

Assembly-Free Structural Dynamics On CPU and GPU

by

Amir M. Mirzendehtdel

A thesis submitted in partial fulfillment of the requirements for the
degree of

Master of Science

(Mechanical Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

2014

Approved:

Prof. Krishnan Suresh

Department of Mechanical Engineering

University of Wisconsin - Madison

TABLE OF CONTENTES

TABLE OF CONTENTES	ii
ABSTRACT.....	iii
ACKNOWLEDGMENTS	iv
LIST OF TABLES	v
LIST OF FIGURES	vi
NOMENCLATURE	viii
Chapter1: Literature Review	1
Chapter 2: Voxelization	8
Chapter 3: Assembly-Free FEA.....	14
Chapter 4: Deflated Conjugate Gradient.....	19
Chapter 5: Parallelization.....	26
Chapter 6: Adaptive Sub-Domain Refinement.....	29
Chapter 7: Numerical Experiments.....	32
Chapter 8: Conclusion.....	47
Chapter 9: References	49

ABSTRACT

Finite Element Analysis helps designers at the early stages of product design through simulation and behavioral prediction. This thesis is on transient finite element analysis, specifically, structural dynamics, where the behavior of a product due to time-dependent loads is desired. A critical computational challenge in structural dynamics is that it typically requires significant amounts of time and memory.

In the present thesis, a fast time-stepping strategy for the Newmark-beta method is developed; the latter is used extensively in modeling structural dynamics. In particular, we speed up the repeated inversion of the linear system of equations in the Newmark-beta method by implementing and merging five distinct but complementary concepts:

- 1- Voxelization
- 2- Assembly-Free FEA
- 3- Deflated Conjugate Gradient
- 4- Parallelization
- 5- Adaptive Sub-Domain Refinement

The resulting *Assembly-Free Deflated Conjugate Gradient* (AF-DCG) version of the Newmark-beta is well-suited for large-scale problems, and can be easily ported to multi-core architectures. Numerical experiments demonstrate that the proposed method is much faster than the well-known commercial software ANSYS.

ACKNOWLEDGMENTS

Foremost, I would like to sincerely thank my advisor, Prof. Krishnan Suresh, for his invaluable support and insights. I would also like to extend my gratitude to my committee members, Prof. Dan Negrut and Prof. Heidi Ploeg for their help. Further, I thank Prof. Vadim Shapiro for his support at early steps of my graduate studies. Moreover, I acknowledge my peers in Engineering Representation Simulation Laboratory (ERSL), Praveen Yadav, Shiguang Deng, Anirudh Krishnakumar, and Anirban Niyogi. Finally, I would like to express my deep appreciation to my parents for their endless support.

LIST OF TABLES

Table 1: Cantilever Beam: Run-time comparison, 8000 and 25000 voxels ANSYS vs. AF-DCG.....	34
Table 2: Cantilever Beam: Run-time comparison of Element-Based vs. Node-Based	35
Table 3: L-bracket: Convergence of local mesh refinement.....	37
Table 4: Rocker: Run-time comparison of AF-CG vs. AF-DCG	39

LIST OF FIGURES

Figure 1: Arduino MEGA 2560: an example of large-scale thin elastic structure a) CAD model b) real model	2
Figure 2: Arduino MEGA 2560: Triangulated surface and bounding box	8
Figure 3: a) Geometry b) Bounding box	9
Figure 4: Voxelization: background grid	9
Figure 5: Voxelization: find intersection points with boundary	9
Figure 6: Voxelization: create Segments	10
Figure 7: Voxelization: Discard elements with a node outside	10
Figure 8: Voxelization: Improve accuracy by using smaller voxels	11
Figure 9: Arduino MEGA 2560: Voxelized	11
Figure 10: Arduino MEGA 2560: Agglomeration of mesh nodes into about 100 groups	23
Figure 11: OpenMP threads	26
Figure 12: Coarse and Fine Mesh over Ω and $\Omega_l \subset \Omega$	29
Figure 13: Sub-Domain Refinement Algorithm: a) Solve coarse mesh and create fine mesh.	30
Figure 14: Plate with hole a) $t = 1e-7$ (s) b) $t = 1e-6$ c) $t = 1e-5$	31
Figure 15: Cantilever Beam: Displacement field	33
Figure 16: Cantilever Beam: Normalized displacement response	33
Figure 17: Cantilever Beam: Run-time comparison of ANSYS vs. AF-DCG	34
Figure 18: Cantilever Beam: Run-time comparison of Element-Based vs. Node-Based	35
Figure 19: L-Bracket: geometry (dimensions in mm) and loading	36
Figure 20: L-Bracket: Local Mesh Refinement	37
Figure 21: L-Bracket: The normalized maximum vonMises stress a) AF-DCG b) ANSYS	37
Figure 22: Rocker: a) Load condition b) Displacement field	38
Figure 23: Rocker: Run-time comparison of AF-CG vs. AF-DCG	39
Figure 24: Rocker: Sensitivity of run-time to number of Groups	40
Figure 25: Battery holder: Thin structure with small features	41

Figure 26: Battery holder: Run-time on CPU and GPU	42
Figure 27: Battery holder: a) Displacement field b) Stress field	42
Figure 28: Battery holder: Normalized maximum stress over time.....	43
Figure 29: Arduino MEGA 2560: Loading and Boundary conditions	44
Figure 30: Arduino MEGA 2560: Sinusoidal force of $F = 3000(1 + \sin(20\pi t))$	45
Figure 31: Arduino MEGA 2560: Run-time comparison between CPU and GPU.....	45
Figure 32: Arduino MEGA 2560: a) Normalized maximum displacement over time.....	46
Figure 33: Arduino MEGA 2560: a) Displacement field b) Stress field.....	46

NOMENCLATURE

Assembly-Free Structural Dynamics On CPU and GPU

Chapter1: Literature Review

1.1. Introduction:

The main focus of the current study is on flexible-body dynamics simulation, which has become quite popular nowadays, especially where we are more interested in how the response changes over time, rather than the final configuration. The present work focuses on Structural dynamics, however opens door for future studies on many other applications. Contact, fatigue, and crack propagation are few of many examples of such cases.

A dynamics simulation can address a wide range of questions, the answers to which might be of great importance to either prevent unexpected failure or improve the design. For a given geometry with known boundary conditions, based on the application one might ask the following question (and many more):

- When does the maximum deflection or stress happen?
- What is the value of maximum value of deflection and stress?
- How much time must pass until we reach static stability?
- At which frequency is the object oscillating?
- What are the fatigue characteristics?
- How does a crack propagate?

Structural dynamics often needs to be performed on geometrically complex products, such as the one in Figure 1 [27], requiring a large number of finite elements to capture all the features. Although, computers have become faster and parallel computing on both CPU and GPU have made it possible for us to perform analysis on more realistic models, there is a lot left to be desired. For instance, we can easily run out of memory while performing a FEA over a model with a million degrees of freedom, which is not a very large problem.

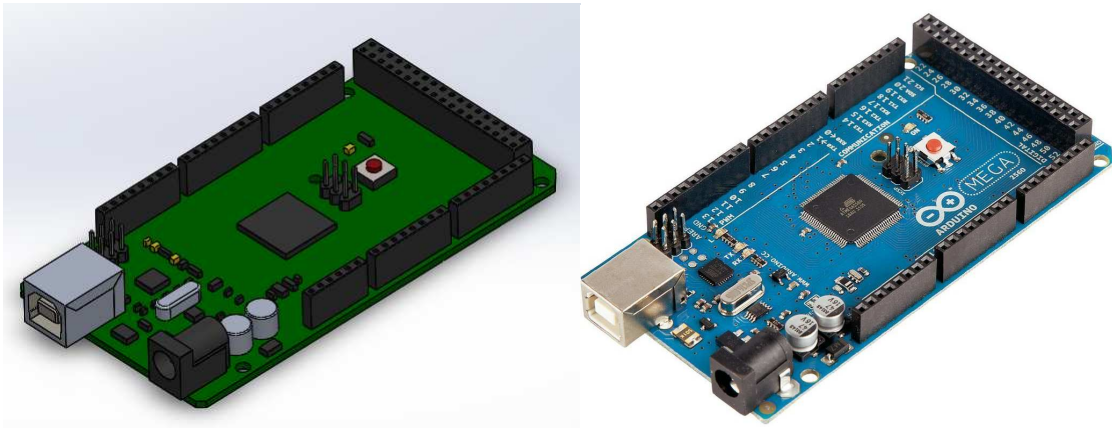


Figure 1: Arduino MEGA 2560: an example of large-scale thin elastic structure
a) CAD model b) real model

1.2. Equation of Motion:

A standard approach to transient analysis of small-displacement elastic bodies (such as the one shown in Figure 1) is to first discretize the geometry via finite elements [2], and then construct a system of second order differential equations in time [1]:

$$M\ddot{u} + C\dot{u} + Ku = f \quad (1.1)$$

Where,

$$\begin{aligned}
 M &: \text{Mass matrix} \\
 C &: \text{Damping matrix} \\
 K &: \text{Stiffness matrix} \\
 f &: \text{External force vector} \\
 u &: \text{Displacement field} \\
 \dot{u} &: \text{Velocity field} \\
 \ddot{u} &: \text{Acceleration field}
 \end{aligned} \tag{1.2}$$

There are various methods that are used to compute Stiffness, Mass and Damping matrices, which are briefly discussed here, more details can be found in [1],[26].

1.3. Stiffness Matrix:

In Elasticity, Stiffness matrix represents both geometry and material of the object under study. In classical FEA, for a system with n nodal degrees of freedom, one needs to populate an n by n matrix by assembling the elemental stiffness values. These values are related to the material's Young's Modulus (E), Poisson's ratio (ν), Shape functions (N), and derivatives of shape functions w.r.t reference nodal coordinates (B).

$$K = \int_V B^T E B dV \tag{1.3}$$

1.4. Mass Matrix:

Mass matrix of an object is a discrete model of the continuous mass distribution, which can be either **Consistent** or **Lumped**. Consistent mass matrix is *usually* more accurate, but expensive to store and use in some algorithms. Lumped mass matrix is diagonal and easier to store, however, the solution may not converge for some special cases. There are numerous types of mass lumping such as particle, HRZ, and optimal.[1] Lumped mass matrix is

preferable when using explicit time integration methods, since taking their inverse is trivial. In the present work, since we are focusing on Newmark-beta (which is an Implicit method), consistent mass matrix is exploited.

$$M = \int_V \rho N^T N dV \quad (1.4)$$

where ρ is density.

1.5. Damping Matrix:

In order to make energy dissipate over time, we can add an artificial damping matrix to the equation of motion. There are two well-known types of damping matrix, 1) Proportional damping (a.k.a Rayleigh damping) and 2) Modal damping.[1] Without loss of generality, we shall assume proportional damping:

$$C = \alpha^d M + \beta^d K \quad (1.5)$$

where α^d and β^d are the damping coefficients.

Note that the first term of equation (1.5), $\alpha^d M$ damps lower or dominant modes, while the second term $\beta^d K$, dissipates the higher modes (which sometimes are actual noise). Perhaps the most important advantage of proportional damping matrix is that there is no need to store a new matrix and we can simply modify (1.1) to include α^d and β^d .

1.6. Direct Time Integration:

Equation (1.1) must be solved through time integration. There are two general approaches here:

1) Explicit

2) Implicit

In explicit methods, the solution at time t is used explicitly to obtain the solution at $t + \Delta t$. This, as it turns out, entails the inversion of the mass matrix M [1]. Since M can be made diagonal via lumping, its inversion is trivial, leading to rapid time-stepping. Explicit methods are easy to implement, however, they are unstable for large time steps Δt . Popular explicit methods are central difference method and fourth-order Runge-Kutta method, that exhibits fourth-order accuracy.[5]

On the other hand, implicit methods such as the Newmark-beta method are harder to implement, yet are unconditionally stable. They require the ‘inversion’ of an effective stiffness matrix[1], which is a computationally demanding task. Next, the Newmark-beta method is explained in more details, since it is of our interest in this study.

