Accelerating Robustness Prediction

Julian Panetta

December 26, 2012

Abstract

The falling prices of $3D$ printers and $3D$ printing services have made rapid prototyping technologies increasingly accessible. Artists and engineers skilled with a $3D$ modeling program can design shapes and have them cheaply, quickly, and easily fabricated in a variety of materials. As this process becomes widespread, however, it is essential to develop tools to predict the strength of the fabricated model; insufficiently reinforced models will break in the customer's hands and might not even survive the printing process. Such a tool should run fast enough not to discourage its use and ideally would provide real-time feedback while the user is modeling. Although such interactive feedback is still out of reach, this paper discusses how to accelerate various stages of an existing robustness prediction pipeline currently under development by James Zhou and Denis Zorin.

1 Introduction

First, we briefly discuss a (over-)simplified view of the existing robustness prediction pipeline and the particular stages that this paper will target for acceleration. The pipeline accepts a triangle mesh of the shape a user wishes to print. It first converts this surface mesh into a $3D$ mesh made up of tetrahedra:

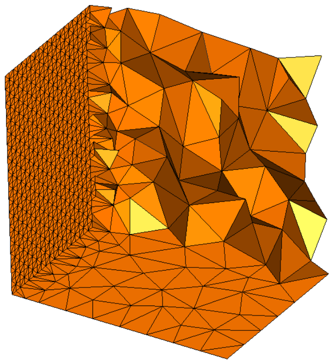


Figure 1 *Cut-away view of a tetrahedral mesh for a bar.*

The pipeline then predicts how the model is likely to deform, and for each of those deformations it computes the corresponding internal force within each tetrahedron. If this internal force exceeds some physically meaningful threshold, the tetrahedron is predicted to break.

1.1 Predicting Deformations

Deformations of an object can be expressed as a function, ϕ, mapping points on the undeformed object to the deformed object. In turn, deformation ϕ can be written in terms of a displacement field, u, over the undeformed object, Ω:

$$\phi(X) = X + u(X) \quad X \in \Omega$$

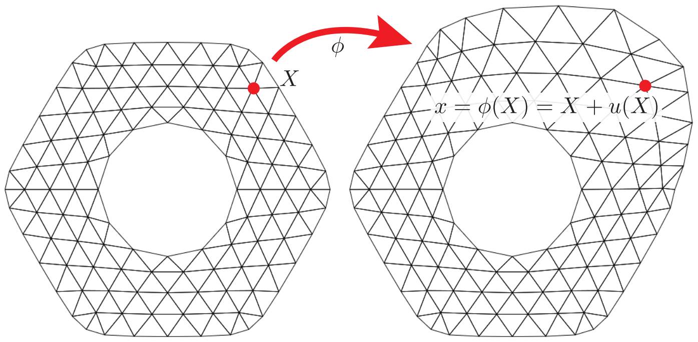


Figure 2 *An example deformation, ϕ, inducing some internal potential energy and forces.*

The entire pipeline is built around a piecewise linear FEM discretization of the linear elasticity model. With this discretization, u is expressed as a per-vertex displacement vector field that is linearly interpolated within each tetrahedron using the standard linear shape functions. The interpretation of u is clear: the 3-vector $\mathbf{u_i} = u(x_i)$ at undeformed vertex x_i is the displacement carrying x_i to it's deformed position.

Because we use linear elasticity, the internal forces will be a linear function of the displacement. After discretization, this relationship is encoded in the stiffness matrix, $\mathbf{K}$. This matrix maps a "flattened" displacement vector $\mathbf{u} = (u_{0x}, u_{0y}, u_{0z}, \dots)^T$ to a "flattened" force vector

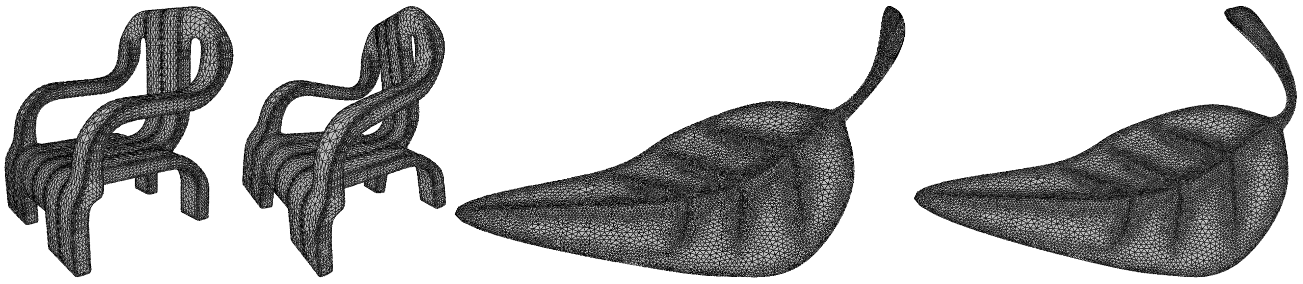


Figure 3 *Chair and leaf models deformed by their lowest (nonzero) energy modal displacements. The undeformed model is shown on the left, and the deformed model on the right.*

$\mathbf{f} = (f_{0_x}, f_{0_y}, f_{0_z}, \dots)^T$:

$$\mathbf{f} = -\mathbf{K}\mathbf{u}$$

With this sign convention, the stiffness matrix will be symmetric positive semidefinite. There is another symmetric matrix, the mass matrix $\mathbf{M}$, that accounts for how mass is distributed throughout the model. The mass matrix is positive definite and can be approximated by a diagonal "lumped mass matrix." This approximation corresponds to assigning to each vertex one quarter of the mass of each adjacent tetrahedron.

