Solution of dynamic solid deformation using hybrid parallelization with MPI and OpenMP

CIMAT

MSc. Miguel Vargas-Félix

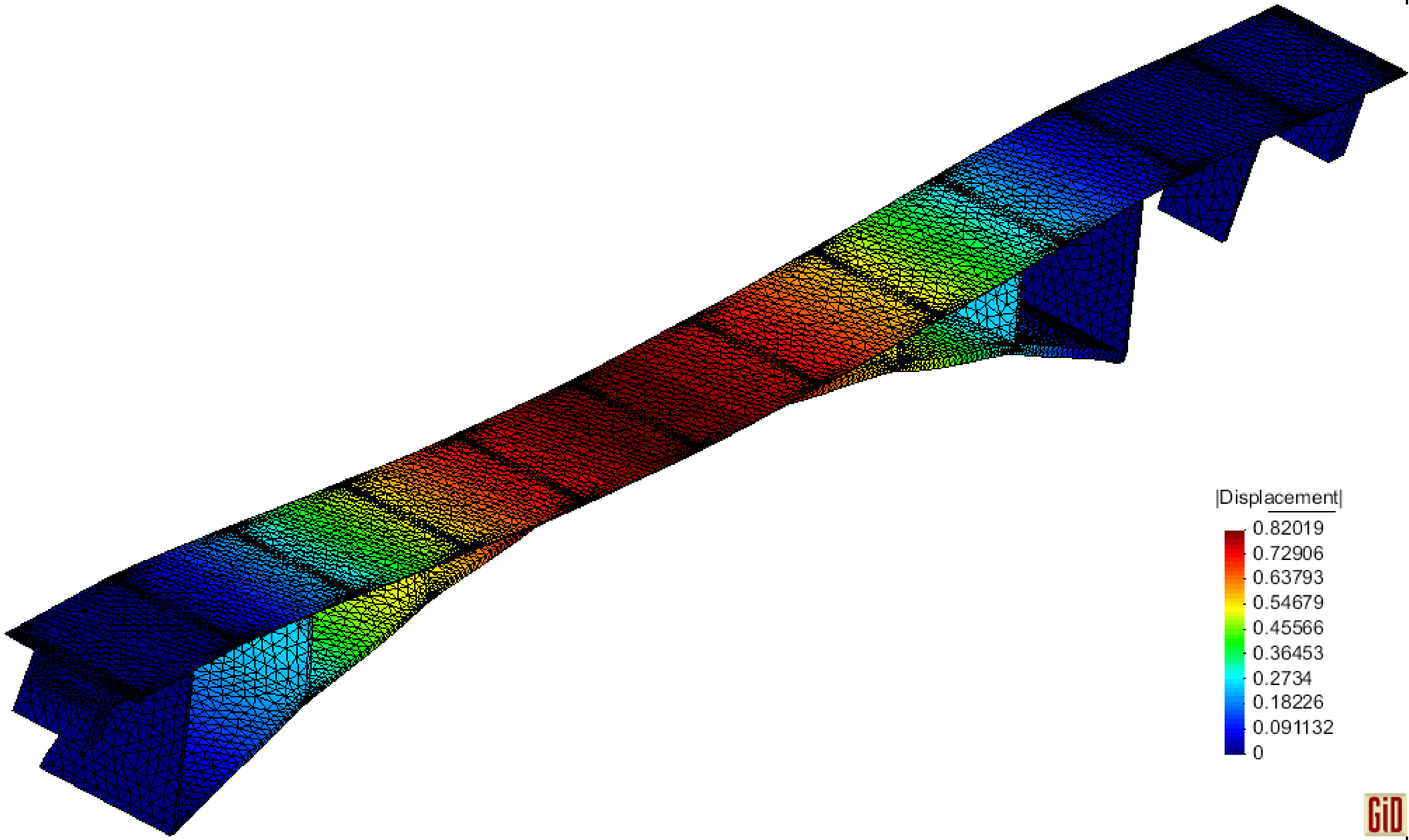
ISUM 2012

Problem description

We want to solve large scale dynamic problems with linear deformation modeled with the finite element method.

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

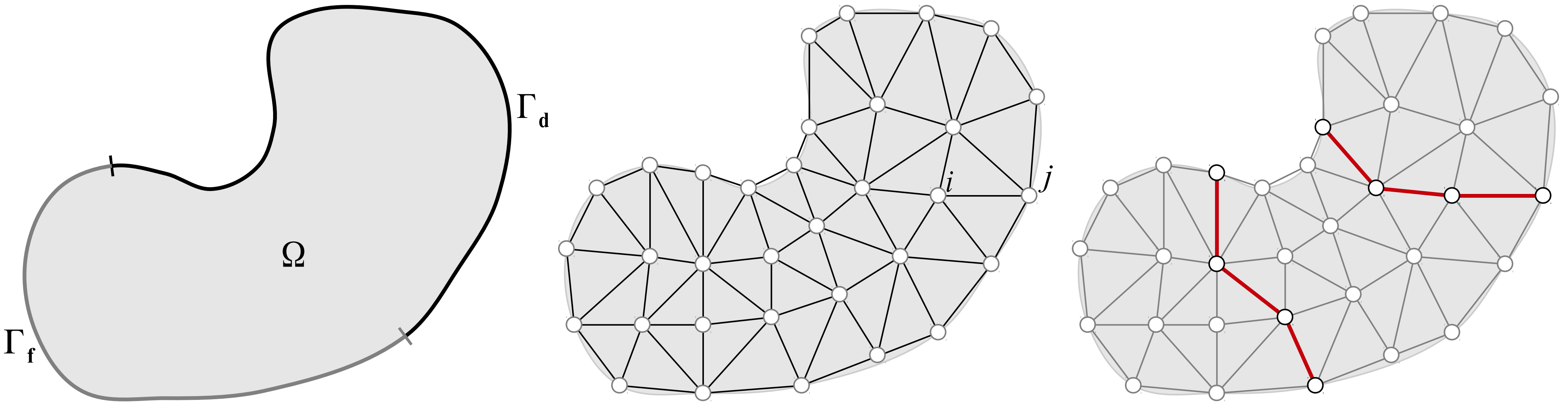
$$\boldsymbol{\sigma} = \boldsymbol{D}(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0) + \boldsymbol{\sigma}_0$$



Where $\boldsymbol{u}$ is the displacement vector, $\boldsymbol{\varepsilon}$ the stain, $\boldsymbol{\sigma}$ the stress. $\boldsymbol{D}$ is called the constitutive matrix. The solution is found using the finite element method with the Galerkin weighted residuals.

