpenMP	68	87.20
16,000	OpenMP + MKL	68 + 68	64.93
16,000	OpenMP + MKL	68 + 16	53.51
16,000	PARDISO: Parallel reordering and solving	68 + 16	53.01
16,000	Parallel Sort	68 + 16	37.86
16,000	PARDISO: Reuse Permutation	68 + 16	26.19

Table 1: Different optimization techniques and corresponding running time on one Cori KNL node. All test cases had 30 Newmark integrator iterations. The second column indicated **accumulated** applied optimization, this was the reason why the last column was monotonically decreasing for the same DOFs. Additionally, the third column indicated the code was in sequential (1), OpenMP threading (68), or OpenMP and MKL threading (a + b) mode.

According to the Algo.1 in Section 2, in general, the global mass and global tangent matrix assembling process can be highly parallel. Moreover, the linear solver can be optimized using a parallel technique to some extend. Thus, our team spent most of the time investigating these two domains.

3.1 C++ Benchmarking

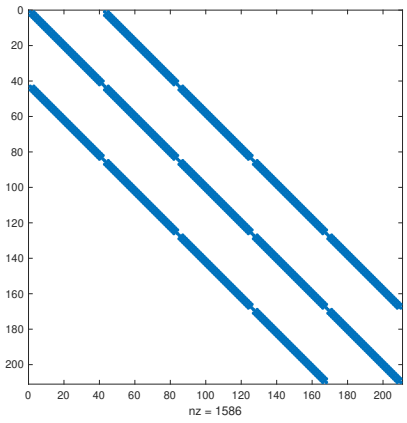
The FE simulation was originally developed in the Matlab environment. In Matlab, there was no explicit parallel region (i.e. `parfor()`) in the code. However, in the matrix multiplication and linear solver region, Matlab might utilize Intel MKL functions (internally) to accelerate the computation. In terms of the matrix storage, the global mass matrix and tangent matrix was stored as sparse matrices in coordinate format (COO) in Matlab.

To get an accurate baseline for later development, the program was migrated naively into the C++ environment [5]. The migration process cost us about 2.5 weeks. In this naive implementation, all matrices were in column major and stored as dense matrices. As for the linear solver, a general dense symmetric solver (`LAPACKE_dsysv()`) was used.

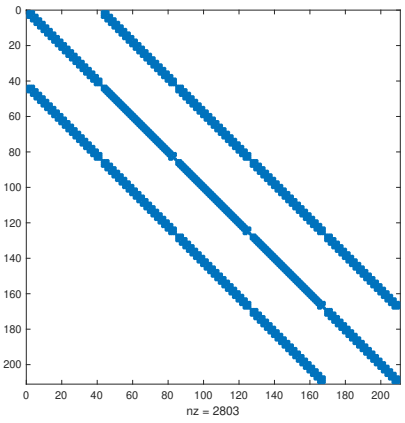
To compare the performance between Matlab and C++ environment, we constructed an example FE simulation running one Cori KNL node. In such example FE simulation, there were 2,560 DOFs, and Newmark integrator stopped at 30 time steps (10 in the pull stage, 20 in the vibration stage). The naive C++ implementation cost 93.55% wall-time less than the original Matlab implementation (Table 1).

3.2 Sparsity

In the naive implementation, all matrices were in column major and stored in dense format. Such implementation was extremely inefficient in memory usage and later matrix computation. Since in a FE simulation, only nodes (DOFs, to be specific) within one element would contribute a non-zero entry in the global mass or tangent matrix. In another word, if each row in the global mass or global tangent matrix represented a linear equation of a DOF, only DOFs that were in the same element would contribute a non-zero column entry. Here, Fig. 1 showed the sparsity of the global mass and tangent matrix in an example FE simulation of 210 DOFs.



(a) Global Mass Matrix



(b) Global Tangent Matrix

Figure 1: Sparsity of global matrices in the FE simulation, each non zero entry was a solid point in the plot, and zero entry was blank in the background

To get a sparse matrix representation, we first created a pair of < index, value > for any global (i.e. large) matrix. The "index" was the entry's column major index in the matrix. The "value" was the actual value of that entry. Secondly, the pair was sorted by "index" for two purposes: to remove overlapped entries with ease and to fit the requirement of the MKL parallel sparse direct solver (PARDISO). After sorting, the pair was traversed to remove any overlapped entry (i.e. accumulating value that had the same index). A byproduct of the traversing was the number of non-zero entry (nnz) in the matrix. Based on the updated pair, three arrays (`row-i`,`col-j`,`val`) were constructed to create the sparse matrix in coordinate format (COO) (`mkl_sparse_d_create_coo`), since the COO matrix was natural in a FE assembling process. COO matrix was also a good format to check correctness in a FE simulation. Then, the COO matrix was converted into compressed sparse row format (CSR) using `mkl_sparse_convert_csr` function, because CSR matrix was the only sparse format PARDISO accepted. Moreover, PARDISO required the CSR matrix to have increased column index within one row, this was achieved in the previous sorting process.