These matrices are all that is needed to write the ODE simulating a model's behavior (in the absence of damping and external forces):

$$\mathbf{M}\ddot{\mathbf{u}} = -\mathbf{K}\mathbf{u}$$

An arbitrary displacement can be decomposed into the full basis of "modal displacements," which are solutions to the generalized eigenvalue problem:

$$\mathbf{K}\mathbf{u}_\lambda = \lambda\mathbf{M}\mathbf{u}_\lambda$$

These basis vectors are special because they evolve very simply; their amplitudes vary sinusoidally over time:

$$\mathbf{M}\ddot{\mathbf{u}}_\lambda = -\mathbf{K}\mathbf{u}_\lambda = -\lambda\mathbf{M}\mathbf{u}_\lambda \quad \Longrightarrow \quad \ddot{\mathbf{u}}_\lambda = -\lambda\mathbf{u}_\lambda$$

The energy stored in mode $\mathbf{u}_\lambda$ depends on λ, with the lowest energy modes corresponding to the smallest eigenvalues. These lowest-energy modes are the cheapest to excite, and are thus the most likely deformations to occur. This means that predicting the model's likely deformations amounts to solving the generalized eigenvalue problem.

This task is called modal analysis. The smallest 6 eigenvalues will be zero, corresponding to the 6 rigid degrees of freedom (3 independent translations and rotations) that do not change the model's shape. The 7^{th} eigenvalue and up correspond to the likely shape deformations. The lowest nonzero modes for two example models are shown in Figure 3.

1.2 Determining Breakages

Given a deformation, we need to decide if the model is going to break; we need to compute the maximum resulting internal forces and check if they exceed some threshold.

The internal forces within each tetrahedron are expressed in terms of the stress tensor, σ. In a continuous model, σ would be a 3×3 matrix-valued function defined over the entire object. After our linear discretization, however, the stress tensor is piecewise constant, and σ for each tetrahedron is just a 3×3 matrix determined by the four vertices' positions and displacements.

Intuitively, the stress tensor σ tells you that if you were to slice the element along a plane with unit normal $\hat{\mathbf{n}}$, each half would exert a force $\sigma\hat{\mathbf{n}}$ on the other. With our model, the stress tensor is symmetric. Therefore, the maximum-magnitude eigenvalue tells you the magnitude of force in the direction of maximum internal force.

To compute the maximum internal forces, then, we must compute the maximum-magnitude eigenvalue of each element's 3×3 stress tensor. This computation is examined in Section 4.

2 Modal Analysis

Here we focus on the task of solving the generalized eigenvalue problem:

$$\mathbf{K}\mathbf{u}_\lambda = \lambda\mathbf{M}\mathbf{u}_\lambda.$$

for the smallest few eigenvalues and eigenvectors. $\mathbf{K}$ and $\mathbf{M}$ are both extremely large matrices (each has three rows and columns for each vertex in the mesh). However, they can be stored efficiently because they are quite sparse. To keep the problem in memory, we must use an algorithm that leaves everything in a sparse format. This means using an iterative algorithm.

We can either use an algorithm designed for the generalized eigenvalue problem or transform our problem into the traditional form. Since $\mathbf{M}$ is positive definite, we can compute it's Cholesky factorization $\mathbf{M} = \mathbf{L}\mathbf{L}^T$ (this is particularly easy if $\mathbf{M}$ is a diagonal lumped mass matrix). Then, letting $\mathbf{x}_\lambda = \mathbf{L}^T\mathbf{u}_\lambda$, we can solve the equivalent problem

$$\mathbf{K}\mathbf{u}_\lambda = \lambda\mathbf{L}\mathbf{L}^T\mathbf{u}_\lambda \quad \Longleftrightarrow \quad \underbrace{\mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-1^T}}_{\mathbf{A}}\mathbf{x}_\lambda = \lambda\mathbf{x}_\lambda$$

where $\mathbf{A}$ is still symmetric and sparse.

The simplest iterative technique for finding eigenvalues is the power method. This method iteratively applies $\mathbf{A}$ to a random vector $\mathbf{x}_0$, magnifying most the component for the eigenvector with largest eigenvalue ($\mathbf{v}_{max}$). Assuming a good separation of eigenvalue magnitudes, this component eventually dominates:

$$\mathbf{x}_k = \frac{\mathbf{A}^k \mathbf{x}_0}{\|\mathbf{A}^k \mathbf{x}_0\|} \approx \mathbf{v}_{max}.$$

The smallest magnitude eigenvalues or the eigenvalues closest to some scalar, μ, can be found by the shift-invert power method. This simply applies the power method to $(\mathbf{A} - \mu\mathbf{I})^{-1}$.

However, when a set of the smallest or largest eigenvalues is desired, the Lanczos algorithm is a better tool. Instead of just choosing the last iterate of the power method, the Lanczos algorithm searches for the optimal eigenvector approximation in the span of all iterates:

$$\mathcal{K}(\mathbf{A}, \mathbf{x}_0, k) = \mathrm{span}(\mathbf{x}_0, \mathbf{A}\mathbf{x}_0, \mathbf{A}^2\mathbf{x}_0, \ldots, \mathbf{A}^k\mathbf{x}_0)$$

This is a special space known as the Krylov subspace. In [5] and [6], the Krylov subspace is shown to contain close approximations to both the smallest and largest eigenvalues' eigenvectors. In other words, if one finds an orthogonal basis for the k^{th} Krylov subspace, $\mathbf{Q}_k$, then the extreme eigenvalues of $\mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k$ are a close approximation to the extreme eigenvalues of $\mathbf{A}$.

Even better, an orthogonal basis for the Krylov subspace can be chosen that tri-diagonalizes $\mathbf{A}$:

$$\mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k = \mathbf{T}_k = \begin{bmatrix} \alpha_0 & \beta_1 & & & \\ \beta_1 & \alpha_1 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

Finding the orthogonal columns of $\mathbf{Q}_k$ and the entries of $\mathbf{T}_k$ can be done with an efficient 3-term recurrence equation that is the foundation of the Lanczos algorithm:

$$\begin{aligned} \mathbf{A}\mathbf{q}_k &= \beta_{k-1}\mathbf{q}_{k-1} + \alpha_k \mathbf{q}_k + \beta_k \mathbf{q}_{k+1} \\ \alpha_k &= \mathbf{q}_k \mathbf{A} \mathbf{q}_k \\ \beta_k &= \|\mathbf{A}\mathbf{q}_k - \beta_{k-1}\mathbf{q}_{k-1} + \alpha_k \mathbf{q}_k\| \end{aligned}$$

This recurrence is derived by looking at the columns of equation $\mathbf{AQ} = \mathbf{QT}$. Beware that, in practice, the iterate vectors occasionally must be re-orthogonalized because inexact arithmetic will allow components of the previous iterate vectors to creep back in.

