CS 5220
Comparative Studies on the Computational Speed Between Skylines and Pardiso Sparse Direct Solver

Marc Aurele Gilles (mtg79)
Wenjia Gu (wg233)
Wensi Wu(382)

December 15, 2015

1 Introduction

The goal of this project was to compare the performance of two different sparse direct solvers. First, we took a look at our in-house quasi-static structural code, $3D\_geom\_nonlin\_truss.c$. The numerical implementation of the code is described in section 2. The $3D\_geom\_nonlin\_truss.c$ is a geometrically nonlinear finite element code embedded with the feature of analyzing truss structures using skyline indexing scheme and Cholesky-like factorization to solve a large system of equations. In section 4, we describe the two different input structures we used for the performance analysis. A detailed description of the skyline indexing scheme is provided in section 5.

The solver in which we compared our original code with is a MKL sparse direct solver: Pardiso. Pardiso has features of solving large symmetric and nonsymmetric linear systems of equaitons, $AX = B$, using parallel LU, LDL^T or LL^T factorization where L, U, and D are the low triangle, upper triangle and the diagonal of matrix A respectively. The solver employes parallel pivoting methods based on OpenMP directives, which result in the robust and memory-efficient performance. In section 6 we will describe how we translated the skyline indexing scheme in the original code into Compressed Sparse Row (csr) format in order to take advantage of the Pardiso solver.

After we successfully hooked Pardiso into our original code, we conducted weak and strong scaling studies for both sparse direct methods. The results can be found in section 7.

2 Numerical Method

The numerical analysis of truss problems can be written in the form:

$$F_l(u) = 0$$

u is the displacement, F is a system of n non-linear equations where n depends on the number of nodes and degrees of freedom, and l is the maximum load factor.

We solve this system of non-linear equation using an iterative method, starting at $l = 0$, and incrementally increasing l until it reaches the user specified maximum $l^\star$. Each $F_l(u) = 0$ equation is solved using the Newton Raphson iteration method. In other words, we repeatedly solve the system equation $F_l(u_{t+1}) \simeq F_l(u_t) + K_l(u_t) * u_{t+1}$ until the solution converges to a defined tolerance close to 0. $K_l(u_t)$ is the Jacobian matrix at u_t, which in this problem is the same as the Stiffness matrix. Each iteration is essentially a linear solve:

$$K_l(u_t) * u_{t+1} = -F_l(u_t)$$

.

3 Initial Profile Result

3.1 Timing

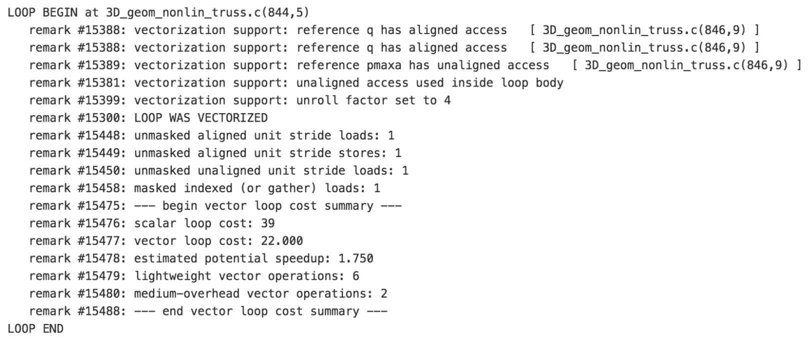
To identify the bottleneck of our original code, we ran the code with a test case, a pyramid made of 59700 nodes and 20100 elements, using amplxe. This case gave us a stiffness matrix size of 39800 * 39800. The CPU time spent in executing the orginal code is show in the table below. As shown, majority of the computation time is spent on the linear solve section due to dimension of the matrix. Although the size of stiffness matrix is very large, we noticed that it is actually very sparse. Hence, implementing a sparse solver would be our primary approach to speed up the code.

Below is the timing results of the original code:

Function	Description	CPU Time
solve	Sparse linear solve	32.560s
intel memset	allocates memory	0.273s
printf fp	prints to file	0.132s
stiff	computes stiffness matrix	0.104s
forces	computes residual forces	0.078s