To compare the dense implementation and sparse implementation, we used the same test case as section 3.1. The sparse implementation cost 77.56% less wall-time than the dense implementation (Table 1).

3.3 Symmetry

Due to the nature of FE method, both mass and tangent matrices were symmetric, using such a feature would be able to save almost half memory compared to non-symmetric cases. Meanwhile, the computation time (for matrix operation) could also be saved.

Using the same example FE simulation in section 3.1, we found that using symmetric optimization yield another 3.1% performance increase in computation time (Table 1). More importantly, we found that such optimization saved 28.66% memory usage in the mass and tangent matrix creation regions.

3.4 OpenMP Threading

As discussed in the Algo.1, the mass matrix and tangent matrix assembling process could be highly parallel using threading programming like OpenMP. After previous optimizations, using HPCToolKit [1], we indeed found that tangent matrix assembling process was the most computational heavy region in our code, not only because the tangent matrix was large to construct, but it was repetitively assembled at each Newton-Raphson iteration.

To use threading programming in the tangent matrix assembling process, we first identified the long and independent `for` loop, which was the loop over element stress divergence residual and element (local) tangent matrix. We also identified the potential data racing region, which was the overlap between element stress divergence, as well as the populating pair of <index, value> for the global tangent matrix. Then we used `omp_lock_t()` (for residual) or `#pragma omp critical` (for pair) to protect the corresponding data structure. Moreover, the sorting process was also accelerated using OpenMP, which will be discussed in detail in later section.

During our debugging stage, we tried to using threading programming to other sections of our code, such as global mass matrix assembling, some matrix-vector operations in the Newmark integrator, or the meshing constructing process. However, those efforts were not beneficial in terms of the code performance.

To measure the performance improvement of using OpenMP, we constructed a larger example of FE simulation compared to the one in section 3.1. Here we used an example of FE simulation of 16,000 DOFs. The Newmark integrator also had 30 iterations. Compared to the sequential code, our OpenMP optimization yielded a 41.1% less computation time, where we used 68 threads on one Cori KNL node (Table 1).

3.5 MKL Threading

Other than the explicit OpenMP threading used in section 3.4, all functions from MKL also supported threading programming. The MKL threading introduced some challenges in our development. For example, for a regular MKL function call, if the input was a large sparse matrix (such as global mass matrix) or a long vector (such as global residual vector), we may need a large number of threads for the best performance. However, if the MKL function operated on a small matrix (such 8×8 local tangent matrix), we may need only 1 thread to reduce the threading overhead. More complicated, if the MKL function was called inside an OpenMP parallel region, we need some nested threading technique to optimize the performance.

By inspection of our code and a lot of try-and-errors, we decided to use 68 OpenMP threads in the tangent matrix assembling process, within the parallel region each MKL function used one thread for any BLAS call. In the Newton-Raphson solver, any MKL function associated with sparse matrix-vector multiplication, sparse matrix-matrix addition, or vector norm used 16 threads.

This part of work cost us about 2 weeks to finish, since it was essentially sweeping a 3D (OpenMP, MKL, Nested) mesh of choices and identifying the best combination. Additionally, compiler option and linking libraries were not trivial when OpenMP and MKL were involved at the same time.

Using the same example FE simulation in section 3.4, our results showed that 68 + 16 (OpenMP + MKL) combination was 38.64% better than the 68 + 1 (OpenMP + MKL) combination, and 17.59% better than the fully threaded (68 + 68, OpenMP + MKL) combination in terms of the computing time (Table 1).

3.6 PARDISO

Although looking for the best linear solver was not the main objective of this project, we applied some optimizations on the linear solver used in the Newton-Raphson iteration. Consider the scale of the FE simulation we set in section 3.4, direct solver such as PARDISO from MKL was used [2, 4, 3].

A naive PARDISO was used since the CSR format sparse tangent matrix was adopted in our code. To accelerate the PARDISO solver, we tuned the `iparm` in PARDISO which controlled behaviors of PARDISO at different stages: reordering and symbolic factorization (`phase 11`) - numerical factorization (`phase 22`) - forward and backward solve (`phase 33`)- free buffer (`phase -1`). Specifically, we used the parallel version of the nested dissection algorithm in the reordering stage, and parallel substitution in the solving stage.

Using the same FE simulation example in section 3.4, those parameters changes in PARDISO yielded a reduction of 1% in computation time compared to naive PARDISO implementation (Table 1).