After running k iterations of the Lanczos algorithm, a direct method like QR can be applied to the tri-diagonal matrix $\mathbf{T}_k$ to obtain the maximal eigenvalue and eigenvector approximations for $\mathbf{A}$. This post-processes typically takes negligible time.

As stated, the Lanczos algorithm's dominant cost is the sparse matrix vector multiply (MatVec) needed once per iteration. This is good because sparse MatVecs are fairly cheap and can be parallelized with various different techniques (see Section 3). But unfortunately, for the modal analysis matrices, plain Lanczos does not seem to converge to the smallest eigenvalues even after thousands of iterations. The typical approach in this situation is to use the shift-invert Lanczos algorithm (apply Lanczos to $(\mathbf{A} - \mu\mathbf{I})^{-1}$).

The shift-invert Lanczos algorithm (with $\mu \approx 0$) converges quickly to the smallest eigenvalues, but it is not as cheap or trivial to parallelize. It requires a Cholesky factorization of $(\mathbf{A} - \mu\mathbf{I})^{-1}$ as a preprocessing step and then two back substitutions per iteration (in place of the MatVec). The Cholesky factorization incurs some additional memory costs—around $4\times$ for our matrices with fill-in reducing permutations—and takes a few seconds. This is undesirable when the goal is a real-time, interactive tool. Also, the back substitution at each iteration, though cheap, is nontrivial to parallelize.

I tried several tricks to avoid this factorization. First, I tried using CG to implement the $(\mathbf{A} - \mu\mathbf{I})^{-1}$ solve. This worked very poorly, partly because of how ill-conditioned $\mathbf{A} - \mu\mathbf{I}$ is. It is also likely that using CG, which searches the Krylov subspaces of $(\mathbf{A} - \mu\mathbf{I})$, is a bad choice for generating basis vectors for the Krylov subspace $(\mathbf{A} - \mu\mathbf{I})^{-1}$.

I also tried existing implementations of other, more modern Krylov-based eigenvalue solvers. For example, the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) Method [7]. It belongs to a class of methods, including [10], and [9], that directly attempt to minimize the Raleigh quotient (minimized by $\mathbf{v}_{min}$):

$$\rho(x) = \frac{\mathbf{x}^T \mathbf{A} \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$$

using nonlinear CG-style algorithms. At least without a better preconditioner, none of these could converge to $\mathbf{A}$'s smallest eigenvalues.

In the end, I focused on parallelizing the MatVec in the hope that eventually I will find an algorithm (or a better preconditioner) that converges well to my matrices' smallest eigenvalues.

3 Sparse MatVec

There are many techniques for accelerating the sparse matrix-vector multiply:

$$\mathbf{b} = \mathbf{A}\mathbf{x}$$

This task is difficult because only two floating point operations are performed for every three or four memory accesses. Also, depending on the entry numbering and ordering, the vector element accesses can be almost random, destroying cache performance. Therefore, in addition to parallelizing the multiplication, we must optimize how the matrix and vector are accessed and stored.

I evaluated various approaches for the GPU (Section 3.1) but ended up working on a CPU-based implementation (Section 3.2); the GPU performance was not particularly impressive, and I found the challenges of a fast CPU implementation more interesting.

3.1 GPU Sparse MatVec

NVIDIA has released a library for sparse matrix computations on their GPUs called CUSP [2], and it includes the optimized multiply routines and storage formats described in [1].

The CUSP library implements four formats for general sparse matrices: COO, CSR, ELL, and Hybrid. The COO ("coordinate") format simply stores a triplet per nonzero entry with that entry's row index, column index, and value. The CSR ("Compressed Sparse Row") format is another standard format that allocates two big arrays to store all the nonzero entries' column indices and values. These arrays are arranged so that all the elements in the same row are contiguous. Then, a third array holds an index per row that points to the start of that row's entries in the column index and value tables. The ELL format (ELLPACK or ITPACK) is a variant of CSR that pads each CSR row with extra zeros so that each row has the same number of entries. If the matrix has approximately the same number of nonzeros in each row, this format can give better performance on SIMD architectures. If there are a few rows with many more nonzeros than average, the Hybrid format can be used. This uses ELL to pad small rows up to a selected size, then handles the remaining elements of the large rows with the COO format.

I benchmarked CUSP on a NVIDIA GeForce GTX 590 GPU. Because there are some rows in our matrices with many more nonzeros than others, the ELL format was unable to fit into the GPU's memory; the chair model's stiffness matrix had a minimum row nonzero count of 12, median of 31, and maximum of 114. This variance is common to tetrahedral meshes—the interior vertices have much higher valence than the surface vertices.

The performance for the chair and leaf models is shown in Table 1 and Table 2, respectively. Likely because of the great variation in row nonzero count, the CSR format was the clear winner. The same was true on my laptop's NVIDIA GeForce GT 320M, though performance was considerably lower (see Tables 3 and 4).

Method	MatVecs/sec	GFLOPS
COO	813.543934	5.253709
HYB	1071.262154	6.918003
CSR	1753.814578	11.325794

Table 1 *GeForce GTX 590 CUSP performance for the chair model's stiffness matrix (*$90555 \times 90555,\ 3228903$ *nonzeros).*

Method	MatVecs/sec	GFLOPS
COO	856.652653	4.873226
HYB	1157.747567	6.586060
CSR	1857.880491	10.568895

Table 2 *GeForce GTX 590 CUSP performance for the leaf model's stiffness matrix (*$82530 \times 82530,\ 2844342$ *nonzeros).*

Method	MatVecs/sec	GFLOPS
COO	186.696985	1.205653
HYB	219.100968	1.414912
CSR	252.776376	1.632381

Table 3 *GeForce GT 320M CUSP performance for the chair model's stiffness matrix (*$90555 \times 90555,\ 3228903$ *nonzeros).*