3.2 Vectorizaiton

On top of introducing sparse direct solver to the original code, we thought of vectorizing the solve function to ensure that it was efficient enough. However, as shown in the vectorization report (figure 1), the solve function was already vectorize based on the report. As a result, we didn't have to do much on vectorization.



```
LOOP BEGIN at 3D_geom_nonlin_truss.c(844,5)
   remark #15388: vectorization support: reference q has aligned access   [ 3D_geom_nonlin_truss.c(846,9) ]
   remark #15388: vectorization support: reference q has aligned access   [ 3D_geom_nonlin_truss.c(846,9) ]
   remark #15389: vectorization support: reference pmaxa has unaligned access   [ 3D_geom_nonlin_truss.c(846,9) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15458: masked indexed (or gather) loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 39
   remark #15477: vector loop cost: 22.000
   remark #15478: estimated potential speedup: 1.750
   remark #15479: lightweight vector operations: 6
   remark #15480: medium-overhead vector operations: 2
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Figure 1: Vectorization report of the solve function

4 Setup: generating input structures

Since our objective is to speed up computation time and conduct scaling studies, we need to be able to generate input structures of variable sizes.

We wrote two scripts that generate two different type of input structures, a Warren truss bridge (see figure 2) , and a ”pyramid” structure (see figure 3) .

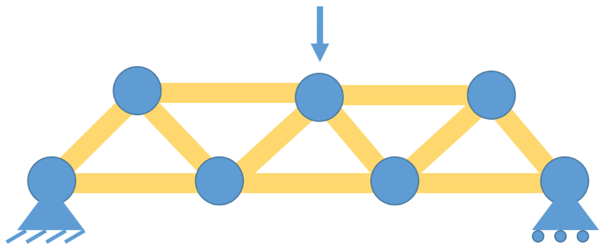


Figure 2: A Warren truss bridge of 7 elements
The blue circles are the nodes, and the yellow bars are the elements

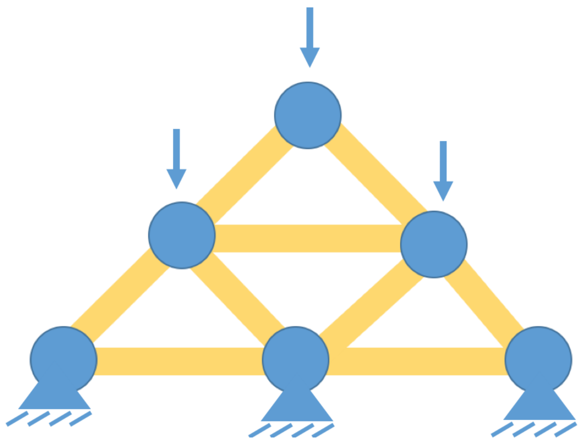


Figure 3: A pyramid structure of 6 elements
The blue circles are the nodes, and the yellow bars are the elements

For each structure we declare the position of each node, the position of each elements (as defined by a pair of nodes), the properties of each element (cross section area and the corresponding young’s modulus), and an applied load on each node. For the Warren truss bridge, the first base node was constrianted with fixed support and the last base node was constrained with roller support. We applied a load to every other node at the top of truss. For the pyramid structure, all base nodes were constrained with fixed support. We applied a load to all boundary nodes.