Additionally, since the PARDISO was used on the same sparse pattern tangent matrix in the Newton-Raphson iteration, we could keep the permutation information on the first call of PARDISO, and reuse the permutation (which was an array) for later calls of PARDISO. Such optimization yielded another 30.82% less computing time compared to parallel reordering and solving PARDISO implementation (Table 1).

3.7 Parallel Sort

As discussed in section 3.2, a sorting process on the pair of <index, value> was necessary in our CSR format matrix construction. After applying previous optimizations, HPCToolKit installed on Cori suggested us the sorting took the most significant portion of our program running wall-time, within one time step in Newmark integrator.

With the availability of threading programming using OpenMP, we then implemented a paralleled merge sort. This algorithm recursively divided the sorting arrays into subsets and assigned them to different threads (tasks) until each subset was less than 32 elements. Then, each leaf thread sorted their corresponding sub-array. After the sub-array sorting was done, the in-place merging process happened also recursively. Both dividing and merging were in $\mathcal{O}(\log N)$ time, where N was the length of the array.

Using the very same example FE simulation in section 3.4, we found that implementing the parallel sort reduced the computation time by 28.58% compared to sequential sorting (Table 1).

3.8 GPU, or not GPU

In our initial proposal, we were planning to use a graphic processing unit (GPU) to accelerate some parts of our simulation. For example, in the global mass (or tangent) matrix assembling process, we could use GPU to loop over each element. Because each local mass (or tangent) matrix was light to compute (not larger than 8×8, usually 4×4), which fitted the light-task "requirement" for each GPU streaming processor (SP) (i.e. thread). And each local mass (or tangent) matrix was independent of other local mass (or tangent) matrix, but shared the identical math operations, which fitted the single instruction multiple data (SIMD) "requirement" of GPU. Another example where the GPU could overtake CPU was the sparse matrix-matrix (SpMM) and spare matrix vector (SpMV) operation in the Newton-Raphson solver. However, we decided not to follow this plan after carefully reviewing and analyzing our code.

First, in our mass (or tangent) matrix assembling process, the number of loops was the number of elements in the FE mesh, such number was small compare to the number of SMs on a P100 GPU. To effectively use GPU for the assembling process, we need a much larger number of elements, which require share-memory programming for other parts of the coding.

For a similar reason, if we used GPU to compute the SpMM or SpMV, the matrices had to be large enough to offset the communication overhead caused by transferring data between CPU and GPU. Using HPCToolKit installed on Cori, we found that the SpMM and SpMV implemented by MKL was already pretty efficient, and they took only 0.8% of the total computation time. In this situation, using CUDA programming would probably degenerate our code performance. Therefore, we decided to reserve the CUDA programming for later development of this project.

4 Performance

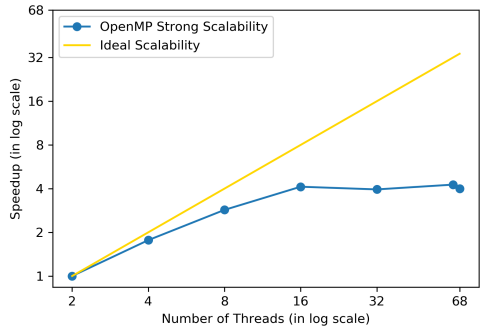
OpenMP Threads	Wall-time (sec)
2	1581.80
4	895.36
8	554.18
16	383.69
32	400.38
64	371.11
68	395.64

(a) Relation between number of threads and running time. Data was collected on one Cori KNL node, where Newmark integrator had 30 iterations. The FE simulation had 225,000 DOFs.

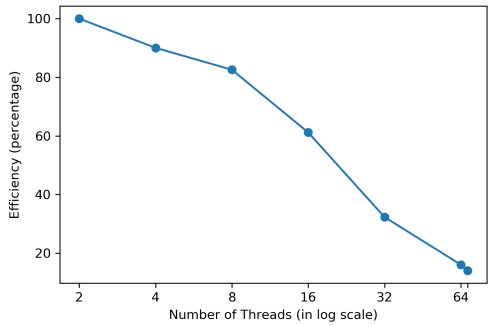
Total FE DOFs	OpenMP Threads	Wall-time (sec)
6,760	2	55.63
12,960	4	61.79
26,010	8	67.38
53,290	16	90.94
106,090	32	172.19
213,160	64	346.45
225,000	68	395.64

(b) Relation between number of threads and running time. Data was collected on one Cori KNL node, where Newmark integrator had 30 iterations. The FE simulation had about 3300 DOFs per thread.

Table 2: Performance analysis tables of strong scaling and weak scaling.



(a) Strong scaling: Speedup as a function of number of threads running for a fixed FE simulation scale - 225,000 DOFs. The golden line indicated the ideal speedup with slope of 1. Blue dots were data in Table.2a.