Schur substructuring method

This is a domain decomposition method without overlapping [Krui04].



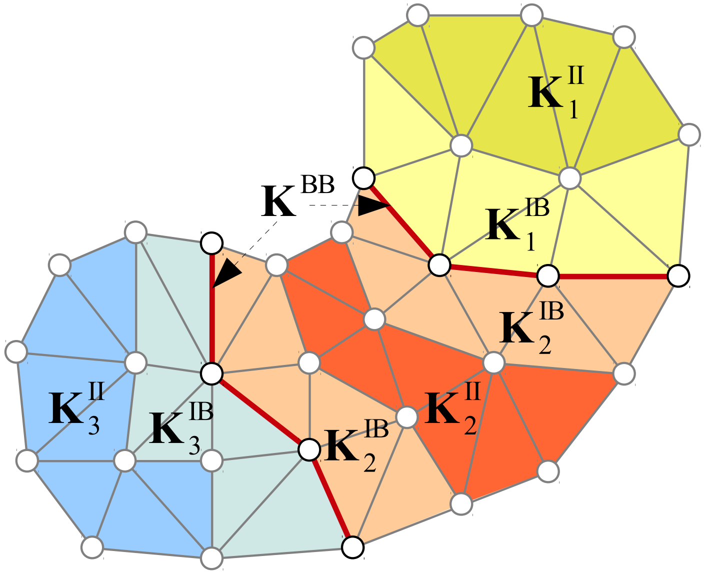
Finite element domain (left), domain discretization (center), partitioning (right).

We start with a system of equations resulting from a finite element problem

$$\mathbf{K}\,\mathbf{d} = \mathbf{f}, \tag{1}$$

where $\mathbf{K}$ is a symmetric positive definite matrix of size $n \times n$.

If we divide the geometry into p partitions, the idea is to split the workload to let each partition to be handled by a computer in the cluster.



$$\begin{pmatrix} K_1^{II} & 0 & 0 & K_1^{IB} \\ 0 & K_2^{II} & 0 & K_2^{IB} \\ 0 & 0 & K_3^{II} & K_3^{IB} \\ K_1^{BI} & K_2^{BI} & K_3^{BI} & \boxed{K^{BB}} \end{pmatrix}$$

Figure 1. Partitioning example.

We can arrange (reorder variables) of the system of equations to have the following form

$$\begin{pmatrix} \mathbf{K}_1^{II} & & \mathbf{0} & & & \mathbf{K}_1^{IB} \\ & \mathbf{K}_2^{II} & & & & \mathbf{K}_2^{IB} \\ \mathbf{0} & & \mathbf{K}_3^{II} & & & \mathbf{K}_3^{IB} \\ \vdots & & & \ddots & & \vdots \\ & & & & \mathbf{K}_p^{II} & \mathbf{K}_p^{IB} \\ \mathbf{K}_1^{BI} & \mathbf{K}_2^{BI} & \mathbf{K}_3^{BI} & \cdots & \mathbf{K}_p^{BI} & \mathbf{K}^{BB} \end{pmatrix} \begin{pmatrix} \mathbf{d}_1^I \\ \mathbf{d}_2^I \\ \mathbf{d}_3^I \\ \vdots \\ \mathbf{d}_p^I \\ \mathbf{d}^B \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1^I \\ \mathbf{f}_2^I \\ \mathbf{f}_3^I \\ \vdots \\ \mathbf{f}_p^I \\ \mathbf{f}^B \end{pmatrix}. \tag{2}$$

The superscript II denotes entries that capture the relationship between nodes inside a partition. BB is used to indicate entries in the matrix that relate nodes on the boundary. Finally IB and BI are used for entries with values dependent of nodes in the boundary and nodes inside the partition.

Thus, the system can be separated in p different systems,

$$\begin{pmatrix} \mathbf{K}_i^{\mathrm{II}} & \mathbf{K}_i^{\mathrm{IB}} \\ \mathbf{K}_i^{\mathrm{BI}} & \mathbf{K}^{\mathrm{BB}} \end{pmatrix} \begin{pmatrix} \mathbf{d}_i^{\mathrm{I}} \\ \mathbf{d}^{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_i^{\mathrm{I}} \\ \mathbf{f}^{\mathrm{B}} \end{pmatrix}, \; i=1\ldots p.$$

For each partition i the vector of unknowns $\mathbf{d}_i^{\mathrm{I}}$ as

$$\mathbf{d}_i^{\mathrm{I}} = \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \left(\mathbf{f}_i^{\mathrm{I}} - \mathbf{K}_i^{\mathrm{IB}} \mathbf{d}^{\mathrm{B}}\right). \tag{3}$$

After applying Gaussian elimination by blocks on (2), the reduced system of equations becomes

$$\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{K}_i^{\mathrm{IB}}\right) \mathbf{d}^{\mathrm{B}} = \mathbf{f}^{\mathrm{B}} - \sum_{i=1}^{p} \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{f}_i^{\mathrm{I}}. \tag{4}$$

Once the vector $\mathbf{d}^{\mathrm{B}}$ is computed using (4), we can calculate the internal unknowns $\mathbf{d}_i^{\mathrm{I}}$ with (3).

It is not necessary to calculate the inverse in (4).

Let's define $\bar{\mathbf{K}}_i^{\mathrm{BB}} = \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{K}_i^{\mathrm{IB}}$, to calculate it [Sori00], we proceed column by column using an extra vector $\mathbf{t}$, and solving for $c=1\ldots n$

$$\mathbf{K}_i^{\mathrm{II}} \mathbf{t} = \left[\mathbf{K}_i^{\mathrm{IB}}\right]_c, \tag{5}$$

note that many $\left[\mathbf{K}_i^{\mathrm{IB}}\right]_c$ are null. Next we can complete $\mathbf{K}_i^{\mathrm{BB}}$ with,

$$\left[\bar{\mathbf{K}}_i^{\mathrm{BB}}\right]_c = \mathbf{K}_i^{\mathrm{BI}} \mathbf{t}.$$

Now lets define $\bar{\mathbf{f}}_i^{\mathrm{B}} = \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{f}_i^{\mathrm{I}}$, in this case only one system has to be solved

$$\mathbf{K}_i^{\mathrm{II}} \mathbf{t} = \mathbf{f}_i^{\mathrm{I}}, \tag{6}$$

and then

$$\bar{\mathbf{f}}_i^{\mathrm{B}} = \mathbf{K}_i^{\mathrm{BI}} \mathbf{t}.$$

Each $\bar{\mathbf{K}}_i^{\mathrm{BB}}$ and $\bar{\mathbf{f}}_i^{\mathrm{B}}$ holds the contribution of each partition to (4), this can be written as

$$\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \bar{\mathbf{K}}_i^{\mathrm{BB}}\right) \mathbf{d}^{\mathrm{B}} = \mathbf{f}^{\mathrm{B}} - \sum_{i=1}^{p} \bar{\mathbf{f}}_i^{\mathrm{B}}, \tag{7}$$

once (7) is solved, we can calculate the inner results of each partition using (3).