1.7. Newmark-beta Method

In 1959, Newmark formulated this second-order accuracy method [4] by writing Taylor expansion of displacement and velocity and then adding implicitness to the resulting equation by introducing two numerical parameters γ and β as shown in (1.6) and (1.7).

$$u_{t+\Delta t} = u_t + \Delta t \dot{u}_t + \frac{(\Delta t)^2}{2} \left[(1 - 2\beta) \ddot{u}_t + 2\beta \ddot{u}_{t+\Delta t} \right] \quad (1.6)$$

$$\dot{u}_{t+\Delta t} = \dot{u}_t + \Delta t \left[(1-\gamma)\ddot{u}_t + \gamma\ddot{u}_{t+\Delta t} \right] \quad (1.7)$$

A convenient choice for the unknown is displacement, therefore we substitute (1.6) with (1.8), in order to find acceleration as function of displacement.

$$\ddot{u}_{t+\Delta t} = \frac{1}{\beta(\Delta t)^2} (u_{t+\Delta t} - u_t) - \frac{1}{\beta\Delta t} \dot{u}_t - \frac{1-2\beta}{2\beta} \ddot{u}_t \quad (1.8)$$

By substituting (1.7) and (1.8) into (1.1), we obtain a simple linear system:

$$K^{eff} u_{t+\Delta t} = f_{t+\Delta t}^{eff} \quad (1.9)$$

where K^{eff} and f^{eff} are effective stiffness matrix and effective force vector, respectively, where: [5]

$$K^{eff} = K + \frac{\gamma}{\beta\Delta t} C + \frac{1}{\beta(\Delta t)^2} M \quad (1.10)$$

$$\begin{aligned} f_{t+\Delta t}^{eff} &= f_{t+\Delta t}^{ext} + f^C + f^M \\ f^C &= C \left(\frac{\gamma}{\beta\Delta t} u_t + \frac{\gamma-\beta}{\beta} \dot{u}_t + \frac{\Delta t(\gamma-2\beta)}{2\beta} \ddot{u}_t \right) \\ f^M &= M \left(\frac{1}{\beta(\Delta t)^2} u_t + \frac{1}{\beta\Delta t} \dot{u}_t + \frac{1-2\beta}{2\beta} \ddot{u}_t \right) \end{aligned} \quad (1.11)$$

The effective stiffness matrix of equation (1.10) is symmetric positive-definite, since both K and M are symmetric positive definite and all of the coefficient are positive too. In linear elasticity, the stiffness and mass matrices remain constant throughout the analysis, and the effective stiffness matrix needs to be computed only once. However, the effective force

vector must be updated at each time step, since it depends on the displacement, velocity, and acceleration fields. In large-deformation models and in Elasto-Plasticity, the effective stiffness matrix can change over time. In this thesis, we assume that the stiffness and mass matrices remain unchanged during simulation.

Chapter 2: Voxelization

2.1. Introduction

The proposed method of solving Equation (1.9) through assembly-free deflated conjugate-gradient; this method is applicable to any finite-element discretization. However, we consider a simple discretization, where the geometry is approximated via uniform hexahedral elements or ‘voxels’. The voxel-approach has gained significant popularity recently due to its robustness and low memory foot-print[15]. One simple choice for representing the boundary is by triangulating the surface of the object [17].

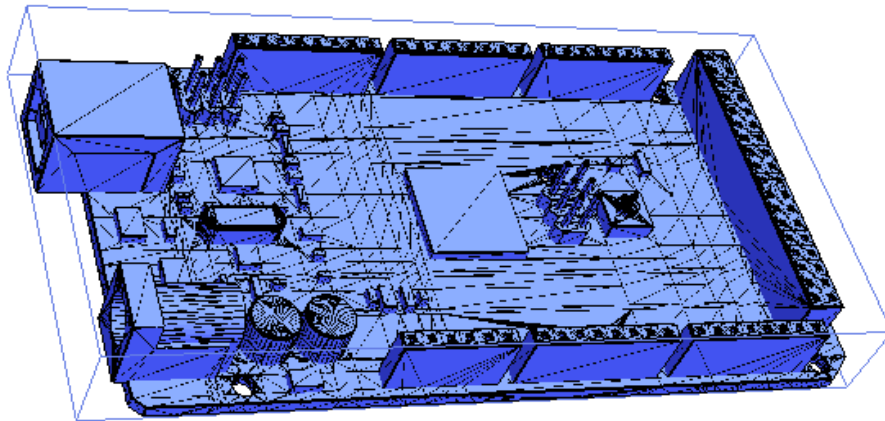


Figure 2: Arduino MEGA 2560: Triangulated surface and bounding box

2.2. Voxelization

For simplicity, let us discuss the 2D case; extension to 3D is then straight forward.

Consider the simple 2D sketch of Figure 3 and its approximate bounding box.

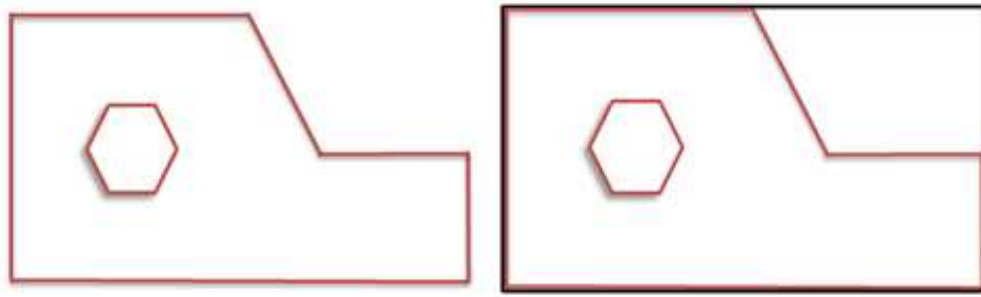


Figure 3: a) Geometry b) Bounding box

The first step after finding the bounding box is to create a background grid, knowing the element dimensions h_x and h_y .

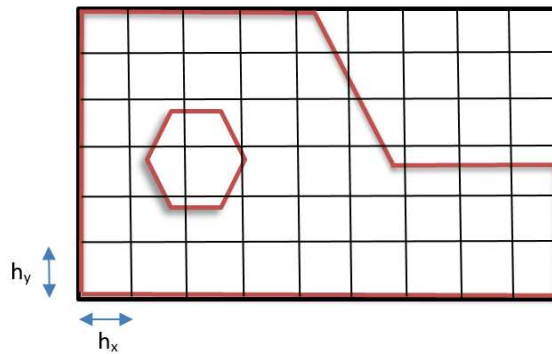


Figure 4: Voxelization: background grid

Next, we walk in x direction, cast rays in y direction and find the intersection points with the boundary.

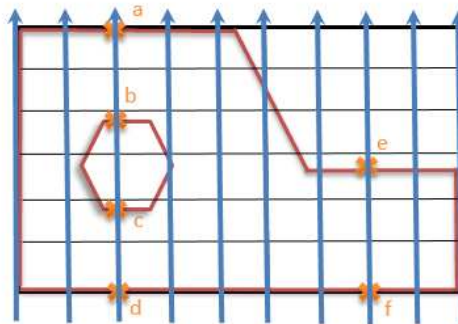


Figure 5: Voxelization: find intersection points with boundary

After gathering all intersection points for each ray, we sort the nodes w.r.t their y coordinates. Observe that for a valid sketch, there are always an even number of intersection points. Then we create line segments starting from the first intersection point and the second one. We continue this for all intersection points, keeping in mind that each point must belong to only one line segment.

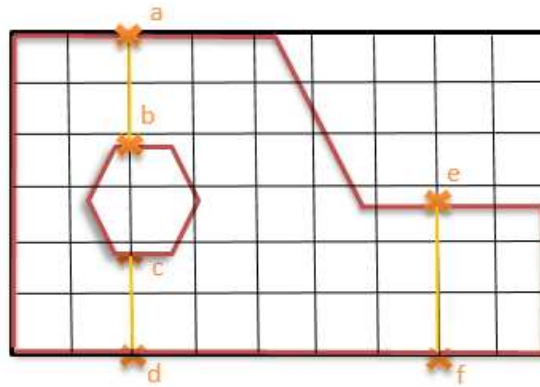


Figure 6: Voxelization: create Segments

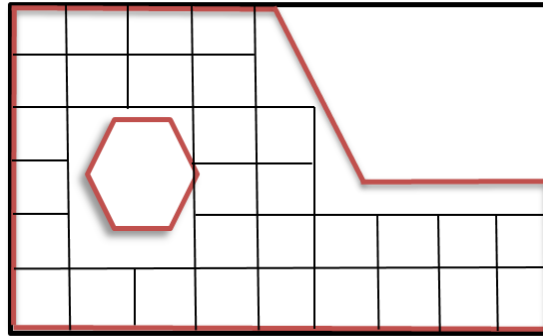


Figure 7: Voxelization: Discard elements with a node outside

We can improve the accuracy by reducing the voxel size and increasing the number of finite elements.

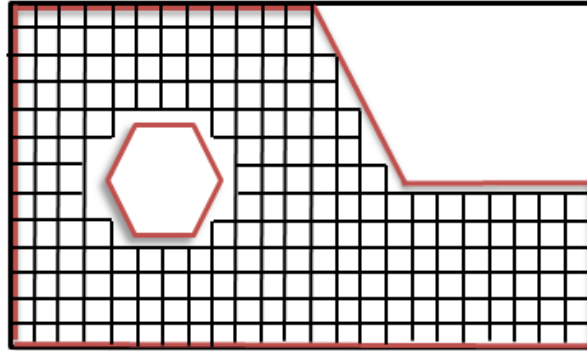


Figure 8: Voxelization: Improve accuracy by using smaller voxels

The only differences in 3D are 1) boundary consists of triangles and 2) we move in x-y plane and cast rays in z direction.

The voxelization of the geometry in Figure 2 is illustrated in Figure 9. It has about 300,000 voxels. Fortunately, even such a large-size problem is easily handled via the proposed method.

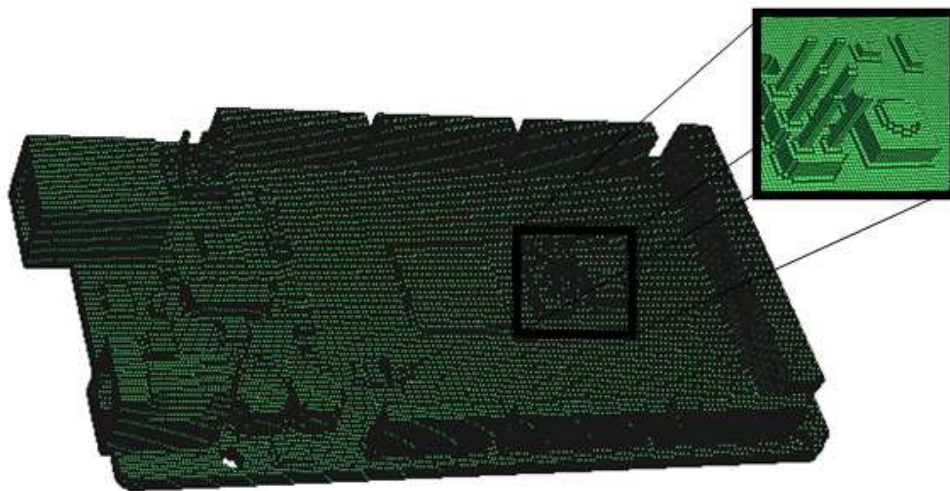


Figure 9: Arduino MEGA 2560: Voxelized

The most significant benefits of voxelization are: (1) it is robust in that it rarely fails (unlike traditional meshing), (2) the mesh storage is compact, (3) computational cost of voxelization is usually negligible and is relatively insensitive to geometric complexity, and (4) it promotes assembly-free analysis.

Typically, the downside of voxelization is that the stresses tend to be less accurate. We mitigate this through two strategies described below.

2.3. Shape Functions

Given a voxelization, one can choose a variety of hexahedral finite element shape functions. The simplest is the set of tri-linear shape functions described in [26], where each node-based shape function is of the form:

$$N_i = 0.125(1 + \xi_i \xi)(1 + \eta_i \eta)(1 + \zeta_i \zeta); i = 1, 2, \dots, 8 \quad (2.1)$$

However, the resulting 8-noded elements are ‘stiff’, and convergence is slow. One could use 20-node or 27-node elements, but this increases the memory requirements significantly.