(b) Weak scaling: Efficiency as a function of number of threads running for a FE simulation scale - 3300 DOFs per thread. Blue dots were one-node data in Table.2b

Figure 2: Performance analysis figures of strong scaling and weak scaling

4.1 Strong Scaling

To obtain the strong scaling characteristics of our code, we fixed our FE simulation scale to be 225,000 DOFs. Then, the number of OpenMP threads was progressively increasing. The computation time was collected in Table 2a. If we defined the "Speedup" as

$$\text{Speedup} = \frac{\text{Walltime(p threads)}}{\text{Walltime(1 thread)}}$$

we could plot the speedup against the number of OpenMP threads, see Fig.2a. It was not difficult to observe that when the number of threads was less than 8, the speedup was relatively close to the ideal (linear) speedup. When the number of threads went up to 16, the speedup deviated more from the ideal speedup. If the number of thread was larger than 16, the speedup of our code was flat at about 4.3.

Those observations fitted the Amdahl's law. By the Amdahl's law, the theoretical speedup was

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

where S was the theoretical speedup, p was the parallel portion of the code, s was the number of processors. Plug in our "flat" speedup 4.3, we found that **76.74% of our code was done in parallel**. For those inherent sequential (such as Newmark integrator, Newton-Raphson nonlinear solver, etc.) code, there was no way we can parallelize them.

4.2 Weak Scaling

To obtain the weak scaling characteristic of our code, we assigned each thread approximately 3,300 DOFs, then progressively increased the number of threading starting at 2. The reason why the DOF per thread was "approximate" was we started at 68 threads (full threading on one Cori KNL node) with 225,000 DOFs, then scale down to 2 threads. Meanwhile, due to the physical dimension of the simulation, we preserved the number of elements in X direction to be 1/5 of the number of elements in Y direction, then we rounded the final calculation to be the nearest integer. The computation time was collected in Table.2b. If we defined the Efficiency as

$$\text{Efficiency} = \frac{\text{Time(1 thread)}}{\text{Time(p threads)}}$$

we could also plot the efficiency as a function of number of threads, see Fig.2b. Clearly, we observed that our efficiency decreased when the number of threads increased. This result suggested that an increasing number of threads might increase overhead of communication. It also suggested that the OpenMP lock and OpenMP critical region would degenerate the efficiency when we had a large number threads. Another reason we had such a efficiency behavior was the serial part of code, which would gradually take more time to run when we increased our DOFs in the FE simulation.

4.3 Run-time Breakdown

To quantify the communication time of our OpenMP code, we used HPCToolKit installed on Cori [1]. We fixed the FE simulation scale to be 16,000 DOFs and progressively increased the number of threads. As shown in Table 3, we observed that the communication time fraction increased significantly when the number of threads reached 32. The result suggested that when the number of threads increased, there might be some scheduling overhead associated with the parallel region in our code. More than this, we had a critical region that used locks to avoid the data racing condition. This section would take a greater percentage of time when the number of threads increased.

Threads	Communication percentage
4	0.45%
32	0.6%
68	6.8%

Table 3: Relation between the computation time spent and the number of threads. All cases run on one Cori KNL node, where Newmark integrator had 30 iterations.

5 Conclusion

In the scope of this project, we migrated, analyzed, parallelized, optimized, and characterized a nonlinear FE simulation. We utilized the nature of the FE simulation (i.e. the structure of the global mass and tangent matrix) to improve the code performance. In addition to that, OpenMP threading and MKL threading were thoroughly investigated and combined to accelerate multiple regions in our code. PARDISO from MKL was also studied and optimized to fit the requirement of the nonlinear solver in the FE simulation. We achieved a 26.19 seconds walltime on a 16,000 DOFs simulation compare to 1515.89 seconds walltime on a 2,560 DOFs simulation at the beginning of the project. Such improvement enabled us to run the nonlinear FE simulation on a much finer spatial and temporal resolution given the same running time, or to be able to run a much longer simulation given the same supercomputer allocation. Thus,

this could lead to new scientific findings in mechanical engineering research. The strong scalability enabled us to calculate the parallel portion we optimized, while the weak scalability revealed some potential optimization in the future. In conclusion, we achieved a reasonable parallel version of a nonlinear mechanical FE simulation, and our team learned much valuable knowledge and skills about Parallel Computing.

References

[1] L. Adhianto et al. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701. DOI: 10.1002/cpe.1553. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553.

[2] Arne De Coninck et al. "Needles: Toward Large-Scale Genomic Prediction with Marker-by-Environment Interaction". In: 203.1 (2016), pp. 543–555. ISSN: 0016-6731. DOI: 10.1534/genetics.115.179887. eprint: http://www.genetics.org/content/203/1/543.full.pdf. URL: http://dx.doi.org/10.1534/genetics.115.179887.