Method	MatVecs/sec	GFLOPS
COO	208.939126	1.188589
HYB	245.431561	1.396183
CSR	272.717438	1.551403

Table 4 *GeForce GT 320M CUSP performance for the leaf model's stiffness matrix (*$82530 \times 82530,\ 2844342$ *nonzeros).*

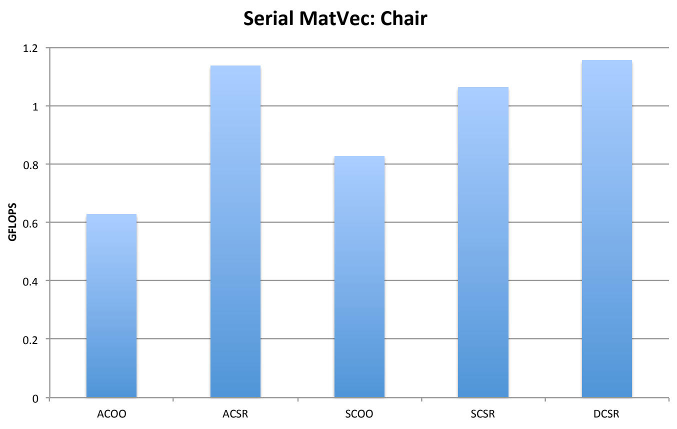
3.2 CPU Sparse MatVec

3.2.1 Serial

I first implemented several serial versions of a sparse matrix-vector multiply. I did a plain COO and CSR implementation as described in Section 3.1. Keeping in mind that the multiply is memory-bound, I took took advantage of the symmetry of our matrices to nearly halve the matrix memory accesses: I stored only the lower triangle and modified the multiply routines to regard each off-diagonal nonzero (i, j) as also an upper triangle entry (j, i). This lead to the SCOO and SCSR formats (symmetric variants of COO and CSR).

There are two downsides to the symmetric variants. First, they require more accesses to the output vector, and the extra accesses are non-sequential. Second, the diagonal nonzeros have to be treated differently because they only count as a single entry (rather than as two entries like the off-diagonals). This requires an new "if" statement in both implementations. Branch prediction prevents this from being too costly, but it still has a performance impact.

The memory access problem is hard to avoid, but I eliminated the branching with my DCSR (diagonal + CSR) format that stores the diagonal entries in a dense vector and the off-diagonal entries in CSR. The diagonal entries are then handled with an SSE vectorized component-wise multiplication, and the off-diagonals are handled by an "if"-less symmetric CSR multiply.



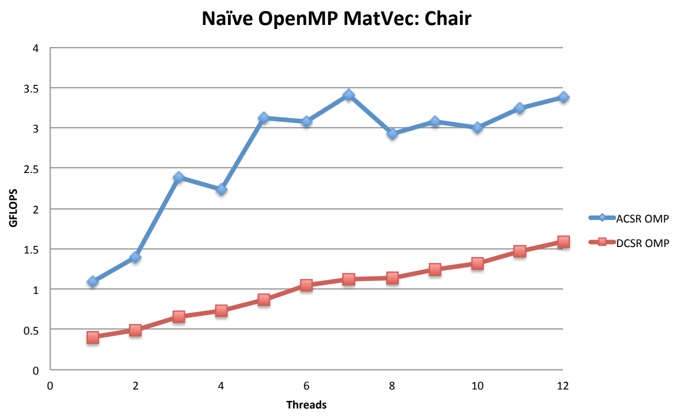


Figure 4 *Performance of the serial MatVec (left) and naïve OpenMP (right) CPU implementations for the chair stiffness matrix on a* 12 *core Intel Xeon X5650 @ 2.67GHz*

The names of all the serial implementations are summarized in Table 5, and their performances are in Figure 4. The Diagonal + Symmetric CSR Format is the narrow winner, and we will see it has an even greater lead when used as the basis for a (smart) parallel implementation.

ACOO, ACSR	Plain (Asymmetric) COO, CSR Format
SCOO, SCSR	Symmetric COO, CSR Format
DCSR	Diagonal + Symmetric CSR Format

Table 5 *Serial CPU MatVec implementation names*

3.2.2 Naïve OpenMP

Because each row of the output vector is computed independently in ACSR, this implementation is trivial to parallelize using OpenMP. This gives a modest 3× speedup on a 12 core machine (Figure 4). Due to the random output vector accesses of the SCSR implementation, atomic operations are needed for a parallel version during emulation of the upper triangle elements. This has a substantial performance impact: the parallel version is actually slower than the single-threaded version until 7 threads are used, and the peak performance is only about 1.35× faster on the 12 core machine (Figure 4).

3.2.3 Partitioning

There are two big problems with the naïve implementations above. First, there is a severe load imbalance; we have already emphasized the high variation in nonzero count our matrix rows exhibit. Second, unless we happen to have a lucky variable numbering/matrix ordering, each thread must access entries spread throughout the entire input vector. Both of these problems can be addressed by a partitioned MatVec inspired by [4].

A given component of the result, b_i, depends only on the input vector components x_j for $(i, j) \in \mathbf{A}$. In other words, the column indices of nonzeros in row i enumerate all the data needed to compute b_i. We could hope to form a cluster of output entries that depend only on the same input data. Then that input data could be cached, improving memory access time.

Since our matrices are symmetric with a nonzero diagonal, the hope is even better: the cluster's input entry indices will be identical to its output entry indices. Thus, when repeatedly applying $\mathbf{A}$ to compute the Krylov vectors, one cluster's elements of all the Krylov vectors can be computed independently of the other clusters'. So, $\mathbf{A}$ and $\mathbf{x}$ can be partitioned into these clusters, and each part can be sent to a separate processor/computation node where it hopefully fits in cache.

The clusters we allude to are the connected components of the undirected graph whose (weighted) adjacency matrix is $\mathbf{A}$. But our matrices' graphs have a single connected component, making such a perfect partitioning impossible. However, we can still partition the graph in a way that minimizes the communication needed between parts. This corresponds to minimizing the edges crossing between parts, since graph edges denote data dependencies. Because each partition is to be run in parallel, we also want them to be approximately the same size for good load balancing.