These two different structure give rise to significantly different running time and sparsity patterns in the stiffness matrix, for a fixed number of elements. Indeed, the truss structure produces a stiffness matrix with a small dense band, which is not the case for the pyramid structure.

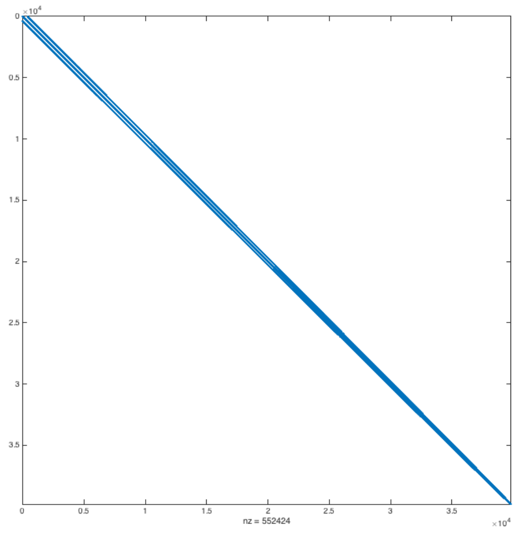


Figure 4: Sparsity structure of the stiffness matrix generated from the pyramid structure

5 Original code

The original code reads in all of the parameter of the structure, and performs numerical method described in section 2. At each iteration generates a stiffness matrix (the Jacobian), and uses a linear solve that takes advantage of the sparsity pattern of the matrix. The matrix storage format is Skyline indexing (described below). The solver takes advantage of the regular access of the skyline indexing to perform a fast Cholesky-like factorization of the form LDL^T. In addition, the original code is running in serial.

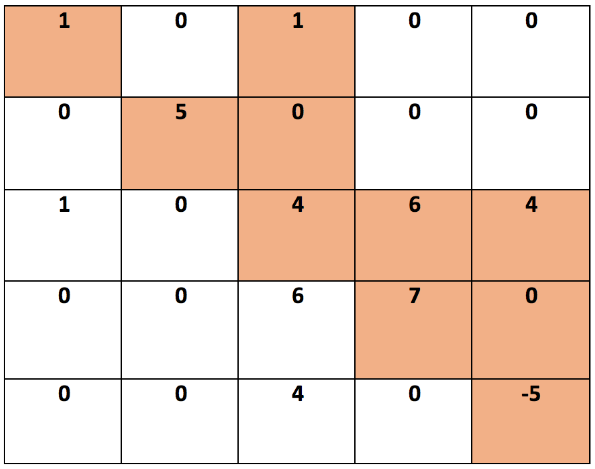
5.1 Skyline

Skyline is a sparse indexing format widely used in finite element codes for structural mechanics. Note that the term "Skyline" also refers to the set of entries from first non-zero to the diagonal in each column. A matrix in skyline format is three arrays:

1. value array: an array contains the values in the stiffness matrix between the first non-zero entry of each column and the diagonal
2. MAXA array: a pointer array stores the index of the diagonal values in the SS array; the last element is the size of the value array +1 in case of the fortran-style indexing.
3. KHT arrary: an array defines the number of entries from first non-zero value of each column to the diagonal of the stiffness matrix; the first element is default to 1.

The matrix above in skyline format would be:

$$values = [1, 5, 4, 0, 1, 7, 6, -5, 0, 6] \quad (1)$$
$$MAXA = [1, 2, 3, 6, 8, 11] \quad (2)$$
$$KHT = [1, 0, 2, 1, 2] \quad (3)$$



1	0	1	0	0
0	5	0	0	0
1	0	4	6	4
0	0	6	7	0
0	0	4	0	-5

Figure 5: Example of a matrix stored in skyline format
The entries in orange are the values stored in skyline arrays