[3] D. Kourounis, A. Fuchs, and O. Schenk. "Towards the Next Generation of Multiperiod Optimal Power Flow Solvers". In: *IEEE Transactions on Power Systems* PP.99 (2018), pp. 1–10. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2017.2789187. URL: https://doi.org/10.1109/TPWRS.2017.2789187.

[4] Fabio Verbosio et al. "Enhancing the scalability of selected inversion factorization algorithms in genomic prediction". In: *Journal of Computational Science* 22.Supplement C (2017), pp. 99–108. ISSN: 1877-7503. URL: https://doi.org/10.1016/j.jocs.2017.08.013.

[5] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html. lulu.com, 2011. ISBN: 978-1-257-99254-6.

6 Appendix

6.1 FE Formulation

Used the total Lagrangian finite element method, we need to solve the linear momentum balance equation at time step t_{n+1}

$$\int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{a}_{n+1} dV + \int_{R_0} \frac{\partial \boldsymbol{\xi}_{n+1}}{\partial \boldsymbol{X}} \cdot \boldsymbol{P}_{n+1} dV = \int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{b}_{n+1} dV + \int_{\Gamma_{q_0}} \boldsymbol{\xi}_{n+1} \cdot \bar{\boldsymbol{p}}_{n+1} dA. \tag{4}$$

When there was no body force and prescribed traction, Eq.4 could be simplified

$$\int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{a}_{n+1} dV + \int_{R_0} \frac{\partial \boldsymbol{\xi}_{n+1}}{\partial \boldsymbol{X}} \cdot \boldsymbol{P}_{n+1} dV = 0. \tag{5}$$

The first term was the acceleration, and the second term was stress divergence.

Using discretization, within each 4-node quadrilateral element, we could have

$$\begin{aligned}
\boldsymbol{\xi}^e_{n+1} &= [\boldsymbol{N}^e][\hat{\boldsymbol{\xi}}^e_{n+1}] \\
\boldsymbol{a}^e_{n+1} &= [\boldsymbol{N}^e][\hat{\boldsymbol{a}}^e_{n+1}] \\
\left\langle \frac{\partial \boldsymbol{\xi}^e_{n+1}}{\partial \boldsymbol{X}} \right\rangle &= [\boldsymbol{B}^e][\hat{\boldsymbol{\xi}}^e_{n+1}] \\
\left\langle \boldsymbol{P}_{n+1} \right\rangle &= \begin{bmatrix} P_{11} \\ P_{21} \\ P_{12} \\ P_{22} \end{bmatrix}_{n+1}.
\end{aligned}$$

Where $[\boldsymbol{N}^e]$ was a 2 by 8 array of interpolation function (shape function), $[\hat{\boldsymbol{\xi}}^e_{n+1}]$ was a 8 by 1 array, standed for the tangent vector, $[\boldsymbol{N}^e][\hat{\boldsymbol{\xi}}^e_{n+1}]$ together would yield a 2 by 1 tangent vector. $[\hat{\boldsymbol{a}}^e_{n+1}]$ was a 8 by 1 array, standed for the acceleration vector, $[\boldsymbol{N}^e][\hat{\boldsymbol{a}}^e_{n+1}]$ together would yield a 2 by 1 acceleration vector. $\left\langle \frac{\partial \boldsymbol{\xi}^e_{n+1}}{\partial \boldsymbol{X}} \right\rangle$ was a 4 by 1 array to represent tangent vector gradient, $[\boldsymbol{B}^e]$ was a 4 by 8 array of interpolation function gradient. $\left\langle \boldsymbol{P}_{n+1} \right\rangle$ was a 4 by 1 array of reshaped first Piola-Kirchhoff stress.

Plug the discretization into the Eq.5, we found a matrix form

$$[\boldsymbol{\xi}^e_{n+1}]^T \left(\int_{\Omega_0} [\boldsymbol{N}^e]^T \rho_0 [\boldsymbol{N}^e] dV \right) [\hat{\boldsymbol{a}}^e_{n+1}] + [\boldsymbol{\xi}^e_{n+1}]^T \int_{\Omega_0} [\boldsymbol{B}^e]^T \left\langle \boldsymbol{P}_{n+1} \right\rangle dV = 0.$$

Since the tangent vector was arbitrary, assembling process led to

$$[\boldsymbol{M}][\hat{\boldsymbol{a}}_{n+1}] + [\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})] = \boldsymbol{0}. \tag{6}$$

Where $[\boldsymbol{M}]$ was the global mass matrix, $[\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})]$ was the global stress divergence term as a function of displacement. This was a **group of nonlinear ordinary differential equations** to be solved.