Unfortunately, these types of optimal graph partitioning problems are NP hard. If this technique is to actually accelerate the matrix vector multiplies, we need a fast partitioning algorithm. Most of the literature on this topic gave algorithms that were too slow, so I came up with a simple breadth first search (BFS)-based algorithm to partition the graph in to P parts.

To have equally-sized partitions, we want the clusters' centers to be distributed evenly around the mesh. Algorithm 1 does a fairly good job of this. Once the centers are chosen, a BFS is used to grow the clusters around the centers. Initially a single BFS was used with the P centers as

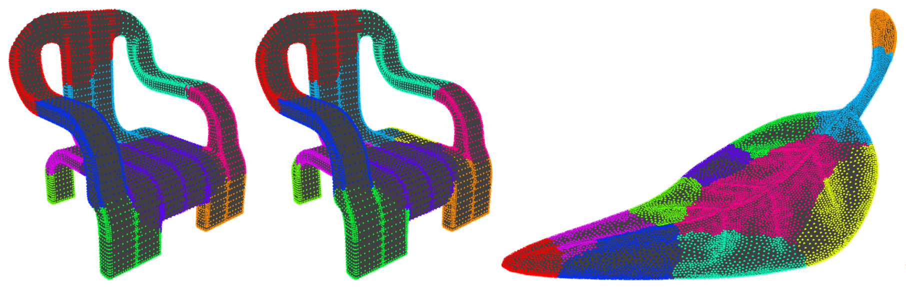


Figure 5 *Vertex partitioning ($P = 11$) visualized by hue: the red vertices belong to the partition with arbitrarily chosen center vertex 0. The "equalized partitioning" is on the right for both pairs. Notice the huge pink cluster in the middle of the leaf is shrunk to a more reasonable size by the equalizing algorithm.*

start vertices, but this lead to high cluster size variance. To address this problem, the "equalized partitioning" variant concurrently runs P separate BFSs, executing a single step of each search in a round-robin schedule. Under "equalized partitioning," each cluster grows at the same rate, and is more likely to reach the same size before running into its neighbors.

Algorithm 1 ChooseCenters(P)

```
1: Choose vertex 0 as a center (arbitrary)
2: for $i = 2$ to $P$ do
3:     Run a BFS to compute each vertex's edge count distance from *any* existing center
4:     Pick the vertex with highest distance as a new center.
5: end for
```

The graph induced by a stiffness matrix is closely tied to the mesh itself. Each mesh vertex spawns three matrix graph vertices (the graph vertices for the x, y, and z displacement variables of that mesh vertex). Edges between these graph vertex triplets correspond to mesh edges. This connection allows us to visualize the graph partitioning as a mesh partitioning and enables us to visually assess the quality of the partition (Figure 5).

3.2.4 Partitioned MatVec

The matrix partitioning can be used to implement a parallel MatVec. Let P_i be the set of indices of all n_i vertices in partition i. We re-index each global index $v \in P_i$ with a bijective mapping to local indices $l_i : P_i \to \{0, \ldots, n_i - 1\}$. The inverse mapping (local to global indices) is also constructed: $g_i : \{0, \ldots, n_i - 1\} \to P_i$.

Then, a symmetric $n_i \times n_i$ local sub-matrix is built from triplets in $\mathbf{A}$ that correspond to edges within the partition:

$$\mathbf{A}_i = \{(l_i(j), l_i(k), v) : (j, k, v) \in \mathbf{A}, j \in P_i, k \in P_i\},$$

and a local sub-vector is built from $\mathbf{x}$:

$$\mathbf{x}_i = \begin{pmatrix} \mathbf{x}\left[g_i\left(P_i[0]\right)\right] \\ \vdots \\ \mathbf{x}\left[g_i\left(P_i\left[n_i - 1\right]\right)\right] \end{pmatrix}.$$

If there were no edges crossing between partitions, one could just compute each $\mathbf{b}_i = \mathbf{A}_i\mathbf{x}_i$ subproblem in parallel and assemble the result using g_i. But, if such edges exist, then their endpoints (vertices along the partition boundaries) will be missing the contributions from their neighbors in the opposite partition.

The solution is, for each vertex $v_i \in P_i$ connected to vertex $v_j \in P_j$, to create "ghost vertices" v_i' in part P_j and v_j' in part P_i. These ghost vertices accumulate the contribution of P_j's vertices to v_i and P_i's vertices to v_j, respectively.

P_i's augmented local sub-vectors (containing ghost vertex variables) and sub-matrix (containing entries for cross-partition edges) are called $\tilde{\mathbf{x}}_i$ and $\tilde{\mathbf{A}}_i$. Now the parallel partitioned multiply consists of computing $\tilde{\mathbf{b}}_i = \tilde{\mathbf{A}}_i\tilde{\mathbf{x}}_i$, exchanging/accumulating ghost vertex values, and finally assembling the result. If $\mathbf{A}$ is to be repeatedly applied, result assembly only needs to happen at the very end; the intermediate results can remain distributed among parts/processors. For this reason, the timing will not count the reassembly cost, though this is cheap anyway.

Notice that any implementation can be used to compute $\tilde{\mathbf{b}}_i = \tilde{\mathbf{A}}_i\tilde{\mathbf{x}}_i$, allowing this subproblem to be optimized independently. The next section tries multiple variants.

3.2.5 OpenMP Results

I implemented the Partitioned MatVec algorithm with OpenMP, using each of the serial methods discussed in Section 3.2.1 to solve the subproblems. The code can be found in `../matvec/CPU/matmul_partition_omp.cc`. The results are shown in Figure 6, and are much better than for the naïve implementation.