Instead, we use the *Wilson* element endowed with three additional bubble-functions of the form of [23],[22]:

$$\begin{aligned} M_1(\xi, \eta, \zeta) &= (1 - \xi^2) \\ M_2(\xi, \eta, \zeta) &= (1 - \eta^2) \\ M_3(\xi, \eta, \zeta) &= (1 - \zeta^2) \end{aligned} \quad (2.2)$$

The resulting element stiffness matrix are of the form:

$$\bar{K}_e = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \quad (2.3)$$

where

$$\begin{aligned} K_{11} &= \int \nabla N^T D \nabla N d\Omega \\ K_{12} &= \int \nabla N^T D \nabla M d\Omega \\ K_{21} &= \int \nabla M^T D \nabla N d\Omega \\ K_{22} &= \int \nabla M^T D \nabla M d\Omega \end{aligned} \quad (2.4)$$

One can condense out the bubble degrees of freedom, resulting again in a reduced 24 degrees of freedom element stiffness matrix [22]:

$$K_e = K_{11} - K_{12}(K_{22} \setminus K_{21}) \quad (2.5)$$

This significantly improves the stress predictions without penalizing the computation since equation (2.5) needs to be carried out once. Similar condensation can be carried out for the mass element matrix. Moreover, an adaptive subdomain refinement is developed as a second strategy which will be discussed in chapter 6.

Chapter 3: Assembly-Free FEA

3.1. Introduction

In classic finite element analysis, the element matrices are typically assembled into global matrices K and M , whose size and structure will depend on distribution of the voxels.

In the present work, the explicit construction of these matrices is avoided through an *Assembly-free* (a.k.a. *matrix-free*) approach [2][12], where neither K nor M are assembled/stored. The basic idea behind Assembly-Free FEA, as shown by Equation (3.1), is to perform **S**parse-**M**atrix **V**ector **M**ultiplication (SpMV) element by element and assemble the resulting vector, instead of assembling the matrix and then multiply by the vector.

$$Ku = \left(\prod_{assemble} K_e \right) u \equiv \prod_{assemble} (K_e u_e) \quad (3.1)$$

Here, we explain two techniques to perform Assembly-Free method, introduced by Yadav and Suresh [24]:

- 1) Element Connectivity Based
- 2) Node Connectivity Based

3.2. Element Connectivity Based

This approach is more intuitive, in that we walk through each node and find the elements it belongs to. Then for each element, local number of the node within the element is found (can be from 1 to 8 for a voxel), so that the corresponding row in the stiffness matrix can be extracted. Once these values are known, we can perform the multiplication with the nodal DOFs of all nodes of the element.

Algorithm: Element Connectivity-Based Assembly-Free

1. Compute element stiffness and mass matrices
2. For $N = 1, 2, \dots, \text{NumNodes}$ Do:
3. Find elements connected to node
4. For $elem = 1, 2, \dots, numNeighbors$ Do
5. Find place of node in element numbering
6. Find the corresponding set of values in element Stiffness (or Mass) matrix
7. Find the nodal values of the element
8. Perform the dot product
9. End-Do neighboring elements
10. End-Do nodes

3.3. Node Connectivity Based

Node connectivity based is slightly more complicated. The idea is to find and store the following information for each node at the pre-processing stage:

1. Relative position (*signature*) w.r.t neighbors
2. Neighbors

For any given node, each of its 8 neighboring elements can either exist or not, therefore there are only $2^8 = 256$ distinct relative configurations for a node to be placed w.r.t its neighbors, but since we assume that each node belongs to at least one element, we subtract the condition where none of the elements exist. Hence, there are 255 of such configurations for which we can find the set of rows that make contribution to the target node. For each node, we find which configuration it is at, and assign the relevant set of nodal rows to it. Next, we find

what the actual neighbors are and store them for each node. Each node can have up to 27 neighboring nodes.

Algorithm: Node Connectivity-Based Assembly-Free;

1. Compute element stiffness and mass matrices
2. Find and store all relative configurations
3. Assign a configuration to each node
4. Find and store the neighboring nodes
5. For $N = 1, 2, \dots, \text{NumNodes}$ Do:
6. Find its signature
7. Find the neighboring nodes
8. Find the nodal values of vector
9. Perform dot product
10. End-Do nodes

3.4. Assembly-Free Newmark-beta Formulation

Since our final goal is to solve Equation (1.9), we need to carry out the Effective Stiffness matrix (K^{eff}), and since our approach is assembly-free, we need to find K^{eff} per element (K_e^{eff}). This is a fairly straight forward procedure, as shown.

Observe that in Equation (1.9), K^{eff} is a linear combination of K and M , therefore the element effective stiffness matrix K_e^{eff} can be computed as follows:

$$K_e^{eff} = \eta K_e + \varsigma M_e \quad (3.2)$$

where:

$$\eta = 1 + \frac{\gamma\beta^d}{\beta h} \quad (3.3)$$

$$\varsigma = \frac{1}{\beta\Delta t^2} + \frac{\gamma\alpha^d}{\beta\Delta t} \quad (3.4)$$

The advantages of a matrix-free analysis are: (1) memory requirements are obviously reduced, and therefore fine resolution transient analysis can be carried out, (2) memory reduction indirectly translates into decreased computational speed [16], and (3) matrix-free multiplication is well suited for parallelization on multi-core architectures [24].

Considering Equations (1.5) and (1.11), one can see that f^{eff} can be expressed in terms of $Ku, K\dot{u}, K\ddot{u}, Mu, M\dot{u}$ and $M\ddot{u}$ as follows:

$$\begin{aligned} f_{t+\Delta t}^{eff} = & f_t^{ext} + \frac{\gamma\beta^d}{\beta\Delta t} Ku_t + \\ & \left(\frac{\gamma\beta^d}{\beta} - \beta^d \right) K\dot{u}_t + \left(\frac{\gamma\beta^d\Delta t}{2\beta} - \beta^d\Delta t \right) K\ddot{u}_t + \\ & \left(\frac{1}{\beta\Delta t^2} + \frac{\gamma\alpha^d}{\beta\Delta t} \right) Mu_t + \left(\frac{1}{\beta\Delta t} + \frac{\gamma\alpha^d}{\beta} - \alpha^d \right) M\dot{u}_t + \\ & \left(\frac{\gamma\alpha^d\Delta t + 1}{2\beta} - \alpha^d\Delta t - 1 \right) M\ddot{u}_t \end{aligned} \quad (3.5)$$

Thus, to compute the effective force at each time-step, one must carry out several SpMV; these can be carried out in an assembly-free manner.

For large-scale problems, Node-based approach is faster than Element-based connectivity. The reason is that in the former, we spend more time in pre-processing stage, hence the stiffness values are known beforehand; while in the latter, the corresponding stiffness rows

must be found every time. Clearly, nodal based is advantageous when multiple multiplications are needed, e.g. transient analysis. (See Figure 18 and Table 2.)

Chapter 4: Deflated Conjugate Gradient

4.1. Introduction

Computationally, the most intensive task in the Newmark-beta method is solving the linear system in equation (2.4). Generally, there are two types of solvers used to solve such system of equations:

- 1) Direct solvers
- 2) Iterative solvers

4.2. Direct and Iterative Solvers: Tradeoff

For sufficiently small finite element problems, i.e. if the stiffness matrix has less than, say, 500,000 degrees of freedom, sparse direct solvers are superior. Direct solvers are robust, and rely on factoring the matrix, for example, into a Cholesky decomposition:

$$K^{eff} = LL^T \quad (4.1)$$

where L is a triangular matrix.

Equation (1.9) turns into (4.2),

$$u_{t+\Delta t} = L^{-T}(L^{-1})f_{t+\Delta t}^{eff} \quad (4.2)$$

Since L is triangular, taking the inversion is trivial. In transient analysis, direct methods are particularly favorable since the factorization needs to be carried just once.

On the other hand, due to the explicit factorization, direct solvers are memory intensive. For instance, to quote from the ANSYS manual [31], “[sparse direct solver] *is the most robust*

solver in ANSYS, but it is also compute- and I/O-intensive". Specifically, for a matrix with *one million degrees of freedom* [4]:

- *Approximately 1 GB of memory is needed for assembly.*
- *However, 10 to 20 GB additional memory is needed for factorization.*

Since memory-access is often the bottle-neck in modern computer architecture [10], this leads to increased clock time. In other words, *reducing memory usage is critical for large-scale problems*.

As mentioned before, the alternative is using an Iterative solver. Iterative solvers are slower than direct solvers. They do not factorize the stiffness matrix, but compute the solution iteratively.

When the stiffness matrix is symmetric and positive-definite, the most common iterative solver is conjugate gradient [21].

Consider the following quadratic to be of order N :

$$f(x) = \frac{1}{2} x^T Q x + x^T b + c \quad (4.3)$$

where Q is a symmetric positive-definite matrix. If we find a set of mutually Q -conjugate vectors, $\{s_i\}_{i=1,2,\dots,N}$, then by starting from any initial point x^0 and performing line-search along direction $d^i = s_i$, the exact minimum of the function will be found.

There are numerous ways to find conjugate directions. Powell, Steepest Descent Direction, Conjugate Gradient, Newton-Raphson, and Quasi-Newton (BFGS) are just some of many. It

is also worth mentioning that Powell's method is a zeroth-order method, meaning that only the function itself needs to be evaluated at each point. Steepest Descent Direction, Conjugate Gradient, and Quasi-Newton are considered first-order methods, since we also need to compute their gradient $G \equiv \nabla f$. Finally, Newton-Raphson is a second-order method, where the Hessian $H \equiv \nabla G$ needs to be carried out, as well.

4.3. Conjugate Gradient Method

The Conjugate Gradient method is probably the most popular first-order method today. The idea is to start from an initial guess and perform the first search along the steepest descent direction. Every future direction from now on is updated such that it would be conjugate to all previous search-directions. The algorithm is as follows:

Algorithm: **Conjugate Gradient (CG);** solve $Kd = f$

1. Find initial search direction at x^0 ; $d^0 = \nabla f(x^0)$, $i = 0$
2. Do While ($\|\nabla f(x^i)\| < Tol$)
3. $i = i + 1$
4. Update $x^i = x^{i-1} + d^{i-1}$
5. Enforce conjugacy; $\beta = -\frac{(d^{i-1})^T K \nabla f(x^i)}{(d^{i-1})^T K d^{i-1}}$
6. Update search-direction; $d^i = \nabla f(x^i) + \beta d^{i-1}$
7. End-Do

The iterative solver must be accelerated either through an efficient pre-conditioner and/or through multi-grid/deflation techniques. Herein, we consider a particular deflation technique proposed in [9].

Equally important is an efficient implementation of SpMV, which has drawn considerable attention from several researchers. For example, see [13] for an implementation of SpMV on graphics-programmable units (GPUs). In this paper, we consider *an assembly-free implementation of SpMV*.

In summary, one can conclude that, for large-scale transient analysis:

- Iterative solvers scale better than direct methods.
- Pre-conditioning and/or multi-grid/deflation is important in iterative techniques.
- Efficient SpMV and reducing memory foot-print will reduce the computational cost per iteration.
- Exploiting multi-core architecture shows promise, but hinges on building parallelization-friendly algorithms.

4.4. Deflated Conjugate Gradient method

Deflation is a popular method for accelerating iterative methods such as conjugate gradient. The concept behind deflation [20] is to construct a matrix W , referred to as the *deflation space*, whose columns ‘approximately’ span the low eigen-vectors of the (effective) stiffness matrix.

Since computing the eigen-vectors is obviously expensive, Adams and others [9], [11] suggested a simple *agglomeration* technique, where finite element nodes are collected into

small number of groups. For example, Figure 10 illustrates agglomeration of the finite element nodes into 100 groups.