6.2 Time Stepping

Eq.6 was a group of nonlinear ordinary differential equations w.r.t. time. To solve the group of ODEs, we used Newmark method with Newmark parameters $\beta = 1/2, \gamma = 1/4$. Given the parameters, and the known solution at time

step t_n, solution at t_{n+1} could be calculated by

$$\frac{1}{\beta\Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1} + \boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1}) = \boldsymbol{M}[(\hat{\boldsymbol{u}}_n + \hat{\boldsymbol{v}}_n\Delta t)\frac{1}{\beta\Delta t^2} + \frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n] \tag{7}$$

$$\hat{\boldsymbol{a}}_{n+1} = \frac{1}{\beta\Delta t^2}(\hat{\boldsymbol{u}}_{n+1} - \hat{\boldsymbol{u}}_n - \hat{\boldsymbol{v}}_n\Delta t) - \frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n \tag{8}$$

$$\hat{\boldsymbol{v}}_{n+1} = \hat{\boldsymbol{v}}_n + [(1-\gamma)\hat{\boldsymbol{a}}_n + \gamma\hat{\boldsymbol{a}}_{n+1}]\Delta t. \tag{9}$$

Where Δt was the time step, and was assumed to be constant. Hence, it was not difficult to see that Eq.7 was the key equation to be solved to progress from t_n to t_{n+1}.

6.3 Nonlinear solver

Eq.7 derived from last section was a **group of nonlinear algebraic equations**, one could used Newton-Raphson method to solve it. To render more formal Newton-Raphson method, Eq.7 could be written as

$$\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}) = \frac{1}{\beta\Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1} + \boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1}) - \boldsymbol{M}[(\hat{\boldsymbol{u}}_n + \hat{\boldsymbol{v}}_n\Delta t)\frac{1}{\beta\Delta t^2} + \frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n] = \boldsymbol{0}. \tag{10}$$

Then, in each iteration, the Newton-Raphson solver yield

$$\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)} = D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})\backslash(-\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})). \tag{11}$$

Where $\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)}$ was the increment of displacement vector from k iteration to $k+1$ iteration, $D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})$ was the "tangent" matrix, and $\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})$ was the residual at iteration k. One of the most computationally heavy task was to form the **tangent matrix**.

6.4 How to get the Tangent matrix

In the Eq.11, we formed the tangent matrix analytically. The first term in Eq.10 generated part of the tangent matrix, element-wisely

$$\boldsymbol{K}_{n+1_{\text{inertial}}}^{e(k)} = \frac{1}{\beta\Delta t^2}\boldsymbol{M}^e = \frac{1}{\beta\Delta t^2}(\int_{\Omega_0}[\boldsymbol{N}^e]^T\rho_0[\boldsymbol{N}^e]dV). \tag{12}$$

This was the inertial stiffness matrix.

The second term in Eq.10 was from, element-wisely

$$\boldsymbol{R}^e(\hat{\boldsymbol{u}}_{n+1}) = \int_{\Omega_0}[\boldsymbol{B}^e]^T\langle\boldsymbol{P}_{n+1}\rangle dV.$$

Further back, this term was from

$$[\boldsymbol{\xi}_{n+1}^e]^T\int_{\Omega_0}[\boldsymbol{B}^e]^T\langle\boldsymbol{P}_{n+1}\rangle dV = \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}_{n+1}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{P}_{n+1}dV. \tag{13}$$

To get another part of tangent matrix, we had to differentiate RHS of Eq.13

$$\begin{aligned} D[\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{P}dV](\boldsymbol{u},\Delta\boldsymbol{u}) &= D[\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}\boldsymbol{S}dV](\boldsymbol{u},\Delta\boldsymbol{u}) \\ &= \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot D\boldsymbol{F}(\boldsymbol{u},\Delta\boldsymbol{u})\boldsymbol{S}dV + \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}D\boldsymbol{S}(\boldsymbol{u},\Delta\boldsymbol{u})dV. \end{aligned} \tag{14}$$

The first term in RHS of Eq.14 was

$$\begin{aligned}\int_{\Omega_0} \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} \cdot D\boldsymbol{F}(\boldsymbol{u}, \Delta\boldsymbol{u})\boldsymbol{S}dV &= \int_{\Omega_0} \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} \cdot \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{S}dV \\ &= \int_{\Omega_0} \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}\boldsymbol{S}^T \cdot \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}dV \\ &= [\boldsymbol{\xi}^e]^T \int_{\Omega_0} [\boldsymbol{B}^e]^T[\boldsymbol{S}_{mod}][\boldsymbol{B}^e][\Delta\boldsymbol{u}^e]dV.\end{aligned}$$

Where

$$[\boldsymbol{S}_{mod}] = \begin{bmatrix} S_{11}\boldsymbol{I}_2 & S_{21}\boldsymbol{I}_2 \\ S_{12}\boldsymbol{I}_2 & S_{22}\boldsymbol{I}_2 \end{bmatrix}.$$

Where $\boldsymbol{I}_2$ was 2 by 2 identity matrix. Thus

$$\boldsymbol{K}^{e(k)}_{n+1_{\text{geo}}} = \int_{\Omega_0} [\boldsymbol{B}^e]^T[\boldsymbol{S}_{mod}][\boldsymbol{B}^e]dV \tag{15}$$

was the geometric stiffness matrix.