Since $\mathbf{K}_i^{\mathrm{II}}$ is sparse and has to be solved many times in (5), a efficient way to proceed is to use a Cholesky factorization of $\mathbf{K}_i^{\mathrm{II}}$. To reduce memory usage and increase speed a sparse Cholesky factorization has to be implemented, this method is explained below.

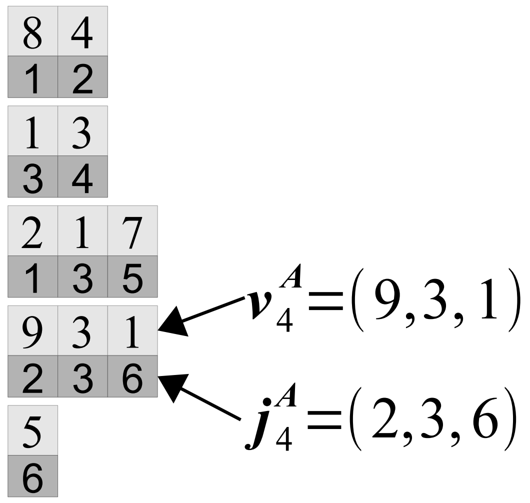
In case of (7), $\mathbf{K}^{\mathrm{BB}}$ is sparse, but $\bar{\mathbf{K}}_i^{\mathrm{BB}}$ are not. To solve this system of equations an sparse version of conjugate gradient was implemented, the matrix $\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \bar{\mathbf{K}}_i^{\mathrm{BB}}\right)$ is not assembled, but maintained distributed.

Matrix storage

An efficient method to store and operate matrices of this kind of problems is the Compressed Row Storage (CRS) [Saad03 p362]. This method is suitable when we want to access entries of each row of a matrix $\boldsymbol{A}$ sequentially.

For each row i of $\boldsymbol{A}$ we will have two vectors, a vector $\boldsymbol{v}_i^A$ that will contain the non-zero values of the row, and a vector $\boldsymbol{j}_i^A$ with their respective column indexes. For example a matrix $\boldsymbol{A}$ and its CRS representation

$$\boldsymbol{A}=\begin{pmatrix} 8 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 2 & 0 & 1 & 0 & 7 & 0 \\ 0 & 9 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix},$$



The size of the row will be denoted by $|\boldsymbol{v}_i^A|$ or by $|\boldsymbol{j}_i^A|$. Therefore the q th non zero value of the row i of $\boldsymbol{A}$ will be denoted by $(\boldsymbol{v}_i^A)_q$ and the index of this value as $(\boldsymbol{j}_i^A)_q$, with $q=1,\ldots,|\boldsymbol{v}_i^A|$.

Cholesky factorization for sparse matrices

For full matrices the computational complexity of Cholesky factorization $\mathbf{A}=\mathbf{L}\,\mathbf{L}^{\mathrm{T}}$ is $O\left(n^3\right)$.

To calculate entries of $\mathbf{L}$

$$L_{ij}=\frac{1}{L_{jj}}\left(A_{ij}-\sum_{k=1}^{j-1}L_{ik}L_{jk}\right),\ \text{for } i>j$$

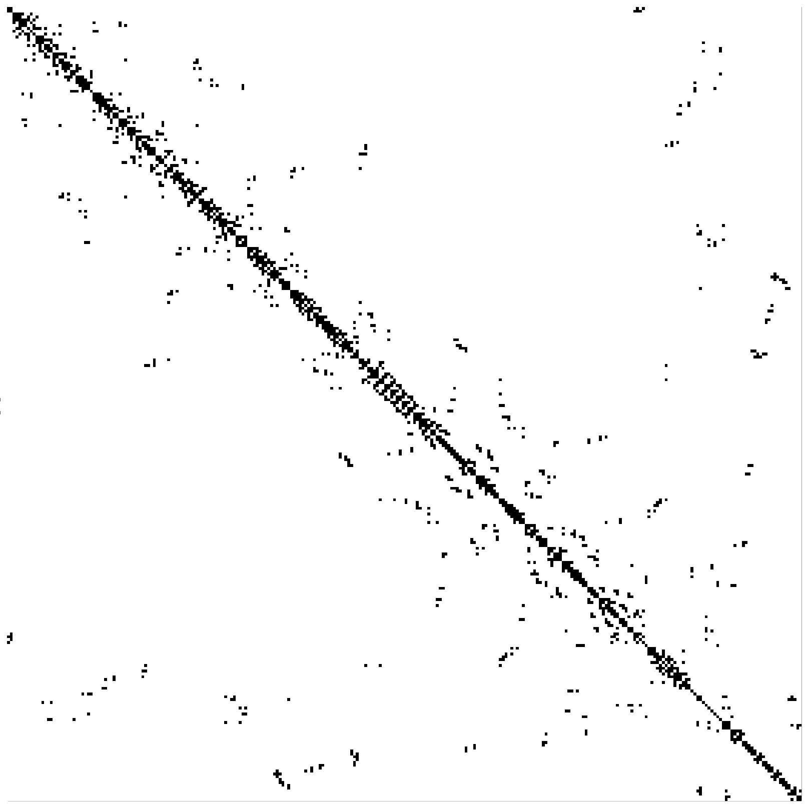
$$L_{jj}=\sqrt{A_{jj}-\sum_{k=1}^{j-1}L_{jk}^2}.$$