The substantial performance drop-off between using 10 and 12 unequalized partitions on the leaf model demonstrates the importance of using the equalizing algorithm one. This same performance drop-off is seen in the MPI results of next section (Figure 8). The drop-off occurs when the partitioner creates an 11th part that is much larger than the previous ones, giving a single abnormally large subproblem that ruins load balancing. This cluster is the pink one mentioned in Figure 5. The size of the subproblems can be gauged by the sub-matrix nonzero counts, which are plot-

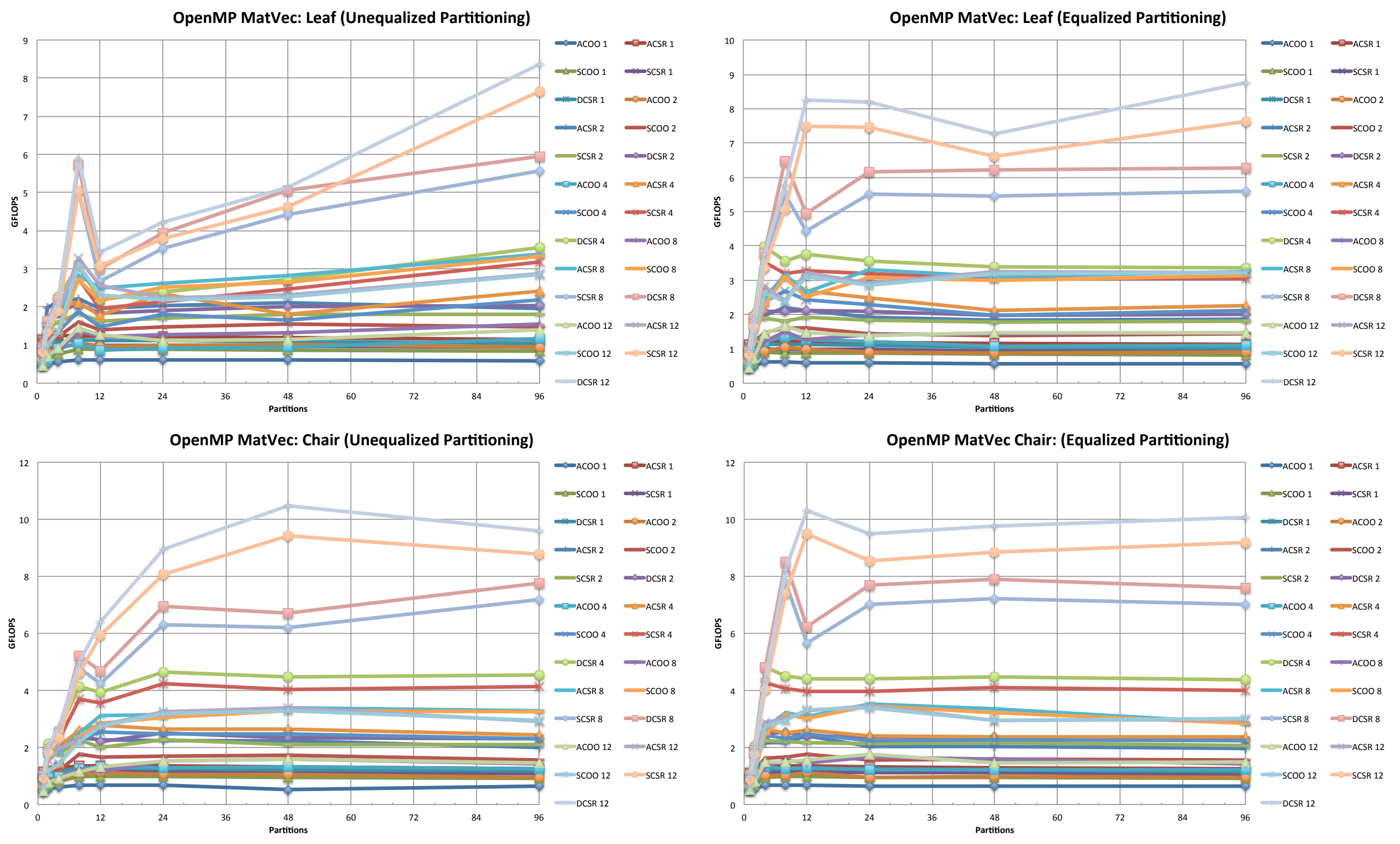


Figure 6 *OpenMP Partitioned Parallel Sparse MatVec performance. Each line plots performance for a combination of method (ACOO, ACSR, SCOO, SCSR, DCSR) and thread count (1 . . . 12) vs. the number of partitions used. All results were gathered on a* 12 *core Intel Xeon X5650 @ 2.67GHz.*

ted in Figure 7 for the 11-part partitions of the leaf model. Notice that equalizing does in fact reduce the variance of problem size.

The Diagonal + Symmetric CSR format is the clear winner, and with equalized partitioning, there is no benefit for using more partitions than threads. The peak performance is 10.463 GFLOPS for the chair model's stiffness matrix, nearly reaching CUSP's performance of 11.326 GFLOPS.

Another noteworthy comparison is that running on my laptop (Dual Core, Core i5 520M @ 2.4GHz with hyperthreading), the 4 thread OpenMP DCSR implementation achieves 1.750 GFLOPS for the chair matrix, beating CUSPS' 1.632 GFLOPS (Table 3).

3.2.6 MPI Results

To scale past 12 cores and to take advantage of multiple independent memory/cache systems, I wrote an MPI implementation using the Diagonal + Symmetric CSR format to solve the subproblems. This implementation is found in `matvec/CPU/matmul_partition.cc`.

In this implementation, one partition is created per processor (since no real performance gain was seen with more partitions in the OpenMP implementation). The root processor partitions the matrix and distributes the subproblems to the other processors. It also tells each processor which ghost variables it must exchange with which other processors. Blocking `MPI_SEND` and `MPI_RECV` calls are used to exchange the ghost variables; because the exchanges are symmetric, is not hard to devise an ordering such that these operations will never deadlock.

I benchmarked my code on two nodes of a supercomputer. Because communication is minimized by the partitioning, performance was still good even when the job spilled across nodes. These results are shown in Figure 8. The efficiency does fall as processors are added (as expected given the rising communication overhead and extra work for ghost vertices), but it remains around 50–60% when equalized partitioning is used.

The peak performance is 16.869 GFLOPS for the chair model's stiffness matrix, easily surpassing CUSP's performance.