The second term in RHS of Eq.14 could be expanded as

$$\begin{aligned}\int_{\Omega_0} \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} \cdot \boldsymbol{F}D\boldsymbol{S}(\boldsymbol{u}, \Delta\boldsymbol{u})dV &= \int_{\Omega_0} \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} \cdot \boldsymbol{F}\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}}\Delta\boldsymbol{E}dV \\ &= \int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}}\frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T)dV.\end{aligned} \tag{16}$$

Incorporating the constitutive law

$$\boldsymbol{S} = 2\mu\boldsymbol{E} + \lambda tr(\boldsymbol{E})\boldsymbol{I}_2.$$

Where $\boldsymbol{I}_2$ was a second-order identity tensor. And

$$\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}} = 2\mu\boldsymbol{I}_4 + \lambda tr()\boldsymbol{I}_2. \tag{17}$$

Where $\boldsymbol{I}_4$ was a fourth-order identity tensor, $tr()$ was a trace operator on any second-order tensor.

Plug Eq.17 into Eq.16, we could have

$$\begin{aligned}&\int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}}\frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T)dV \\ &= \int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot 2\mu\boldsymbol{I}_4\frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T)dV \\ &+ \int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot \lambda tr(\frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T))\boldsymbol{I}_2dV \\ &= 2\mu\int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot \frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T)dV + \lambda\int_{\Omega_0} tr(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}})tr(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F})\boldsymbol{I}_2dV.\end{aligned} \tag{18}$$

Both terms in RHS of Eq.18 could be written in matrix forms as

$$2\mu\int_{\Omega_0} \frac{1}{2}(\boldsymbol{F}^T\frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}^T\boldsymbol{F}) \cdot \frac{1}{2}(\frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}^T\boldsymbol{F} + \frac{\partial \Delta\boldsymbol{u}}{\partial \boldsymbol{X}}\boldsymbol{F}^T)dV = [\boldsymbol{\xi}^e]^T\int_{\Omega_0} 2\mu[\boldsymbol{B}^e]^T[\boldsymbol{F}_{mod1}]^T[\boldsymbol{F}_{mod1}][\boldsymbol{B}^e][\Delta\boldsymbol{u}^e]dV.$$

Where

$$[\boldsymbol{F}_{mod1}] = \begin{bmatrix} F_{11} & F_{21} & 0 & 0 \\ \frac{1}{2}F_{12} & \frac{1}{2}F_{22} & \frac{1}{2}F_{11} & \frac{1}{2}F_{21} \\ \frac{1}{2}F_{12} & \frac{1}{2}F_{22} & \frac{1}{2}F_{11} & \frac{1}{2}F_{21} \\ 0 & 0 & F_{12} & F_{22} \end{bmatrix}.$$

And

$$\lambda \int_{\Omega_0} tr(\boldsymbol{F}^T \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}) tr(\frac{\partial \Delta \boldsymbol{u}}{\partial \boldsymbol{X}}^T \boldsymbol{F}) \boldsymbol{I}_2 dV = [\boldsymbol{\xi}^e]^T \int_{\Omega_0} \lambda [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod2}]^T [\boldsymbol{F}_{mod2}] [\boldsymbol{B}^e] [\Delta \boldsymbol{u}^e] dV.$$

Where

$$[\boldsymbol{F}_{mod2}] = \begin{bmatrix} F_{11} & F_{21} & F_{12} & F_{22} \end{bmatrix}.$$

Thus,

$$\boldsymbol{K}^{e(k)}_{n+1_{\text{mat}}} = \int_{\Omega_0} 2\mu [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod1}]^T [\boldsymbol{F}_{mod1}] [\boldsymbol{B}^e] dV + \int_{\Omega_0} \lambda [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod2}]^T [\boldsymbol{F}_{mod2}] [\boldsymbol{B}^e] dV \tag{19}$$

was the material stiffness matrix.