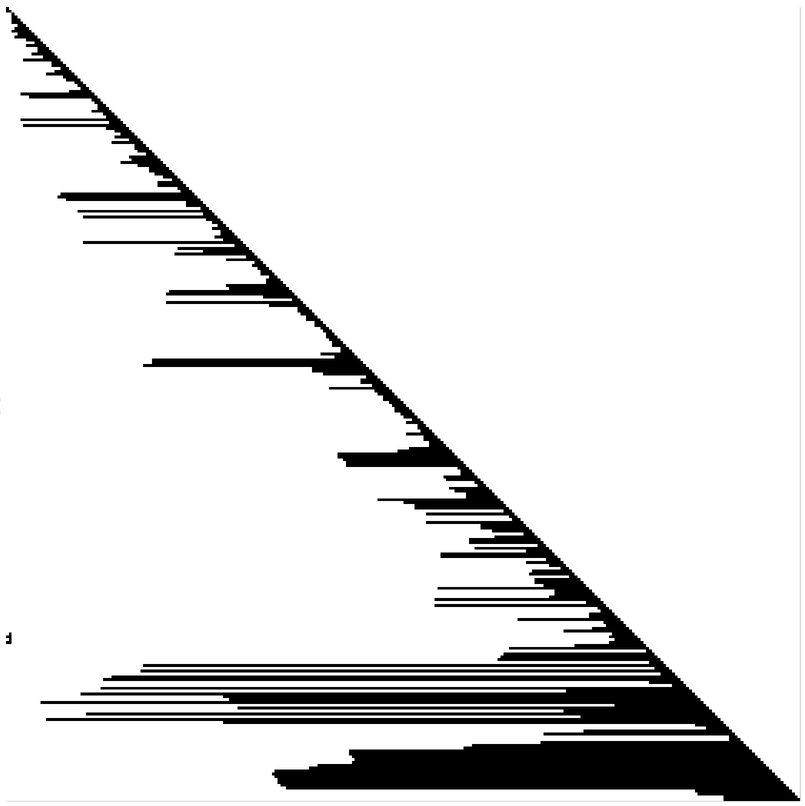
We use four strategies to reduce time and memory usage when performing this factorization on sparse matrices:

1. Reordering of rows and columns of the matrix to reduce fill-in in $\mathbf{L}$. This is equivalent to use a permutation matrix to reorder the system $\left(\boldsymbol{P}\,\boldsymbol{A}\,\boldsymbol{P}^{\mathrm{T}}\right)\left(\boldsymbol{P}\,\boldsymbol{x}\right)=\left(\boldsymbol{P}\,\boldsymbol{b}\right)$.
2. Use symbolic Cholesky factorization to obtain an exact $\mathbf{L}$ factor (non zero entries in $\mathbf{L}$).
3. Organize operations to improve cache usage.
4. Parallelize the factorization.

Matrix reordering

We want to reorder rows and columns of $\boldsymbol{A}$, in a way that the number of non-zero entries of $\boldsymbol{L}$ are reduced. $\eta(\boldsymbol{L})$ indicates the number of non-zero entries of $\boldsymbol{L}$.