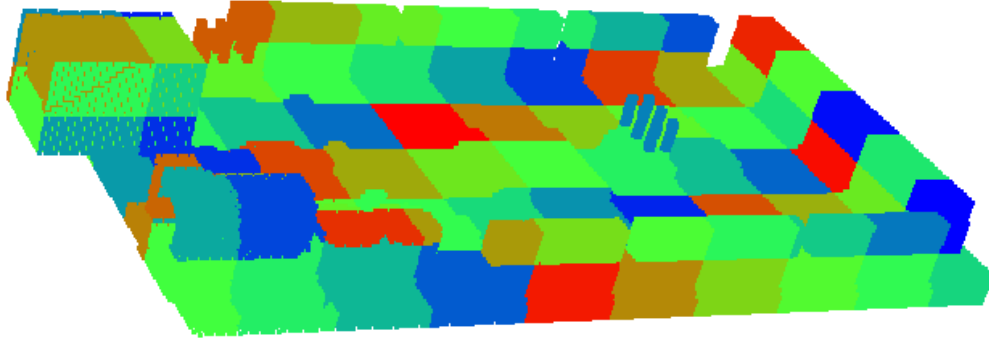


Figure 10: Arduino MEGA 2560: Agglomeration of mesh nodes into about 100 groups

As a step towards constructing the W matrix, nodes within each group are collectively treated as a rigid body. The motivation is that these agglomerated rigid body modes mimic the low-order eigen-modes. Then displacement of each node within a group is expressed as:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & z & -y \\ 0 & 1 & 0 & -z & 0 & x \\ 0 & 0 & 1 & y & -x & 0 \end{pmatrix} \lambda_g \quad (4.4)$$

where

$$\lambda_g = \{u_0, v_0, w_0, \theta_x, \theta_y, \theta_z\}^T \quad (4.5)$$

are the six unknown rigid body motions associated with the group, and (x, y, z) are the relative coordinates of the node with respect to the geometric center of the group. Observe that Equation (4.4) is essentially a restriction operation similar to that of multi-grid [14],[7].

Once the mapping in equation (4.4) is constructed for all the nodes, they can be ‘assembled’ to result in a deflation matrix W :

$$d = W\lambda \quad (4.6)$$

where d is the $3N$ ($N : Node$) degrees of freedom, λ is the $6G$ ($G : Group$) degrees of freedom associated with the groups. One can now exploit the W matrix to create the *Deflated Conjugate Gradient* (DCG) algorithm described below:[24]

Algorithm: Deflated CG (DCG); solve $Kd = f$

1. Construct the deflation space W
2. Choose d_0 where $W^T r_0 = 0$ & $r_0 = f - Kd_0$
3. Solve $W^T K W \mu_0 = W^T K r_0$; $p_0 = r_0 - W \mu_0$
4. For $j = 1, 2, \dots, m$, do:
 5. $\alpha_{j-1} = \frac{r_{j-1}^T r_{j-1}}{p_{j-1}^T K p_{j-1}}$
 6. $d_j = d_{j-1} + \alpha_{j-1} p_{j-1}$
 7. $r_j = r_{j-1} - \alpha_{j-1} K p_{j-1}$
 8. $\beta_{j-1} = \frac{r_j^T r_j}{r_{j-1}^T r_{j-1}}$
 9. Solve $W^T K W \mu_j = W^T K r_j$ for μ
 10. $p_j = \beta_{j-1} p_{j-1} + r_j - W \mu_j$
11. End-Do

When $N \gg G$, i.e., when the number of mesh nodes far exceeds the number of groups:

- The primary computation is the SpMV, e.g. Kx in steps 5 and 9.

- Additional computations include the restriction operation $W^T x$ in step 9, the prolongation $W\mu$ in step 10, and the solution of the linear system $(W^T K W)\mu = y$ in step 9.

The one-time coarse matrix $W^T K W$ construction in step 3 can be viewed as a series of SpMV, followed by a series of restriction operations. Observe that the deflation matrix W is also sparse.

Chapter 5: Parallelization

5.1. Parallelization On CPU

Parallel computing is solving set of relatively small problems concurrently, as opposed to solving them sequentially. This evolutionary idea allows us to analyze larger and more complicated problems faster, and since our goal is solving large-scale problems more efficiently, parallelization plays a pivotal role in our method.

Parallelization can be reached on both Central Processing Unit (CPU) and Graphical Processing Unit (GPU). Programming on GPU takes more time and experience, but it is worthwhile due to its high efficiency. The reason is that CPU is designed to perform many different tasks, while GPU has one particular purpose which is designed optimally for.

5.2. Parallelization On CPU

Parallelization on CPU is implemented using an API called **Open Multi-Processing** (OpenMP), which consists of compilers, libraries, and environmental variables. The basic idea is to run a master thread, which then launches number of slave threads to perform tasks simultaneously. Figure 11 Shows how the thread launching works in OpenMP [29],

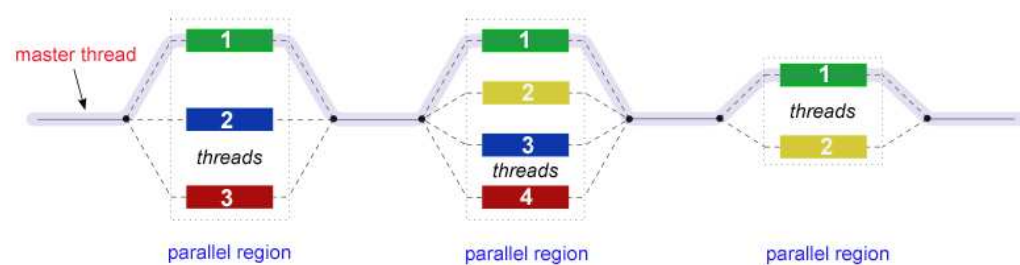


Figure 11: OpenMP threads

As mentioned before, we can exploit OpenMP to easily enhance the overall performance by parallelizing vector operations, e.g. SpMV.

5.3. Parallelization On GPU

Parallel computing on GPU is extremely efficient for large-scale problems that can be divided into many simple smaller ones. Due to its high number of cores, a great number of these simpler tasks can be solved concurrently.

CUDA is a very popular platform for GPU computing introduced by NVidia. CUDA is considered a high-level language, which can be based on C, C++, FORTRAN, etc. [33]

The main idea is to launch a lot of cores, usually referred to as *threads*, to complete a set of simple tasks. A group of threads with the same shared memory create a *Block*, which can be arranged as a three dimensional array of threads (blockDim.x, blockDim.y, and blockDim.z). *Grid* is basically a group of blocks, which can be organized as an array of blocks in two dimensions (gridDim.x and gridDim.y). All blocks of a grid must be of the same dimensions and they can have 512 (or 1024 on newer GPUs) threads inside them. Threads of a grid execute the same *Kernel* function to complete an assignment. A kernel function must have a specified dimension at the launching time, which cannot be changed after execution. Also, it is worth mentioning that each block consists of units called *Warps*, which are groups of threads launched simultaneously. Each warp has 32 threads and can also be known as unit of thread scheduling in SMs.[27]

Some of the common challenges in GPU programming are *latency* and *race condition*.

Latency is the time spent on accessing data from either the main memory or GPU cache.

Race condition happens when multiple threads try to access the same location of memory at the same time [27].

Assembly-Free FEA is implemented on GPU as follows. To begin with, we have to decide whether we want to assign threads to nodes or elements. Observe that both algorithms for Element-Based and Node-Based Connectivity are performing loops over nodes. The reason is to avoid race condition. As elements share nodes, if we assign a thread to an element two threads might want access to the shared node at the same time. While by assigning threads to nodes, each thread is writing data to its own nodal DOF, hence no race condition occurs.

Most of the parallelization here is done for SpMV, e.g. $Ku = f$.

Chapter 6: Adaptive Sub-Domain Refinement

6.1. Introduction

In this section, we introduce an Adaptive Sub-Domain Refinement algorithm to improve the accuracy of stress prediction. The main reason is that one can question the validity of the solution, since the mesh is non-conforming.

Consider domain Ω to be discretized by coarse mesh and we are interested in improving the results of the region Ω_l where stress concentration has occurred. Therefore, by generating a local fine mesh over Ω_l and carefully imposing the right boundary conditions, we can obtain a better prediction for stress over the refined domain. Further, one can improve the solution over Ω by adding more steps as discussed in [7].

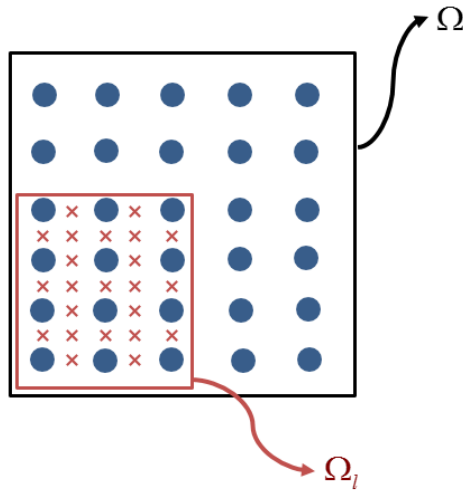


Figure 12: Coarse and Fine Mesh over Ω and $\Omega_l \subset \Omega$

6.2. Sub-Domain Refinement Algorithm

Sub-modeling [1] (or local mesh refinement) is a classic idea in finite element analysis where after a global problem is solved, one creates a higher-resolution mesh is created around regions of stress concentrations.

Assume that we have already solved equation (1.9) over Ω (coarse mesh) and we want to improve the results over some known region $\Omega_l \subset \Omega$. The method consists of the following steps:[6],[18]

- 1) Interpolate the boundary values of Ω_l from solution of Ω . (Dirichlet B.C.)
- 2) Solve equation (1.9) over Ω_l
- 3) Compute Stress values over Ω_l
- 4) Smooth the results

Here we can either report the obtained results, or push the improved solution back to the coarse mesh.

Figure 13 illustrates the algorithm more clearly,

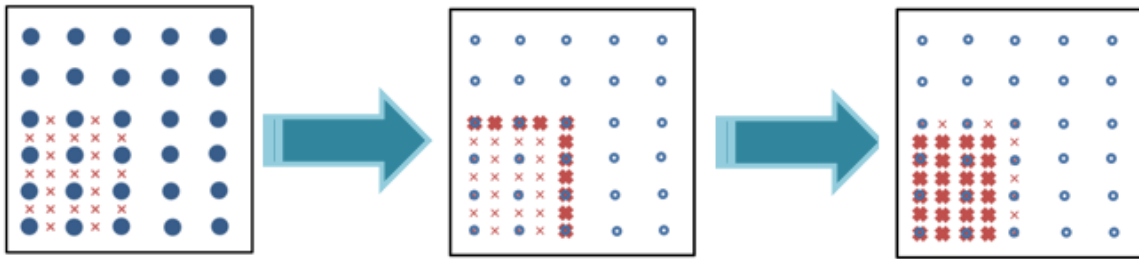


Figure 13: Sub-Domain Refinement Algorithm: a) Solve coarse mesh and create fine mesh. b) Interpolate boundary values of fine mesh. c) Solve fine mesh

6.3. Adaptivity

In most commercial packages, the user needs to know the location(s) of stress concentration beforehand to apply refinement. In order to have a robust method for transient analysis, the algorithm must be able to update the refinement regions as time passes and loading conditions change. To this end, at each time step (or once in a number of time steps) we find the critical locations, create the local fine mesh, and solve fine problems.

Ideally, one should generate a conforming mesh in the refined regions (which is a work of near future), but in this thesis we voxelize sub-domains too. Figure 14 shows how the stress concentration domain changes through time for a plate with hole.

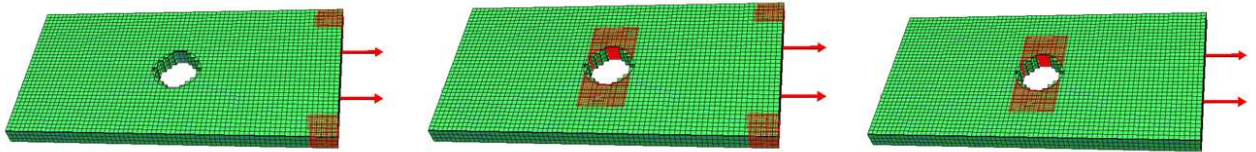


Figure 14: Plate with hole a) $t = 1e-7$ (s) b) $t = 1e-6$ c) $t = 1e-5$

Chapter 7: Numerical Experiments

7.1. Introduction

In this Section, we present results from numerical experiments based on the proposed algorithm. All experiments were conducted on a Windows 7 64-bit machine with the following hardware: Intel Core i7 CPU running at 3.4GHz with 8 GB of memory, and a graphics card of GeForce GTX-760; parallelization on CPU and GPU were implemented through OpenMP and CUDA, respectively.