The skyline indexing takes advantage of the fact that matrices that arise in this field are usually banded, symmetric positive definite matrices. Solving the system with such a matrix is usually (like in our code) by doing a sparse Cholesky-like decomposition. What makes the skyline indexing attractive is that the fill happening during the decomposition is only within the "skyline".

Though the skyline format is usually very efficient for small systems, it is known that the format can be less than ideal in bigger systems, where the "band" of the matrix grows large, and becomes itself sparse.

6 Optimized code

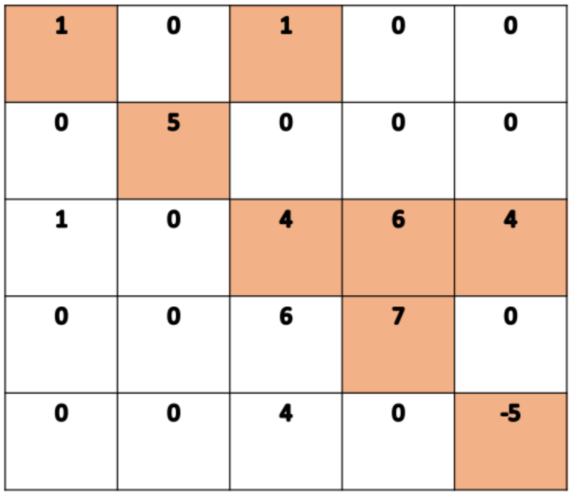
To optimize the original code, we translated the skyline format into a Compressed Sparse Row (csr) format and then used the MKL parallel sparse solver: Pardiso.

6.1 Compressed sparse row

Compressed Sparse Row (CSR)format is the most common sparse indexing format in scientific computing. A matrix in CSR format constitutes of three arrays:

1. the value array which contains the non-zero elements of the matrix
2. the column pointer array, where element i is the number of the column in A that contains the i-th value in the values array
3. the row array, where the element j of this integer array gives the index of the element in the values array that is first non-zero element in a row j of A

Note that since pardiso a symmetric solver, we only need to solve the upper triangular part of the matrix



1	0	1	0	0
0	5	0	0	0
1	0	4	6	4
0	0	6	7	0
0	0	4	0	-5

Figure 6: Example of a matrix stored in CSR format
The entries in orange are the entries saved in the symmetric CSR format

The matrix above in skyline format would be:

$$values = [1, 1, 5, 4, 6, 4, 7, -5] \tag{4}$$

$$columns = [1, 3, 2, 3, 4, 5, 4, 5] \tag{5}$$

$$rows = [1, 3, 4, 7, 8, 9] \tag{6}$$

6.2 Pardiso

6.3 Timing comparisons

Below is a figure showing the comparisons of running time between the original solver and the pardiso solver with standard settings on the truss structure.

We observe that on the truess structure, both solver are very fast (about 1 second for a structure with 30.000 elements, but the original solver is slightly better. This is likely due to the fact that the original solver is very efficient on this system generated by the structure. Indeed as observed earlier the stiffness matrix in this case have a small but dense band, which is where the solver using the skyline format excels.