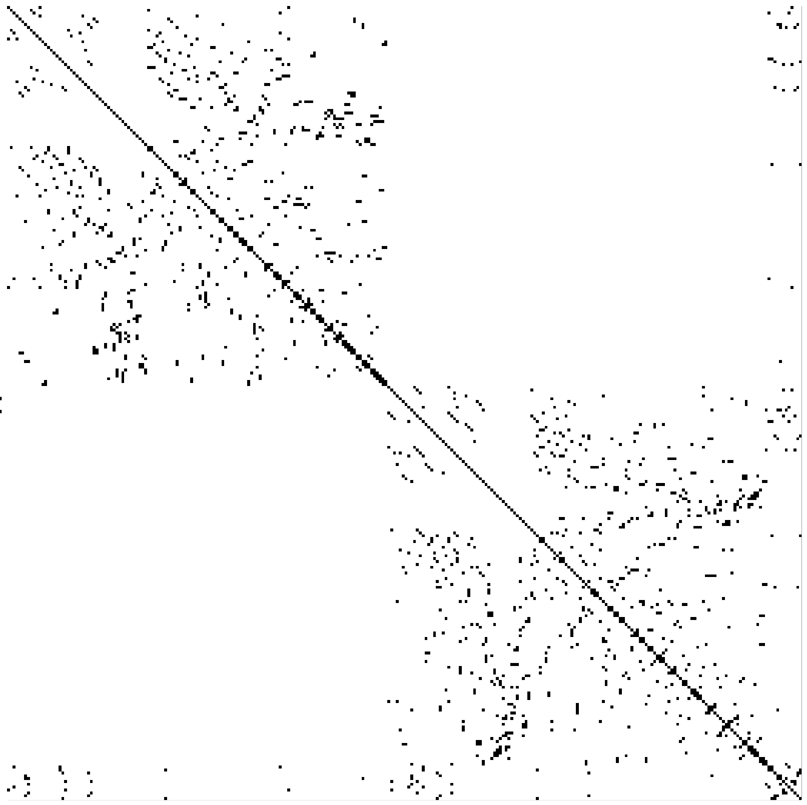
$\boldsymbol{A} =$ 

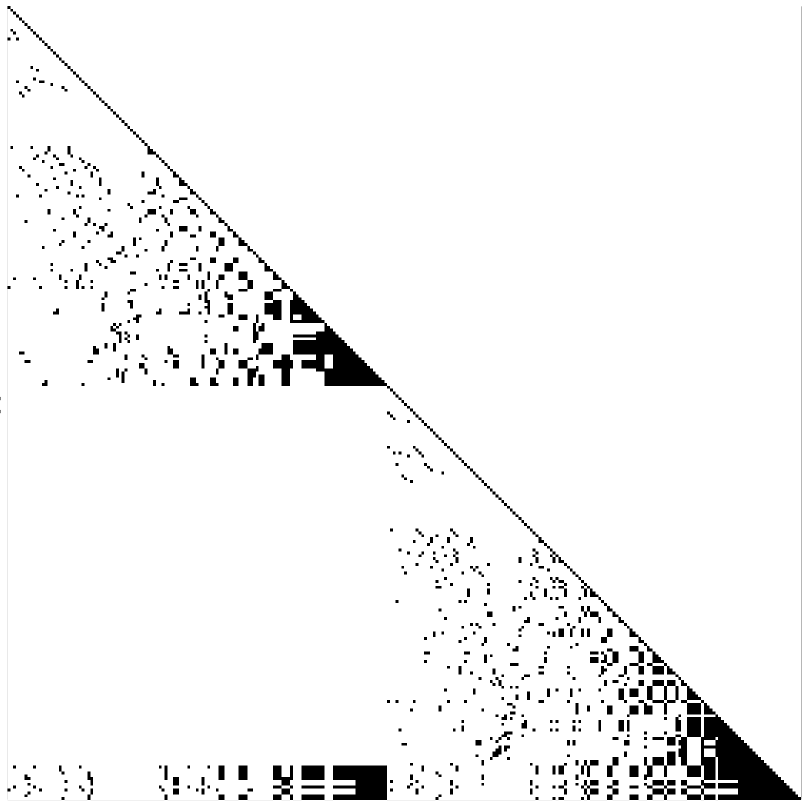
$\boldsymbol{L} =$ 

The stiffness matrix to the left $\boldsymbol{A} \in \mathbb{R}^{556\times 556}$, with $\eta(\boldsymbol{A}) = 1810$. To the right the lower triangular matrix $\boldsymbol{L}$, with $\eta(\boldsymbol{L}) = 8729$.

There are several heuristics like the minimum degree algorithm [Geor81] or a nested dissection method [Kary99].

By reordering we have a matrix $\boldsymbol{A}'$ with $\eta(\boldsymbol{A}')=1810$ and its factorization $\boldsymbol{L}'$ with $\eta(\boldsymbol{L}')=3215$. Both factorizations solve the same system of equations.

$\boldsymbol{A}'=$ 

$\boldsymbol{L}'=$ 

We reduce the factorization fill-in by

$$\frac{\eta(\boldsymbol{L}')=3215}{\eta(\boldsymbol{L})=8729}=0.368.$$

To determine a “good” reordering for a matrix $\boldsymbol{A}$ that minimize the fill-in of $\boldsymbol{L}$ is an NP complete problem [Yann81].

Symbolic Cholesky factorization

The algorithm to determine the L_{ij} entries that area non-zero is called symbolic Cholesky factorization [Gall90].

Let be, for all columns $j=1\dots n$,

$$\boldsymbol{a}_j \overset{\text{def}}{=} \left\{k>j \;\middle|\; A_{kj}\neq 0\right\},$$
$$\boldsymbol{l}_j \overset{\text{def}}{=} \left\{k>j \;\middle|\; L_{kj}\neq 0\right\}.$$

The sets $\boldsymbol{r}_j$ will register the columns of $\boldsymbol{L}$ which structure will affect the column j of $\boldsymbol{L}$.

```
$\boldsymbol{r}_j \leftarrow \emptyset,\ j \leftarrow 1\dots n$
for $j \leftarrow 1\dots n$
    $\boldsymbol{l}_j \leftarrow \boldsymbol{a}_j$
    for $i\in\boldsymbol{r}_j$
        $\boldsymbol{l}_j \leftarrow \boldsymbol{l}_j\cup\boldsymbol{l}_i\setminus\{j\}$
    end_for
    $p \leftarrow \begin{cases}\min\{i\in\boldsymbol{l}_j\} & \text{if } \boldsymbol{l}_j\neq\emptyset\\ j & \text{other}\end{cases}$
    $\boldsymbol{r}_p \leftarrow \boldsymbol{r}_p\cup\{j\}$
end_for
```

$$\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & & & & a_{16}\\ a_{21} & a_{22} & a_{23} & a_{24} & & \\ & a_{32} & a_{33} & & a_{35} & \\ & a_{42} & & a_{44} & & \\ & & a_{53} & & a_{55} & a_{56}\\ a_{61} & & & & a_{65} & a_{66}\end{pmatrix}
\quad
\boldsymbol{L}=\begin{pmatrix} l_{11} & & & & & \\ l_{21} & l_{22} & & & & \\ & l_{32} & l_{33} & & & \\ & l_{42} & l_{43} & l_{44} & & \\ & & l_{53} & l_{54} & l_{55} & \\ l_{61} & l_{62} & l_{63} & l_{64} & l_{65} & l_{66}\end{pmatrix}$$

$$\boldsymbol{a}_2=\{3,4\} \qquad \boldsymbol{l}_2=\{3,4,6\}$$

This algorithm is very efficient, its complexity in time and space has an order of $O(\eta(\boldsymbol{L}))$.

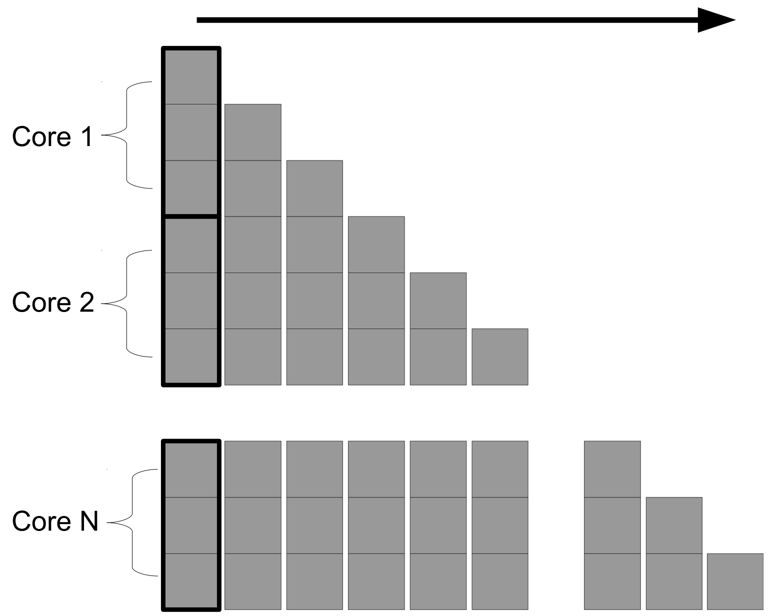
Parallelization of the factorization

The calculation of the non-zero L_{ij} entries can be done in parallel if we fill $\boldsymbol{L}$ column by column [Heat91].