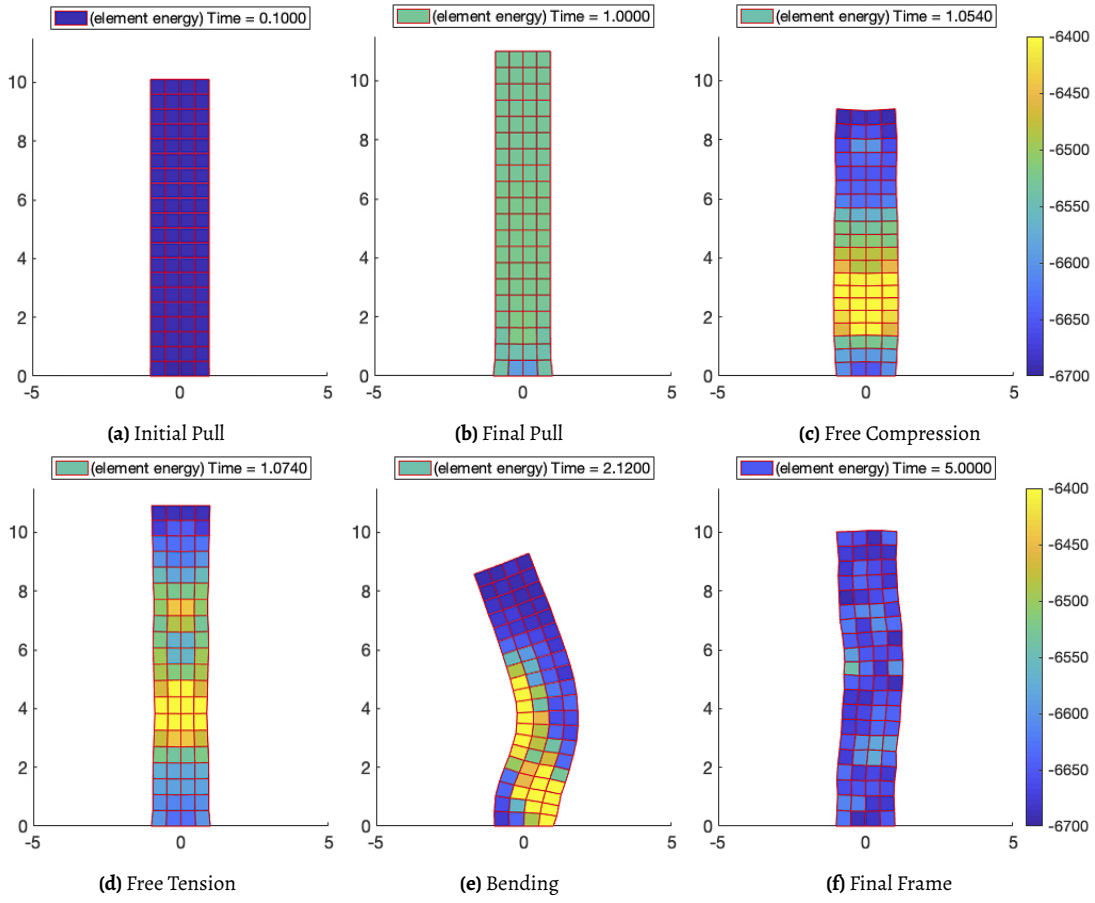
Finally, assembling Eq.12, Eq.15 and Eq.19, the global tangent matrix in Eq.11 could be formed. When the tangent matrix was known, iterative linear system solver could be used to solve Eq.11. Each solution of Eq.11 was the incremental displacement vector in the Newton-Raphson iteration, the solution should added to the current displacement vector. When the residual of current displacement vector was less than some tolerance, Eq.10, hence Eq.7, was successfully solved. To this end, we progress the displacement vector from t_n to $t_n + 1$. Following the displacement, velocity and acceleration vector could be updated using Eq.8 and Eq.9.

6.5 General FE Implementation

Proceeding derivations showed all mathematical and computational details. To solve the real finite element problem, firstly we need a mesh. Mesh was made of node list and element list. Secondly we prepared the output files. Thirdly, the prescribed boundary conditions should be initialized. Then, the program called the Newmark Integrator to start the simulation and incrementally push the time forward. Meanwhile, all displacement, velocity and acceleration vectors, as well as the residual of Newton-Raphson iteration, at each time step was written in the output files. The difference between PULLUP stage and VIBRATION stage was the boundary condition. Finally, when the main program stopped, the solving process was done.

6.6 Correct Results from Matlab

The top nodes of the 2D rubber bar was pulled up to $u = 1m$ at $t = 1.000s$, then released to introduce free vibration. Some representative frame was list in Fig.3. According to the strain contouring in Fig.3, as the top nodes moving up, the strain energy in the bar was increased. When the prescribed boundary condition was released, the bar started vibrating and cyclically generated compression and tension strain within the structure. Analytically, the bar should vibrating symmetrically forever. However, due to accumulation of numerical error, the vibration of bar became unbalanced and unpredictable after around $t = 2s$. Which could be clearly showed in the last two subfigure in the Fig.3.



(a) Initial Pull **(b)** Final Pull **(c)** Free Compression

(d) Free Tension **(e)** Bending **(f)** Final Frame

Figure 3: Deformed mesh colored by strain energy per element