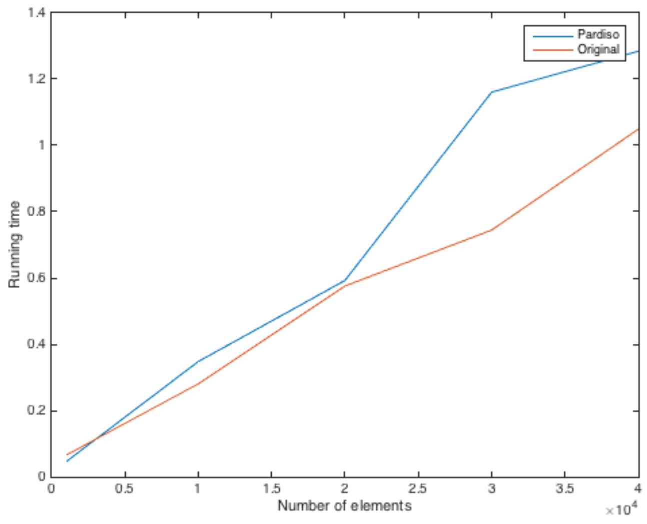


Figure 7: Comparison of running time on chain structures

Below is a figure showing the comparisons of running time between the original solver and the pardiso solver with standard settings on the pyramid structure.

Contrary to the truss structure, the pardiso solver is much faster on the pyramid structure. This is likely due to the fact that the skyline format keeps a very high number of zeros, and therefore performs a lot of unnecessary arithmetic. Indeed, for the case where we have a 40.000, over 98% of the entries saved by the skyline format are zeros.

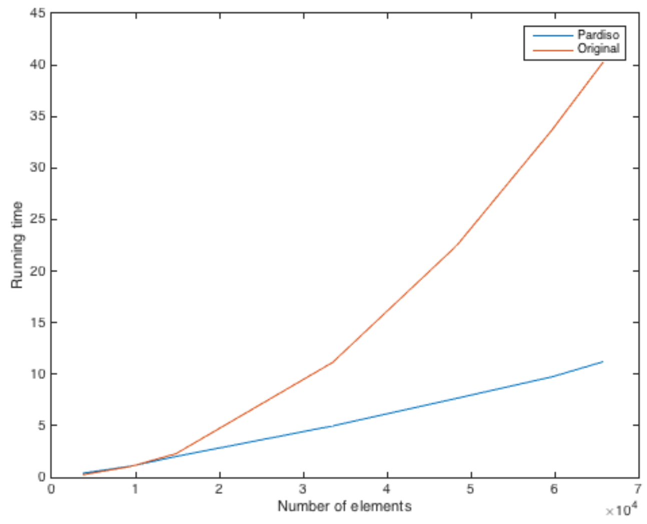


Figure 8: Comparison of running time on pyramid structures

7 Scaling studies

The base case considered here is the original finite element code embedded with skyline indexing scheme and Cholesky factorization method. We are comparing the base case with a optimize case where the skyline indexing scheme was replaced by csr and the Cholesky factorization solve was replaced by Pardiso solver. We first measured the running time of the optimaized code by using a problem size of n = 39800 where n is the size of the stiffness matrix. For weak scaling shown in figure 9, we fixed the number of processor to 1 and consider the range of 1 to 12 threads. For strong scaling shown in figure 10, we fixed the number of OpenMP thread to 1 and consider the range of processors from 1 to 12. We measured speedup as the ratio of the CPU time spent on solving the linear system of equation for the base case verse the CPU time spent on solving the linear system of equation for the optimatized case. The strong scaling of the speedup plot is shown in figure 11.

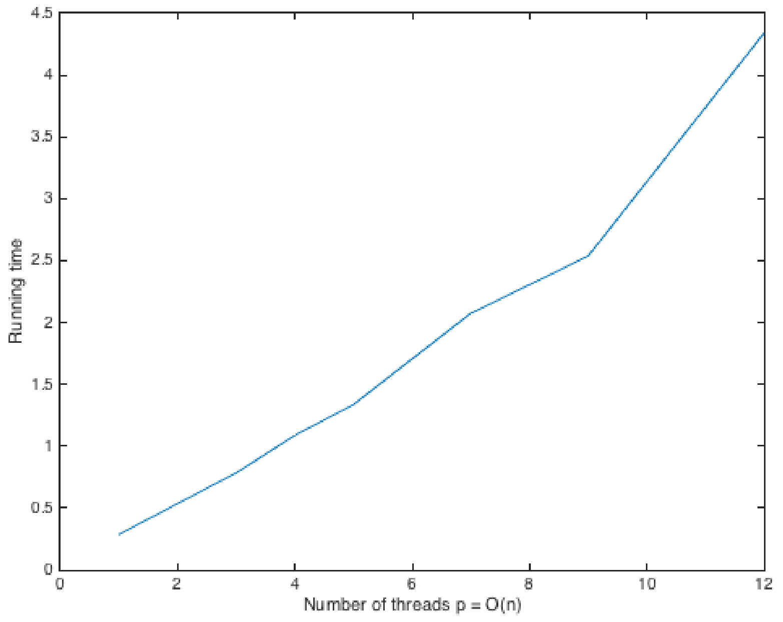


Figure 9: Fixing number of processor = 1(?), and varying the number of OpenMP threads from 1 to 12

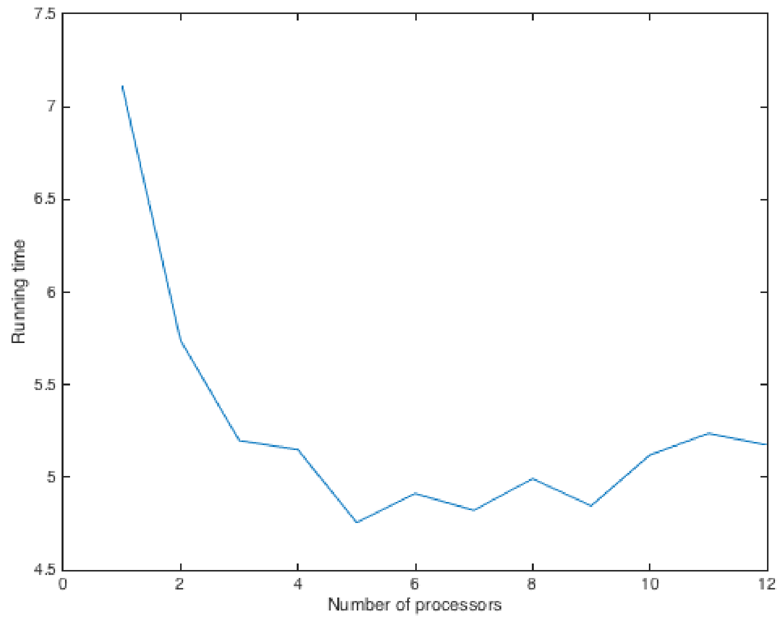


Figure 10: Fixing number of OpenMP thread = 1, and varying the number of processors from 1 to 12

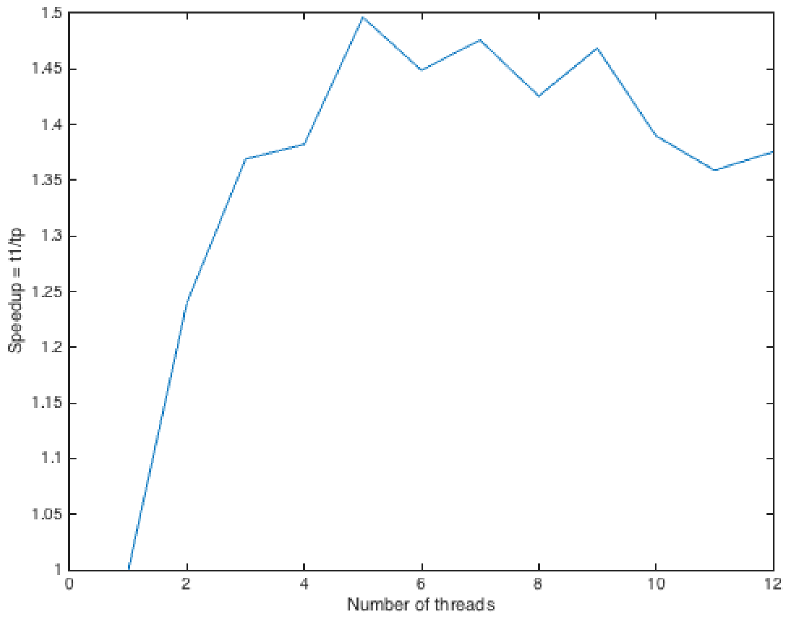


Figure 11: Speedup ratio vs. varying the number of OpenMP threads from 1 to 12

8 Conclusion