Let $J(i)$ be the indexes of the non-zero values of the row i of $\boldsymbol{L}$. Formulae to calculate L_{ij} are:

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{\substack{k \in (J(i) \cap J(j)) \\ k < j}} L_{ik} L_{jk} \right), \text{ para } i > j$$

$$L_{jj} = \sqrt{A_{jj} - \sum_{\substack{k \in J(j) \\ k < j}} L_{jk}^2}.$$

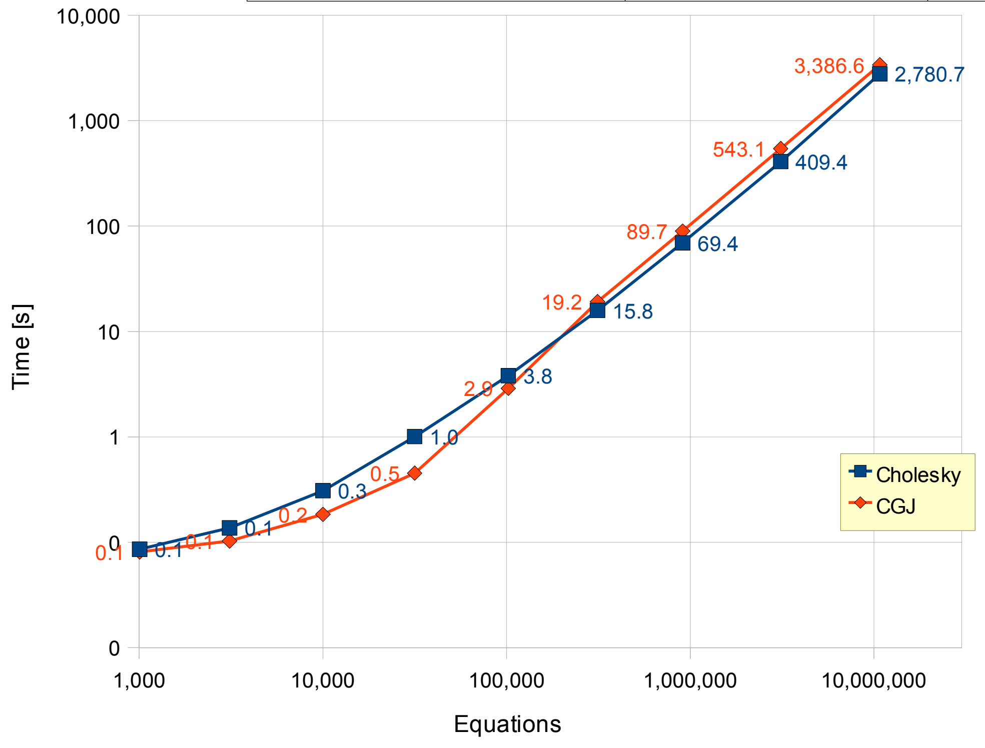


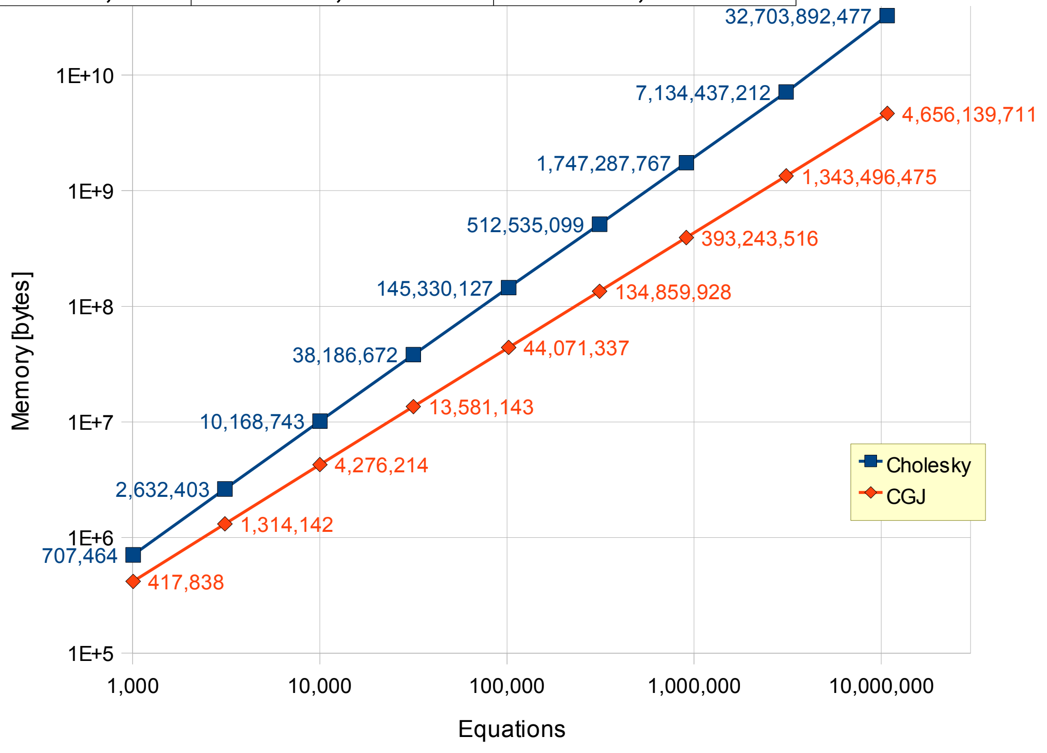
The paralellization was made using the OpenMP schema.

How efficient is it?

The next table shows results solving a 2D Poisson equation problem, comparing Cholesky and conjugate gradient with Jacobi preconditioning. Several discretizations where used.

Equations	nnz(A)	nnz(L)	Cholesky [s]	CGJ [s]
1,006	6,140	14,722	0.086	0.081
3,110	20,112	62,363	0.137	0.103
10,014	67,052	265,566	0.309	0.184
31,615	215,807	1’059,714	1.008	0.454
102,233	705,689	4’162,084	3.810	2.891
312,248	2’168,286	14’697,188	15.819	19.165
909,540	6’336,942	48’748,327	69.353	89.660
3’105,275	21’681,667	188’982,798	409.365	543.110
10’757,887	75’202,303	743’643,820	2,780.734	3,386.609