For all experiments, the Newmark coefficients were:

$$\begin{aligned}\gamma &= 0.5 \\ \beta &= 0.25\end{aligned}\tag{6.1}$$

The material properties were those of steel:

$$\begin{aligned}E &= 2.1e11N / m^2 \\ \nu &= 0.28 \\ \rho &= 7700kg / m^3\end{aligned}\tag{6.2}$$

7.2. Speed

In the first experiment, we compare the proposed assembly-free deflated conjugate gradient (AF-DCG) against the popular commercial finite element software, ANSYS[31]. The geometry is a steel cantilever beam of dimension $0.5 \times 0.02 \times 0.05$ (meters). A tip-force of one Newton is applied at $t = 0$ as illustrated in Figure 15, and maintained thereafter. In ANSYS, the ‘Brick 8 node 185’ element was used, while the AF-DCG relies on the Wilson element described earlier.

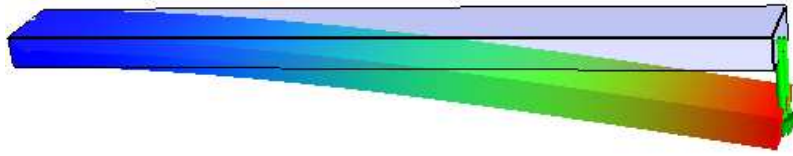


Figure 15: Cantilever Beam: Displacement field

The damping coefficients are $\alpha^d = 0$ and $\beta^d = 5E - 4$. The analysis time is 0.2 seconds, with $\Delta t = 0.0005$ s.

In both implementations, with 8000 elements, the maximum deflection was reached at around 0.01 seconds, where:

$$\begin{aligned} \delta_{\max} &= 5.88E - 6 \text{ (m)} & (\text{ANSYS}) \\ \delta_{\max} &= 5.92E - 6 \text{ (m)} & (\text{AF-DCG}) \end{aligned} \quad (6.3)$$

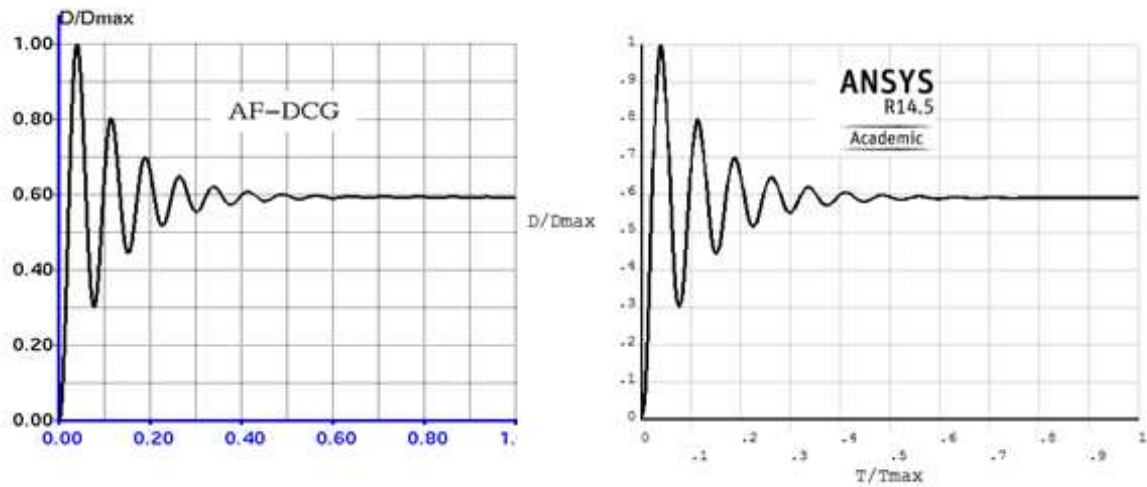


Figure 16: Cantilever Beam: Normalized displacement response

a) AF-DCG b) ANSYS

The slight difference can be attributed to the difference in the two shape functions used.

Figure 16 illustrates the normalized tip displacement in AF-DCG and ANSYS classic. Both methods converged to the static deflection as expected.

To compare the computational costs, the geometry was discretized using two different mesh sizes: 8000 elements and 25000 elements. With each mesh size, three different solvers were considered (1) ANSYS-direct, (2) ANSYS-pre-conditioned conjugate gradient (PCG), and (3) proposed AF-DCG. Figure 17 compares the computational times. As one can observe, AF-DCG is about seven times faster than ANSYS for the smaller mesh size, and about fifteen times faster for the larger mesh size.

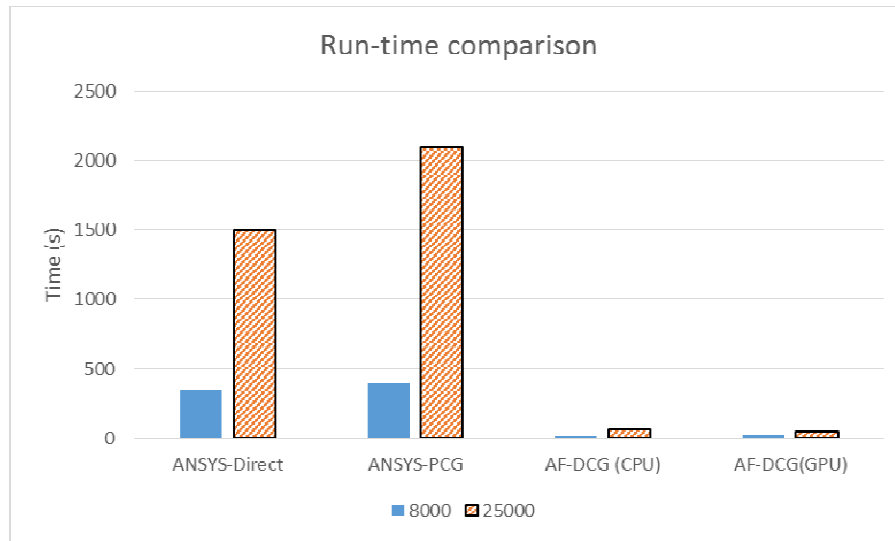


Figure 17: Cantilever Beam: Run-time comparison of ANSYS vs. AF-DCG

	<i>ANSYS Direct</i>	<i>ANSYS PCG</i>	<i>AF-DCG (CPU)</i>	<i>AF-DCG (GPU)</i>
<i>8000 voxels run-time(s)</i>	350	400	22	24
<i>25000 voxels run-time(s)</i>	1500	2100	66	53

**Table 1: Cantilever Beam: Run-time comparison, 8000 and 25000 voxels
ANSYS vs. AF-DCG**

The next important comparison can be made between Node-Based and Element-Based connectivity methods discussed in chapter 3. Figure 18 and Table 2 summarize the results for the cantilever beam.

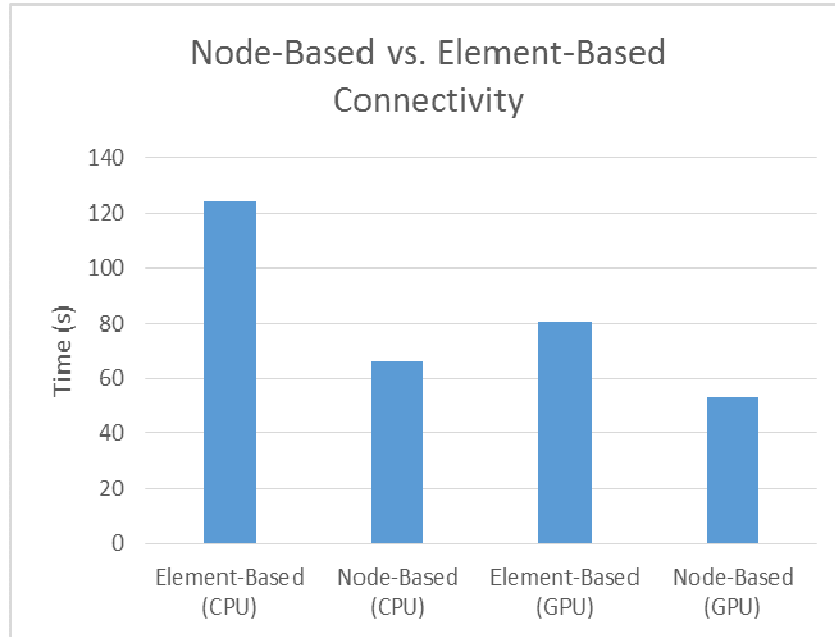


Figure 18: Cantilever Beam: Run-time comparison of Element-Based vs. Node-Based

	<i>Element-Based (CPU)</i>	<i>Node-Based (CPU)</i>	<i>Element-Based (GPU)</i>	<i>Node-Based (GPU)</i>
<i>run-time(s)</i>	124.15	66.3	80.39	53.3

Table 2: Cantilever Beam: Run-time comparison of Element-Based vs. Node-Based

7.3. Accuracy

The geometry is illustrated in Figure 19; material properties are those of steel. A force of 5000N was applied at time $t = 0$, and maintained thereafter. The total analysis time is 0.008 seconds with $\Delta t = 0.00002$ s; the damping coefficients are $\alpha^d = 0; \beta^d = 2E - 5$.

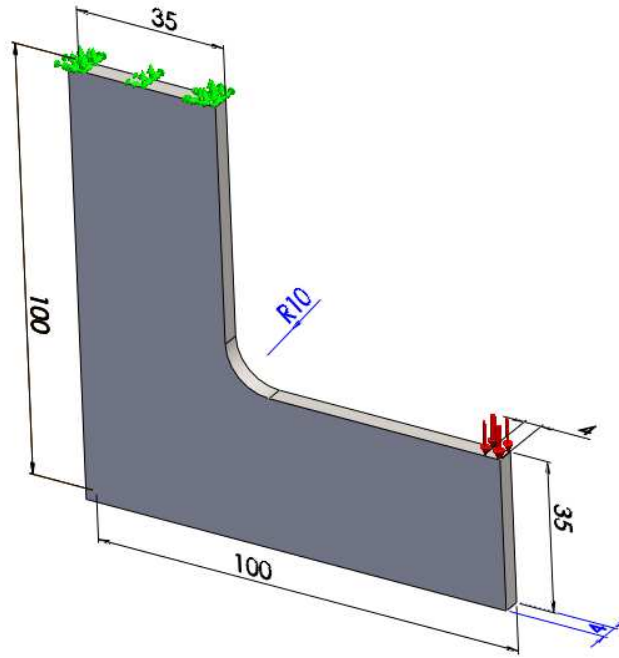


Figure 19: L-Bracket: geometry (dimensions in mm) and loading

The geometry was discretized using 4,000 elements. Despite the fact that we rely on a non-conforming voxel-mesh (as opposed to a high-quality conforming mesh in ANSYS), the stress predictions are fairly close. In fact the stress prediction with a 16 and 64 level refinements are similar to results obtained from ANSYS with 20,000 and 25,000 conforming elements.