4 Stress Tensor Eigenvalues

We turn now to the other target for optimization, finding the eigenvalues of potentially millions of 3×3 matrices (recall

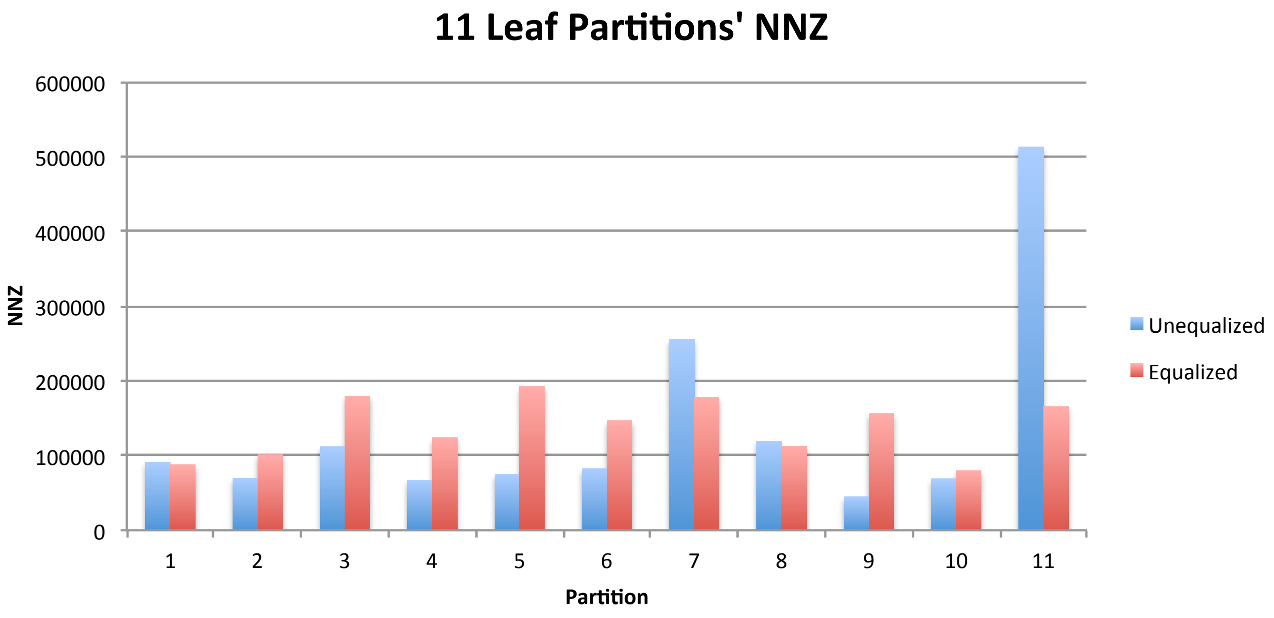


Figure 7 *The parts' sub-matrix nonzero counts with and without equalization.*

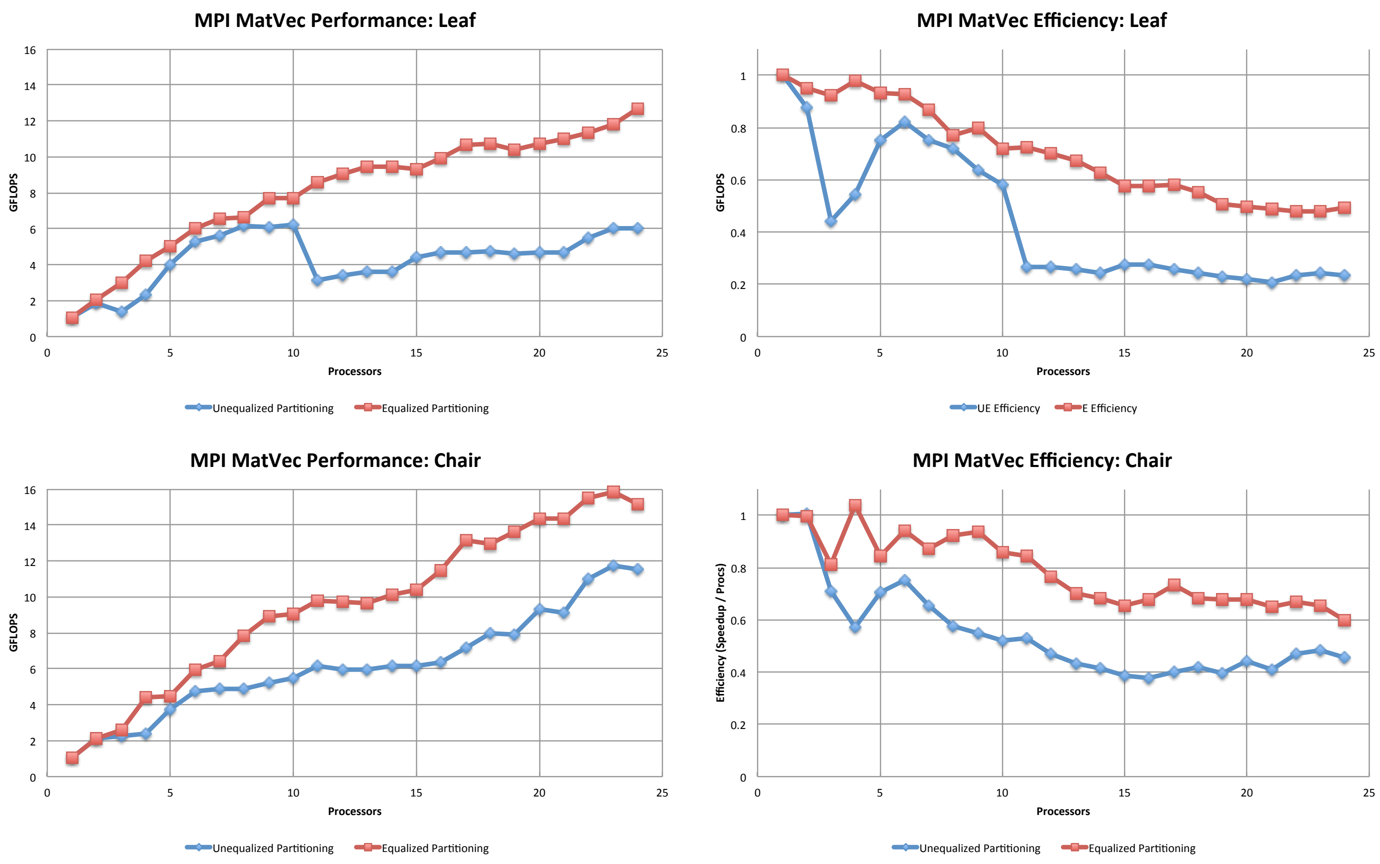


Figure 8 *MPI Partitioned Parallel Sparse MatVec performance and efficiency. These results were gathered on two nodes of a cluster, each with a* 12 *core Intel Xeon X5650 @ 2.67GHz.*