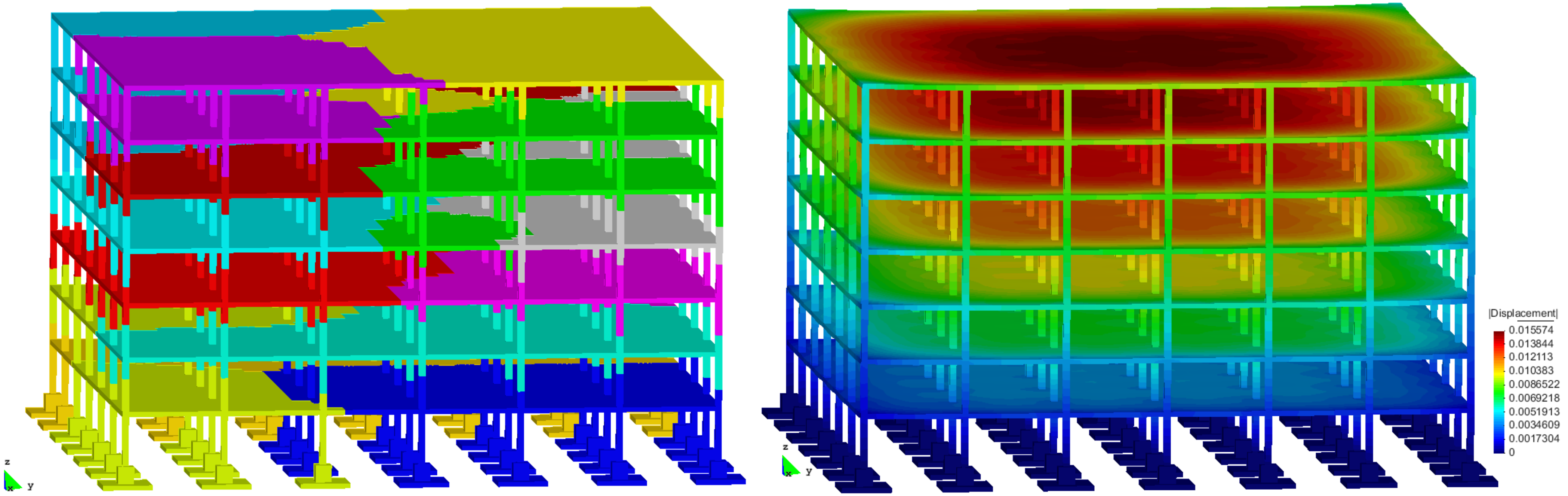




Numerical experiment, building deformation

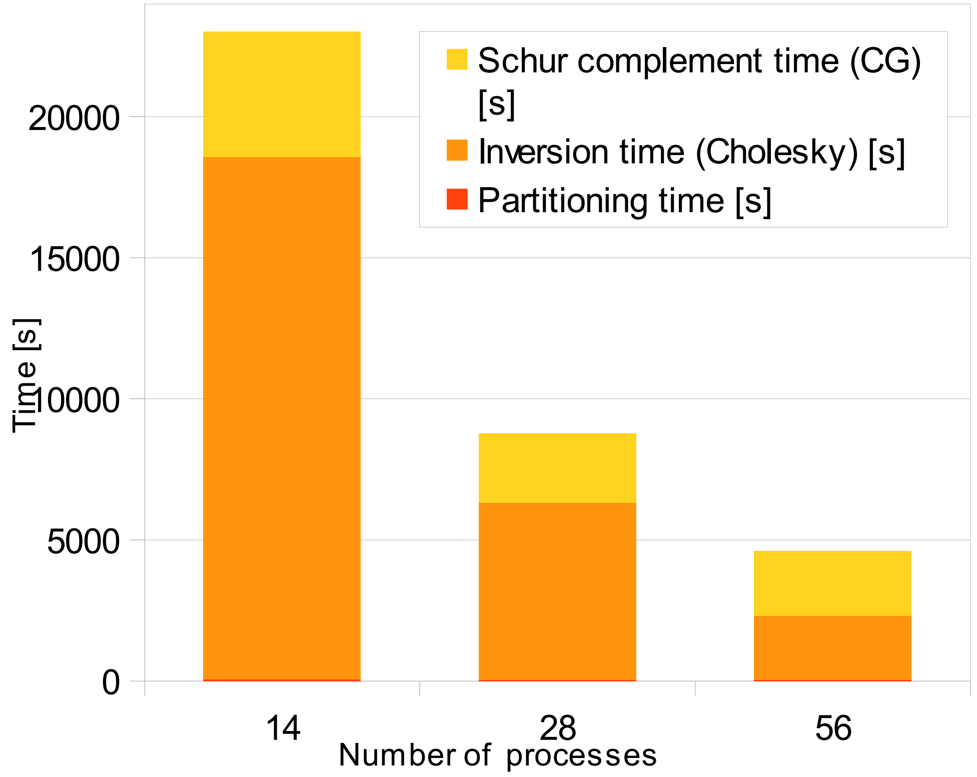
We useda cluster with 15 nodes, each one with two dual core Intel Xeon E5502 (1.87GHz) processors, a total of 60 cores.

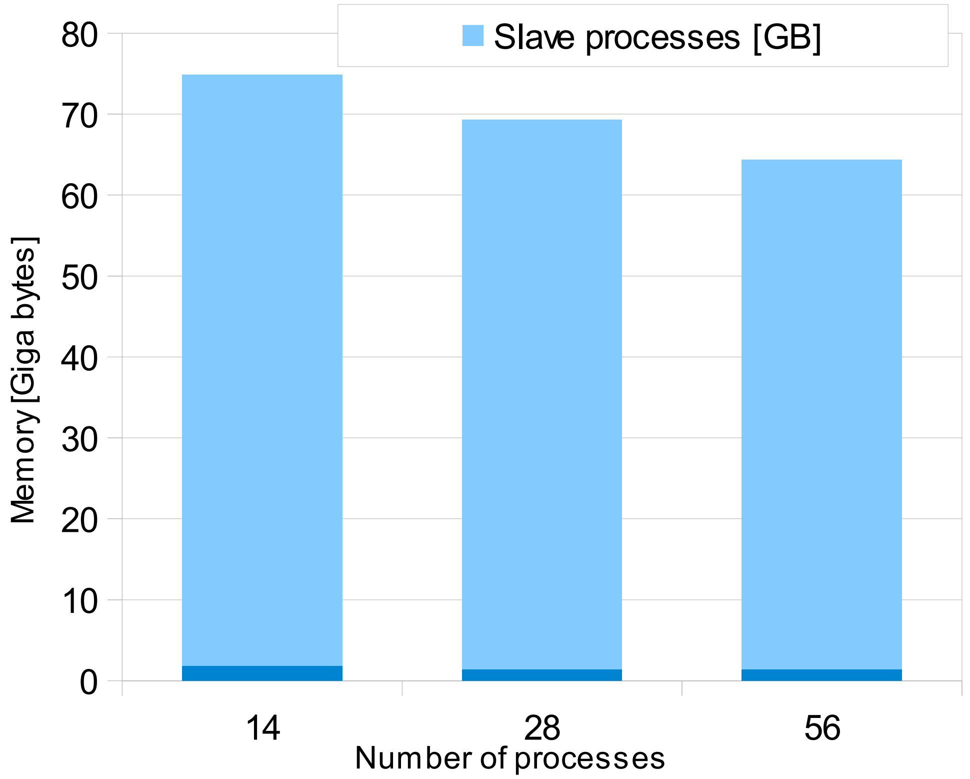
The problem tested is a 3D solid model of a building that is deformed due to self weight. The geometry is divided in 1’336,832 elements, with 1’708,273 nodes, with three degrees of freedom per node the resulting system of equations has 5’124,819 unknowns. Tolerance used is 1x10^{-10}.