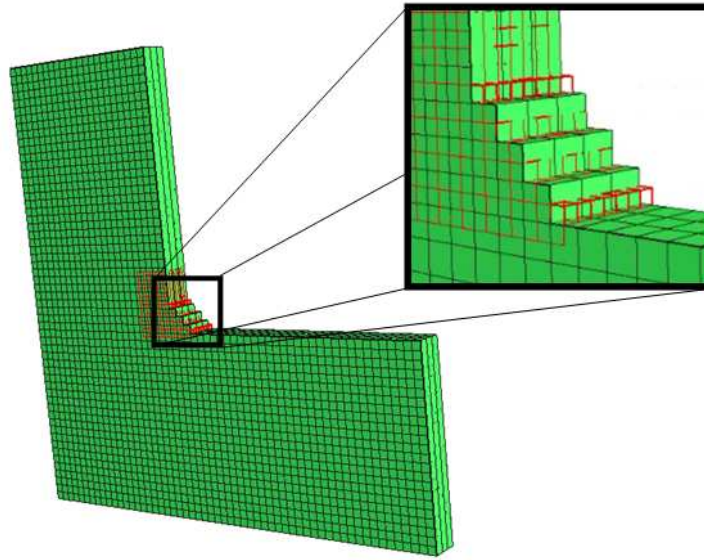


Figure 20: L-Bracket: Local Mesh Refinement

Table 3 shows the convergence of local mesh refinement.

<i>Refinement level</i>	<i>0</i>	<i>8</i>	<i>12</i>	<i>16</i>	<i>64</i>
<i>Max. Stress (MPa)</i>	<i>387.3</i>	<i>399.5</i>	<i>407.9</i>	<i>431.5</i>	<i>437.5</i>

Table 3: L-bracket: Convergence of local mesh refinement

Figure 21 shows the normalized maximum stress through time in AF-DCG and ANSYS.

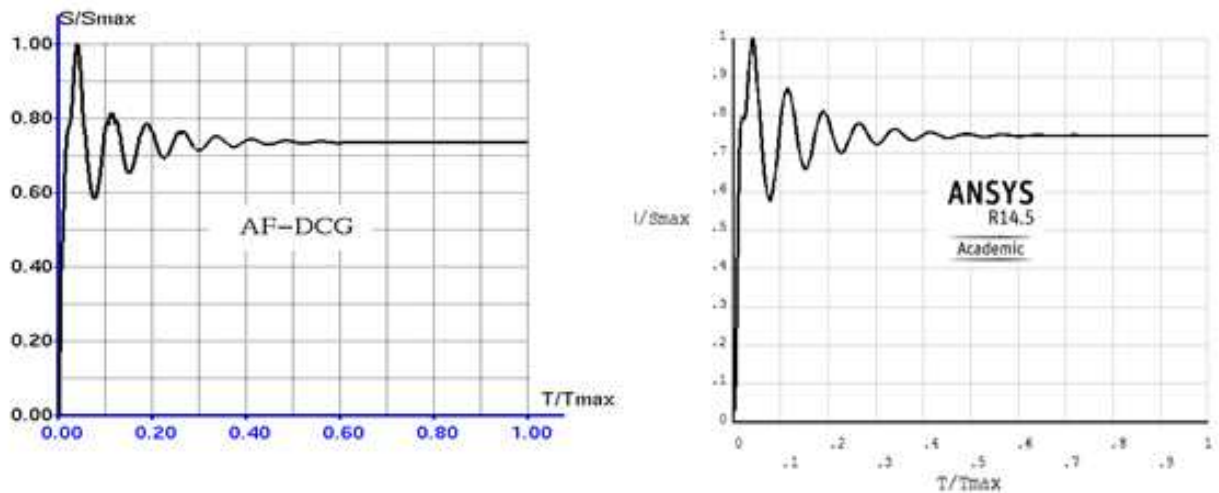


Figure 21: L-Bracket: The normalized maximum vonMises stress a)AF-DCG b) ANSYS

7.4. Importance of Deflation

The purpose of this experiment is to highlight the importance of deflation. The ‘plain’ CG can be exceedingly slow for large finite element models, especially for thin structures whose stiffness matrices are typically ill-conditioned.

For this experiment, the geometry and loading are illustrated in Figure 22.a, while Figure 22.b illustrates the static deformation. The geometry is discretized into 40000 hexahedral elements, and the analysis time 0.0125 seconds while $\Delta t = 0.0001$ s. The damping coefficients are $\alpha^d = 0; \beta^d = 2E - 5$.

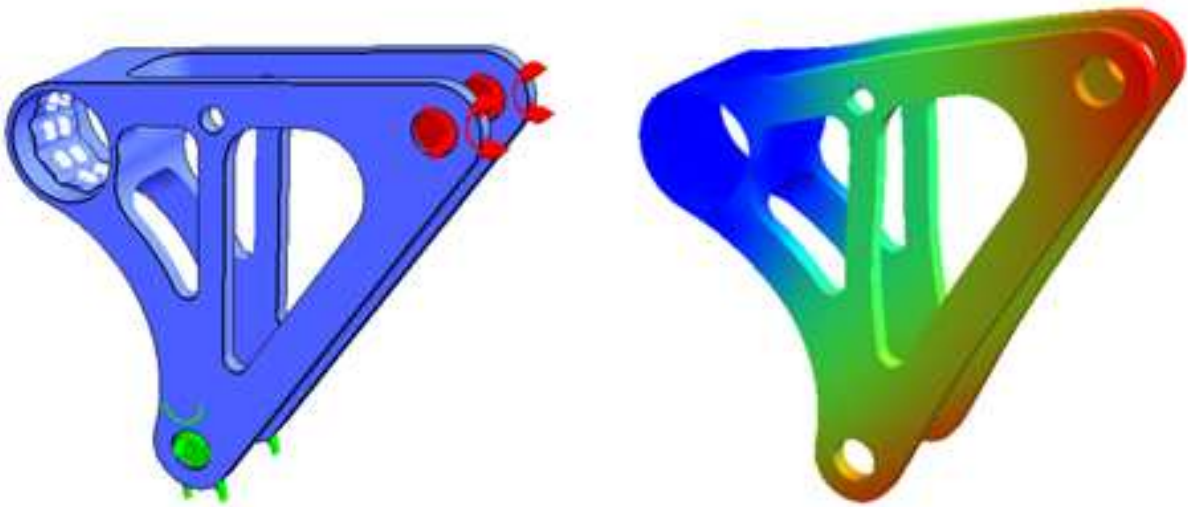


Figure 22: Rocker: a) Load condition b) Displacement field

Figure 23 and Table 4 compare the run time of AF-CG (i.e., without deflation) and AF-DCG (i.e., with deflation); the importance of deflation is self-evident.

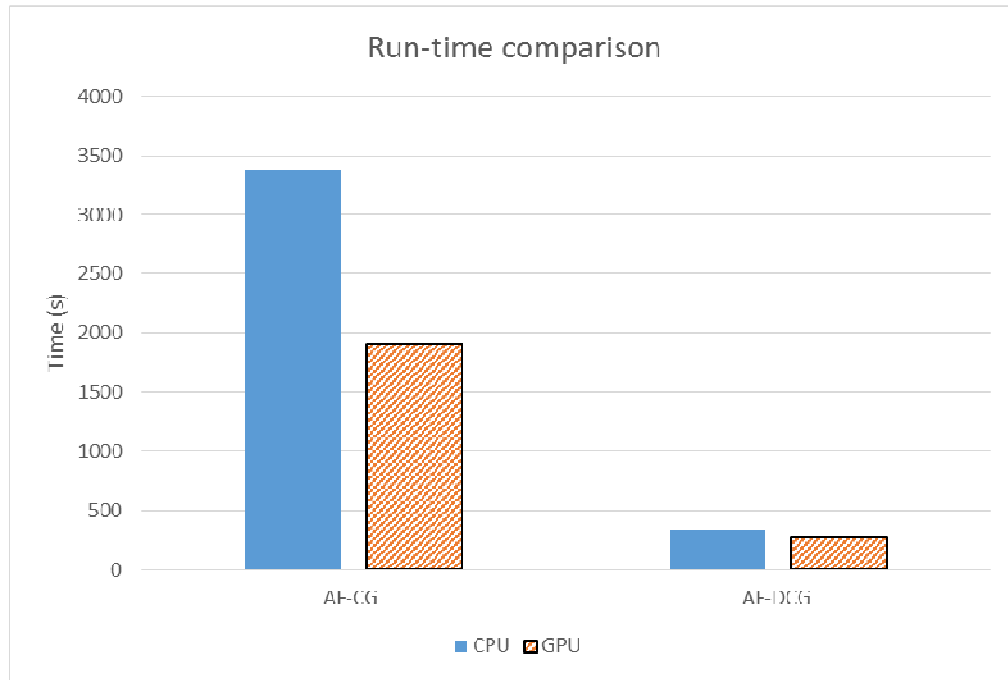


Figure 23: Rocker: Run-time comparison of AF-CG vs. AF-DCG

	<i>AF-CG (CPU)</i>	<i>AF-CG (GPU)</i>	<i>AF-DCG (CPU)</i>	<i>AF-DCG (GPU)</i>
<i>Run-time (s)</i>	3385	1905	336	277

Table 4: Rocker: Run-time comparison of AF-CG vs. AF-DCG

Next, we test the sensitivity of the run-time to number of agglomeration groups.

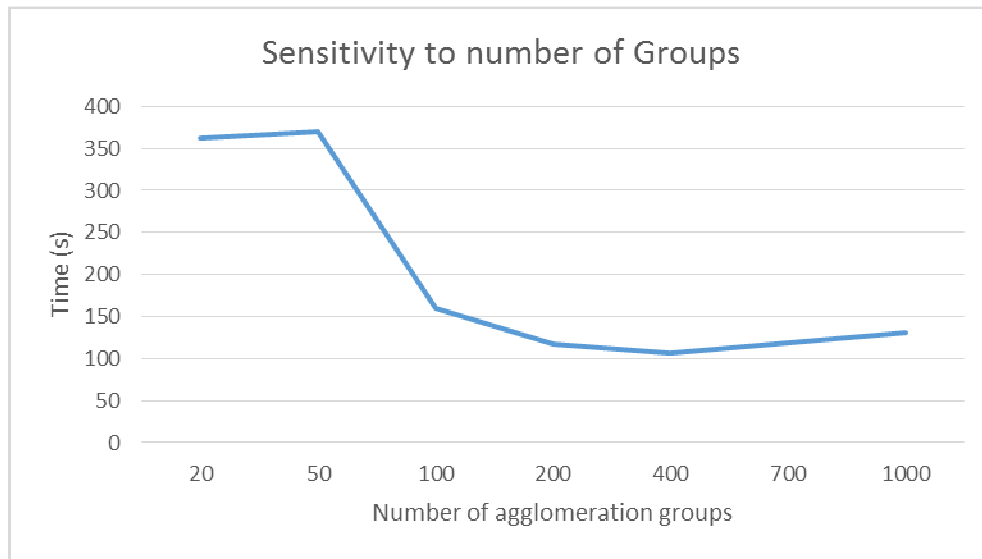


Figure 24: Rocker: Sensitivity of run-time to number of Groups

From Figure 24 we can conclude that as we increase the number of groups the run-time does not necessarily decrease and there is an optimum number of agglomeration groups (here about 400), which balances the extra time due to computation of groups (preprocess time) and reduced time in solving the linear system (process time).

7.5. Large-Scale Problems

In the present section, we present two large-scale examples to illustrate the advantages of the proposed method further. These problems are more realistic and are closer to the industrial model.

7.5.1. Battery Holder

This experiment illustrates the advantages of voxelization (as opposed to a conforming mesh). Consider the battery holder geometry in Figure 25, the small features present in the geometry can result in meshing-failure for a conforming mesh algorithm. A non-conforming voxel-mesh is insensitive to such details since it only approximates the geometry up to the resolution of the mesh. The geometry was discretized using 80,000 voxels. This resulted in a system with 317,000 nodal DOFs. A step-force of 1 N was applied on all the battery locations.