Core i5 520M @ 2.4 GHz

N	OMP Time	SSE Time	SSE Speedup	3x3 Time (OMP)	3x3 Time (SSE)	Memory (MB)
8000	0.001909	0.000721	2.647711512	238.625	90.125	0.192
80000	0.011529	0.003915	2.944827586	144.1125	48.9375	1.92
800000	0.104563	0.038956	2.684130814	130.70375	48.695	19.2
8000000	0.917951	0.372261	2.465880122	114.743875	46.532625	192
80000000	9.387443	3.895451	2.409847538	117.3430375	48.6931375	1920

Core i7 3930K @ 3.2 GHz

N	OMP Time	SSE Time	SSE Speedup	3x3 Time (OMP)	3x3 Time (SSE)	Memory (MB)
8000	0.000318	0.000115	2.765217391	39.75	14.375	0.192
80000	0.002231	0.000652	3.421779141	27.8875	8.15	1.92
800000	0.019827	0.004452	4.453504043	24.78375	5.565	19.2
8000000	0.193981	0.042834	4.528668814	24.247625	5.35425	192
80000000	1.937165	0.431384	4.490581477	24.2145625	5.3923	1920

Table 6 *OpenMP and OpenCL eigenvalue solver benchmarks for N symmetric 3×3 matrices on two machines. The Core i5 supports 4-wide SSE instructions, giving a 2.5–$3\times$ speedup, while the Core i7 supports, 8-wide AVX instructions, giving a 2.75–$4.5\times$ speedup.*

that the maximum-magnitude eigenvalue of the per-element stress tensor gives the maximum internal force in that element).

For this task, we use the Jacobi eigenvalue algorithm as proposed in [8]. We could easily have solved the third degree characteristic polynomial, but that approach is unstable. Jacobi's eigenvalue algorithm consists of repeatedly conjugating the matrix by orthogonal matrices, which is a provably stable operation. For 3×3 matrices, I found the Jacobi's algorithm converges to nearly within machine precision of the eigenvalues in just 4 full passes, making it extremely fast.

The basic idea behind the Jacobi eigenvalue algorithm is to iteratively conjugate by Givens rotations matrices, $\mathbf{Q}_k$, each chosen to zero out a particular off-diagonal entry:

$$\mathbf{A}_{k+1} = \mathbf{Q}_k^{-1}\mathbf{A}_k\mathbf{Q}_k = \mathbf{Q}_k^T\mathbf{A}_k\mathbf{Q}_k$$

Future iterations will reintroduce off-diagonal values that were already cancelled, but the net result of each conjugation is to reduce the off-diagonal sum of squares (causing an equal increase in the diagonal sum of squares since the total Frobenius norm is invariant to conjugation by orthogonal matrices). One can give bounds on the off-diagonal norm after k iterations (see [6]).

[8] describes how to compute approximate Givens rotations that still give provable convergence but avoid branching. This modified algorithm is ideal for a SIMD implementation: the computation path for each matrix is identical. [8] provides an implementation of their 3×3 SVD algorithm using SSE intrinsics, but these intrinsics make the code very difficult to read. I decided to implement a 3×3 eigensolver for symmetric matrices using OpenMP. Because auto-vectorization was unable to vectorize my code, I also wrote an OpenCL version using the `float8` datatype to force vectorization. The benefit of this is the same readable code can also run on the GPU if desired.

I operate on the symmetric matrices in compressed form: the lower triangle of each matrix is stored as 6 scalars. I worked out the formulas for the compressed output of the conjugation by a givens rotation matrix, and the resulting code is found in `fast_eig33/fast_eig33.c`. The optimized OpenCL code is in `fast_eig33/fast_eig33.cl`.

Performance on two machines is shown in Table 6. The Core i7 machine was able to compute a set of eigenvalues every 5.3ns (amortized) at peak performance. This is so fast that it doesn't make sense to run the code on the GPU (even though it is already implemented in OpenCL); the benchmarks reveal that the largest problem size one could fit in the memory of a modern GPU would take no more than a few seconds. The models a user might create are unlikely to even come close to this size.

5 Future Work

The glaring omission in this optimization effort is the absence of an algorithm using only a sparse MatVec to find the smallest eigenvalues. Without this algorithm, the optimised sparse MatVec cannot be used for modal analysis. I will continue to search for a method that is capable of converging to the smallest eigenvalues of my particular matrices at a feasible rate.

In the meantime, I will focus on the other bottlenecks of the pipeline. I plan to construct the stiffness matrix on the GPU—the full matrix can be built up from independent per-element stiffness matrices. It does require an efficient reduction into the full sparse matrix. Techniques for this are explored in [3]. I also plan to parallelize the stress tensor construction.

References

[1] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[2] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.

[3] Cris Cecka, Adrian J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5):640–669, 2011.

[4] James Demmel. Communication-avoiding algorithms for linear algebra and beyond. Presented at the November 9 CS Seminar at NYU, New York, NY, 2012.

[5] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[6] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[7] A. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

[8] Aleka McAdams, Andrew Selle, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations. technical report, University of Wisconsin - Madison, May 2011.

[9] P. Quillen and Q. Ye. A block inverse-free preconditioned Krylov subspace method for symmetric generalized eigenvalue problems. *Journal of Computational and Applied Mathematics*, 233:1298–1313, January 2010.

[10] H. Yang. Conjugate gradient methods for the rayleigh quotient minimization of generalized eigenvalue problems. *Computing*, 51:79–94, 1993.