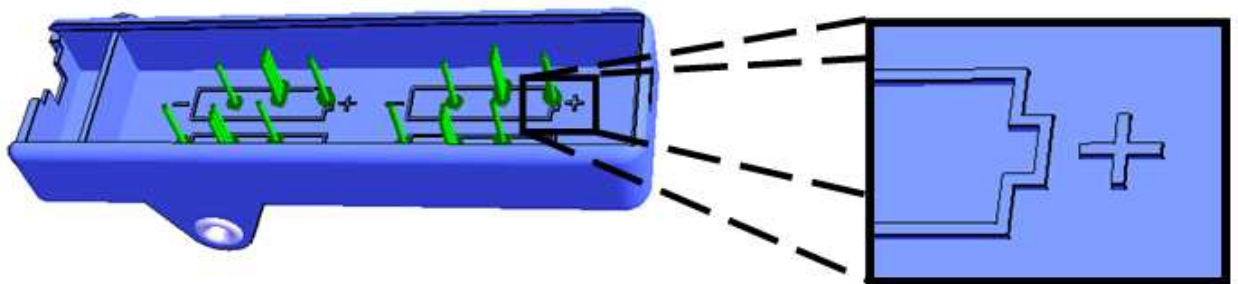


Figure 25: Battery holder: Thin structure with small features

The analysis is run for 0.04 seconds with a time step of 0.0001s. The damping coefficients are $\alpha^d = 0; \beta^d = 0.0001$. The study is done both on CPU and GPU using 1000 agglomeration groups, and the run-times are presented in Figure 26.

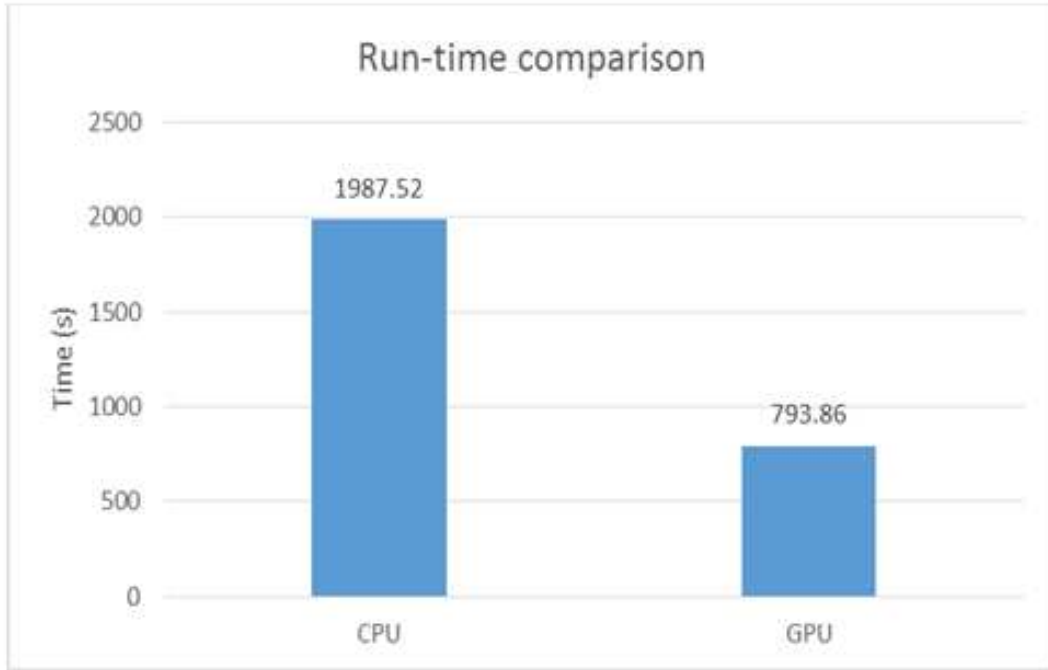


Figure 26: Battery holder: Run-time on CPU and GPU

Figure 27 shows Battery holder's displacement and stress field at $t = 0.04$ s.

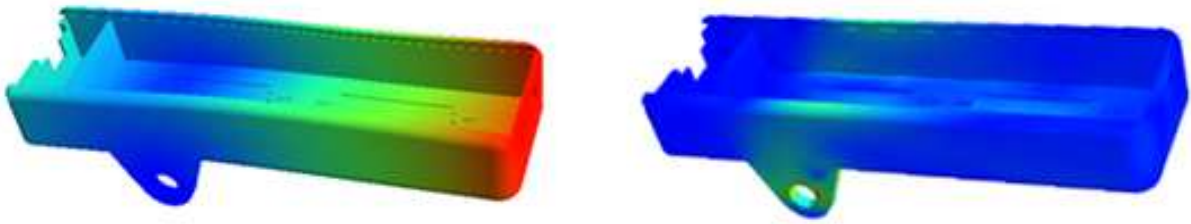


Figure 27: Battery holder: a) Displacement field b) Stress field

Figure 28 illustrates normalized maximum stress throughout the analysis.

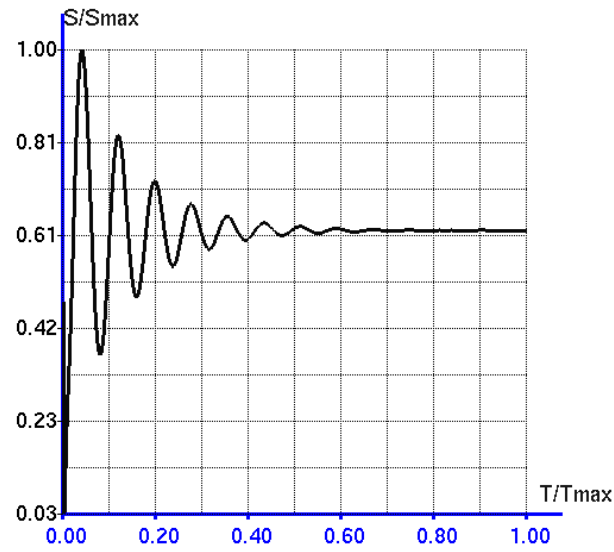


Figure 28: Battery holder: Normalized maximum stress over time

7.5.2. PCB (Arduino MEGA 2560)

Electric boards are perfect case studies for a large-scale transient FEA studies. They are usually very detailed filled with thin features; therefore require a very fine mesh. Further, they are constantly undergoing various dynamic loads, from manufacturing and shipping to operation, which makes them popular case studies for time-dependent analysis, especially fatigue analysis. Here, we study Arduino MEGA 2560 (Figure 1), a microcontroller widely used for R/C applications.

Since the model is highly detailed, the mesh has over 1 million DOFs or about 300,000 voxels (Figure 9). The sinusoidal force of Figure 29 is acting on the board as shown in Figure 30.

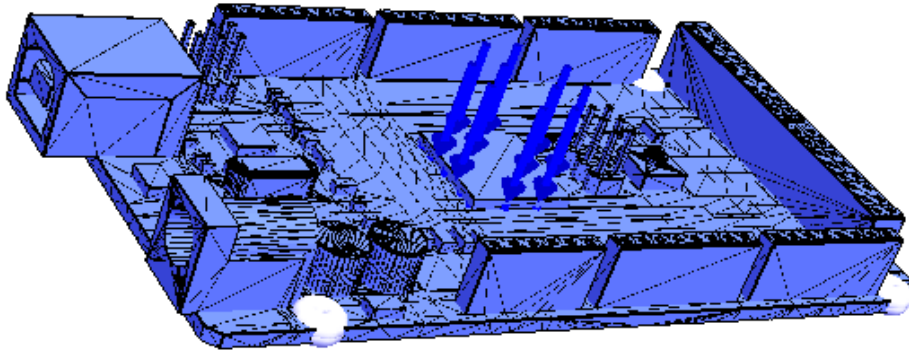


Figure 29: Arduino MEGA 2560: Loading and Boundary conditions

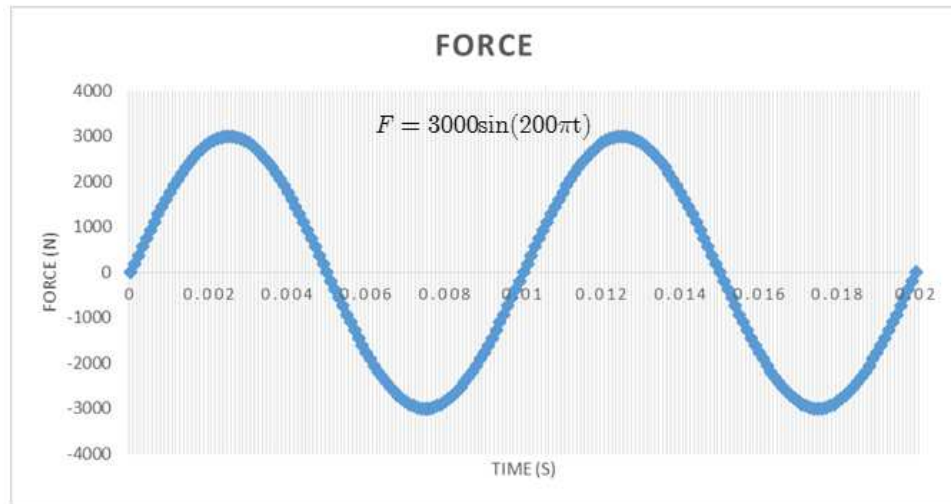


Figure 30: Arduino MEGA 2560: Sinusoidal force of $F = 3000(1 + \sin(20\pi t))$

The study is run on both CPU and GPU. As shown in Figure 31, the run-time on GPU is considerably shorter, which demonstrates the importance of parallelization in the proposed method.

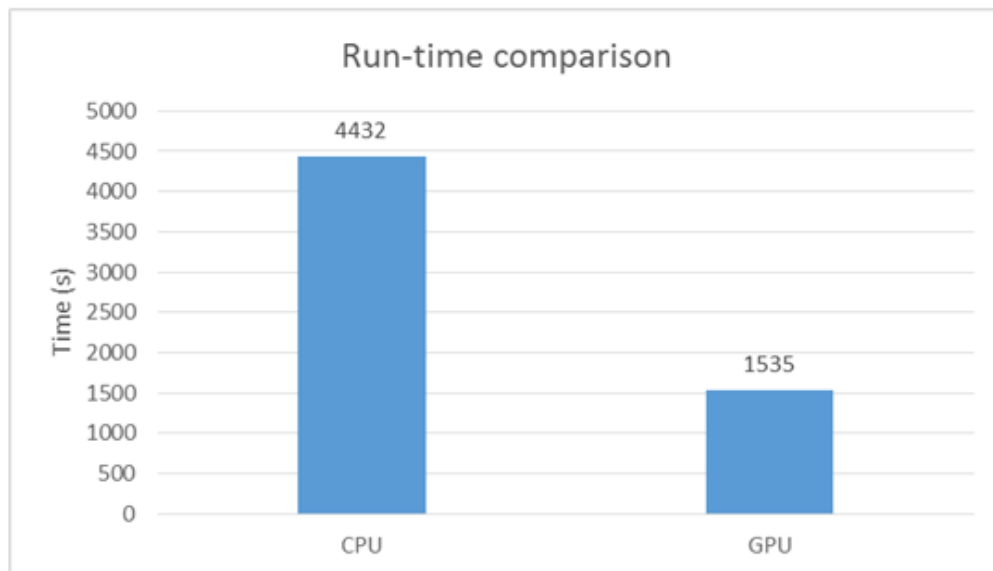
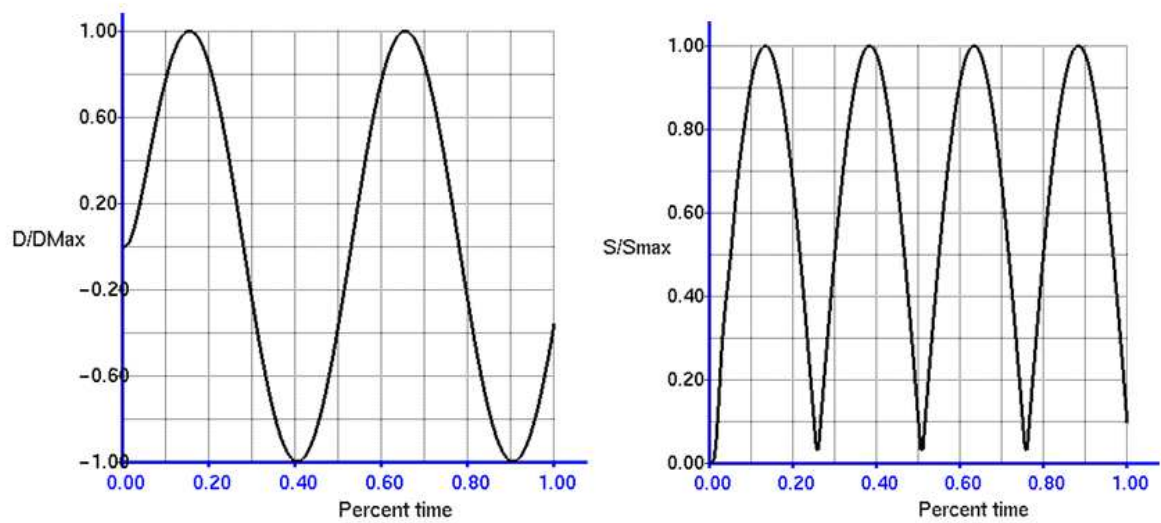


Figure 31: Arduino MEGA 2560: Run-time comparison between CPU and GPU

Figure 33 shows displacement field and stress fields caused by above loading condition.



**Figure 32: Arduino MEGA 2560: a) Normalized maximum displacement over time
b) Normalized maximum stress over time**

Finally, Figure 33 shows the displacement and stress fields at $t=0.02$ s.

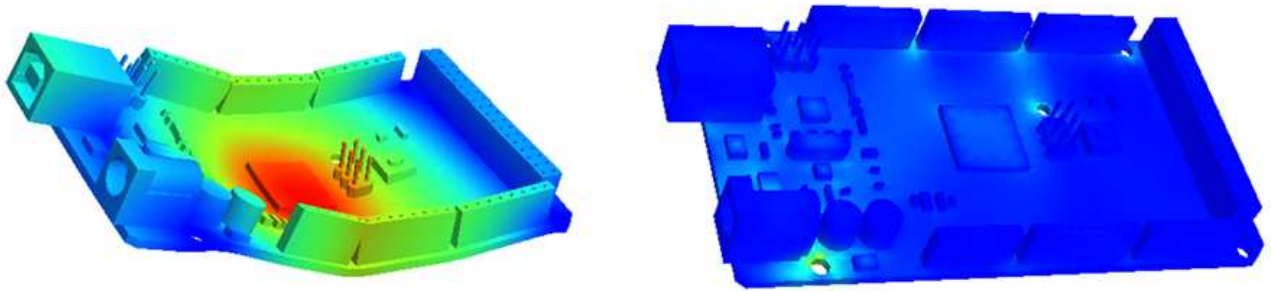


Figure 33: Arduino MEGA 2560: a) Displacement field b) Stress field

Chapter 8: Conclusion

8.1. Conclusion

A fast time-stepping approach for structural dynamics was developed based on Newmark-beta. The method merges different concepts to improve both speed and accuracy for large-scale problems. Comparisons were made with commercial software packages, which revealed the dominance of the proposed method with no loss in accuracy (provided the geometry can be accurately voxelized).

8.2. Future Work

The proposed AF-DCG can become the foundation for future work on Drop test, Fatigue modeling, and Crack propagation. Further, one of the most important applications of the proposed method is bone modelling and simulating the dynamics of bone structures for the following reasons and more:

- 1- CT scans represent bones in units of voxels, thus pre-processing time can reduce significantly.
- 2- Bones are often times large-scale models.
- 3- Researchers are usually interested in dynamic response of bones, which can take up to days to complete one run using current commercial software packages.

Moreover, there are still some weaknesses remained to solve in near future. Perhaps the most important one is stress prediction, which requires an accurate and robust algorithm to provide correct results, regardless of geometric complexities.

8.3. Overall Algorithm

The algorithm is quite similar to classic Newmark-beta method for transient elasticity problems, and proceeds as follows:

Algorithm: AF-DCG Newmark-beta; solve $M\ddot{u} + C\dot{u} + Ku = f$

1. Model the geometry, voxelize, and set initial conditions
2. Set material properties
3. Compute K_e and M_e
4. Set Newmark coefficients (β & γ)

Set damping coefficients (α^d & β^d)

Set duration and step size (T & Δt)

5. $K_e^{eff} = \eta K_e + \varsigma M_e$ where

$$\eta = 1 + \frac{\gamma\beta^d}{\beta h}, \quad \varsigma = \frac{1}{\beta\Delta t^2} + \frac{\gamma\alpha^d}{\beta\Delta t}$$

6. While $t_n \leq T$, do:

Update $f_{t+\Delta t}^{eff} = f_{t+\Delta t}^{ext} + f^C + f^M$

Solve $K_e^{eff} u_{t+\Delta t} = f_{t+\Delta t}^{eff}$ using DCG (section 3.3)

$$\ddot{u}_{t+\Delta t} = \frac{1}{\beta(\Delta t)^2} (u_{t+\Delta t} - u_t) - \frac{1}{\beta\Delta t} \dot{u}_t - \frac{1-2\beta}{2\beta} \ddot{u}_t$$

$$\dot{u}_{t+\Delta t} = \dot{u}_t + \Delta t \left[(1-\gamma) \ddot{u}_t + \gamma \ddot{u}_{t+\Delta t} \right]$$

Compute strains and stresses

Sort stresses and create local problems

Solve local problems

Compute fine Strains and stresses

7. End-While

Chapter 9: References

- [1] Cook, Robert D., David S. Malkus, Michael E. Plesha, and Robert J. Witt. *Concepts and Applications of Finite Element Analysis, 4th Edition*. 4 edition. New York, NY: Wiley, 2001.
- [2] Hughes, Thomas J. R., Itzhak Levit, and James Winget. “An Element-by-Element Solution Algorithm for Problems of Structural and Solid Mechanics.” *Computer Methods in Applied Mechanics and Engineering* 36, no. 2 (February 1983): 241–54. doi:10.1016/0045-7825(83)90115-9.
- [3] Love, A. E. H. *A Treatise on the Mathematical Theory of Elasticity*. 4 edition. New York: Dover Publications, 2011.
- [4] Newmark, N. M. “A Method of Computation for Structural Dynamics.” *Proc. ASCE* 85, no. 3 (1959): 67–94.
- [5] Keierleber, Colin Walker. “Higher-Order Explicit and Implicit Dynamic Time Integration Methods.” *ETD Collection for University of Nebraska - Lincoln*, January 1, 2003, 1–406.
- [6] Cook, Robert D. *Finite Element Modeling for Stress Analysis*. 1st edition. New York: Wiley, 1995.
- [7] Hackbusch, Wolfgang. *Multi-Grid Methods and Applications*. Springer, 2003.
- [8] Hulbert, Gregory M. “Time Finite Element Methods for Structural Dynamics.” *International Journal for Numerical Methods in Engineering* 33, no. 2 (January 30, 1992): 307–31. doi:10.1002/nme.1620330206.
- [9] Adams, Mark. *Evaluation of Three Unstructured Multigrid Methods on 3D Finite Element Problems in Solid Mechanics*. University of California, Berkeley, Computer Science Division, 2000.
- [10] Arbenz, Peter, G. Harry van Lenthe, Uche Mennel, Ralph Müller, and Marzio Sala. “A Scalable Multi-Level Preconditioner for Matrix-Free M-Finite Element Analysis of Human Bone Structures.” *International Journal for Numerical Methods in Engineering* 73, no. 7 (February 12, 2008): 927–47. doi:10.1002/nme.2101.

- [11] Aubry, R., F. Mut, S. Dey, and R. Löhner. “Deflated Preconditioned Conjugate Gradient Solvers for Linear Elasticity.” *International Journal for Numerical Methods in Engineering* 88, no. 11 (December 16, 2011): 1112–27. doi:10.1002/nme.3209.
- [12] Augarde, C. E., A. Ramage, and J. Staudacher. “An Element-Based Displacement Preconditioner for Linear Elasticity Problems.” *Computers & Structures* 84, no. 31–32 (December 2006): 2306–15. doi:10.1016/j.compstruc.2006.08.057.
- [13] Bell, Nathan, and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.
<http://sbel.wisc.edu/Courses/ME964/Literature/techReportGarlandBell.pdf>.
- [14] Briggs, William L. “A Multigrid Tutorial.” Accessed May 4, 2014.
<http://www.math.ust.hk/~mawang/teaching/math532/mgtut.pdf>.
- [15] Düster, A., J. Parvizia, Z. Yang, and E. Rank. “The Finite Cell Method for Three-Dimensional Problems of Solid Mechanics.” *Computer Methods in Applied Mechanics and Engineering* 197, no. 45–48 (August 15, 2008): 3768–82.
doi:10.1016/j.cma.2008.02.036.
- [16] Göddeke, Dominik, Robert Strzodka, and Stefan Turek. “Performance and Accuracy of Hardware-Oriented Native-, Emulated- and Mixed-Precision Solvers in FEM Simulations.” *International Journal of Parallel, Emergent and Distributed Systems* 22, no. 4 (2007): 221–56. doi:10.1080/17445760601122076.
- [17] Karabassi, Evaggelia-Aggeliki, Georgios Papaioannou, and Theoharis Theoharis. “A Fast Depth-Buffer-Based Voxelization Algorithm.” *Journal of Graphics Tools* 4, no. 4 (1999): 5–10. doi:10.1080/10867651.1999.10487510.
- [18] Mao, K. M., and C. T. Sun. “A Refined Global-Local Finite Element Analysis Method.” *International Journal for Numerical Methods in Engineering* 32, no. 1 (July 1, 1991): 29–43. doi:10.1002/nme.1620320103.
- [19] Requicha, Aristides G. “Representations for Rigid Solids: Theory, Methods, and Systems.” *ACM Comput. Surv.* 12, no. 4 (December 1980): 437–64.
doi:10.1145/356827.356833.

- [20] Saad, Y., M. Yeung, J. Erhel, and F. Guyomarc'h. "A Deflated Version of the Conjugate Gradient Algorithm." *SIAM Journal on Scientific Computing* 21, no. 5 (January 1, 2000): 1909–26. doi:10.1137/S1064829598339761.
- [21] Saad, Yousef. *Iterative Methods for Sparse Linear Systems: Second Edition*. SIAM, 2003.
- [22] Taiebat, Hossein H., and J. P. Carter. "Three-Dimensional Non-Conforming Elements." *Centre for Geotechnical Research, The University of Sydney, Sydney* 808 (2001): 2001.
- [23] Wilson, E. L., R. L Taylor, W. P. Doherty, and J. Ghaboussi. "Incompatible Displacement Models(isoparametric Finite Elements in Solid and Thick Shell Structural Analysis)." *Numerical and Computer Methods in Structural Mechanics A* 74–17756 06–32 (1973): 43–57.
- [24] Yadav, Praveen, and Krishnan Suresh. "Assembly-Free Large-Scale Modal Analysis on the Graphics-Programmable Unit." *Journal of Computing and Information Science in Engineering* 13, no. 1 (2013): 011003.
- [25] Zangmeister, T., H. Andrä, and R. Müller. "Comparison of XFEM and Voxelbased FEM for the Approximation of Discontinuous Stress and Strain at Material Interfaces." *TECHNISCHE MECHANIK* 33, no. 2 (2013): 131–41.
- [26] Zienkiewicz, Olek C., and Robert L. Taylor. *The Finite Element Method for Solid and Structural Mechanics*. Butterworth-Heinemann, 2005.
- [27] "ECE/ME/EMA/CS759 High Performance Computing Lectures and Course Material 2013." Accessed May 5, 2014. <http://sbel.wisc.edu/Courses/ME964/2013/>.
- [28] "Arduino Mega 2560 - SOLIDWORKS, Other, STEP / IGES - 3D CAD Model - GrabCAD." Accessed May 4, 2014. <http://grabcad.com/library/arduino-mega-2560--1>.
- [29] Barney, Blaise, and Lawrence Livermore. "OpenMP." Accessed May 4, 2014. <https://computing.llnl.gov/tutorials/openMP/>.
- [30] "3D CAD Design Software SolidWorks." Accessed May 4, 2014. <https://www.solidworks.com/>.
- [31] "ANSYS - Simulation Driven Product Development." Accessed May 4, 2014. <http://www.ansys.com/>.

- [32] “OpenMP.org.” Accessed May 4, 2014. <http://openmp.org/wp/>.
- [33] “Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA.” Accessed May 4, 2014. http://www.nvidia.com/object/cuda_home_new.html.