Substructuration of the domain (left) resulting deformation (right)

Number of processes	Partitioning time [s]	Inversion time (Cholesky) [s]	Schur complement time (CG) [s]	CG steps	Total time [s]
14	47.6	18520.8	4444.5	6927	23025.0
28	45.7	6269.5	2444.5	8119	8771.6
56	44.1	2257.1	2296.3	9627	4608.9

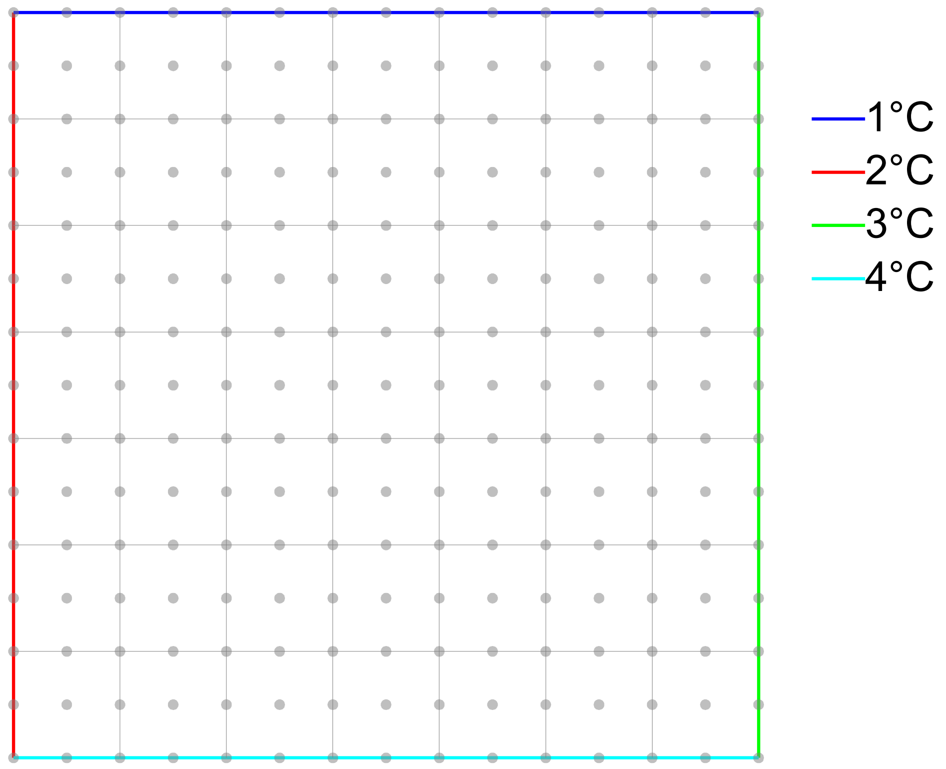


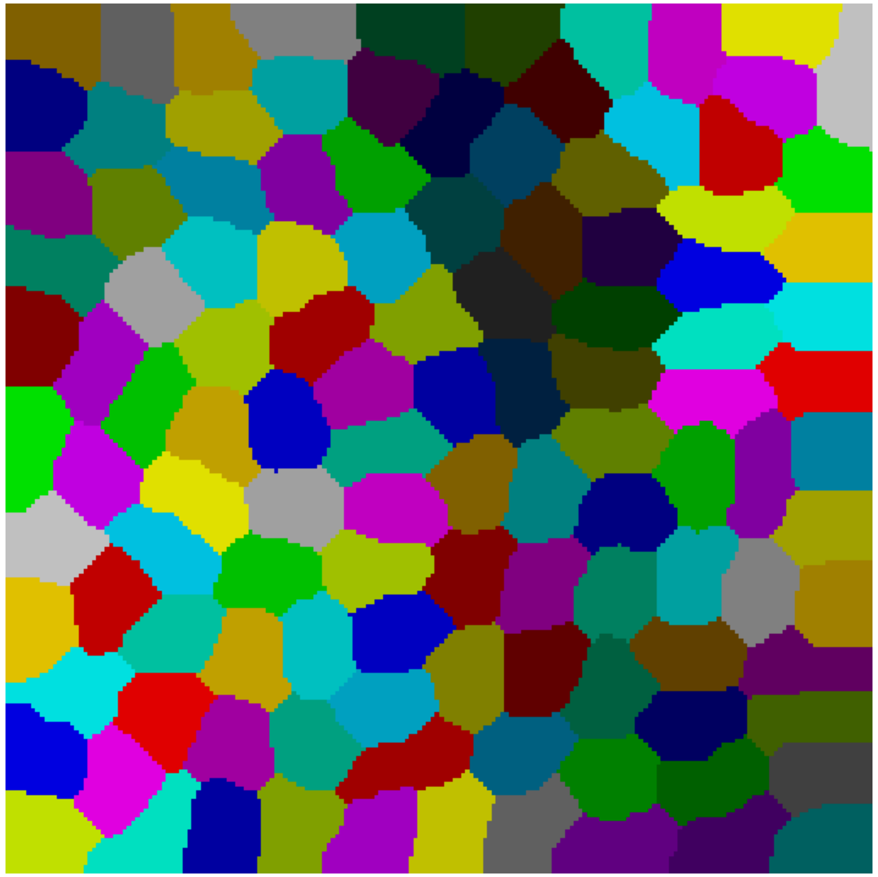


Number of processes	Master process [GB]	Slave processes [GB]	Total memory [GB]
14	1.89	73.00	74.89
28	1.43	67.88	69.32
56	1.43	62.97	64.41

Larger systems of equations

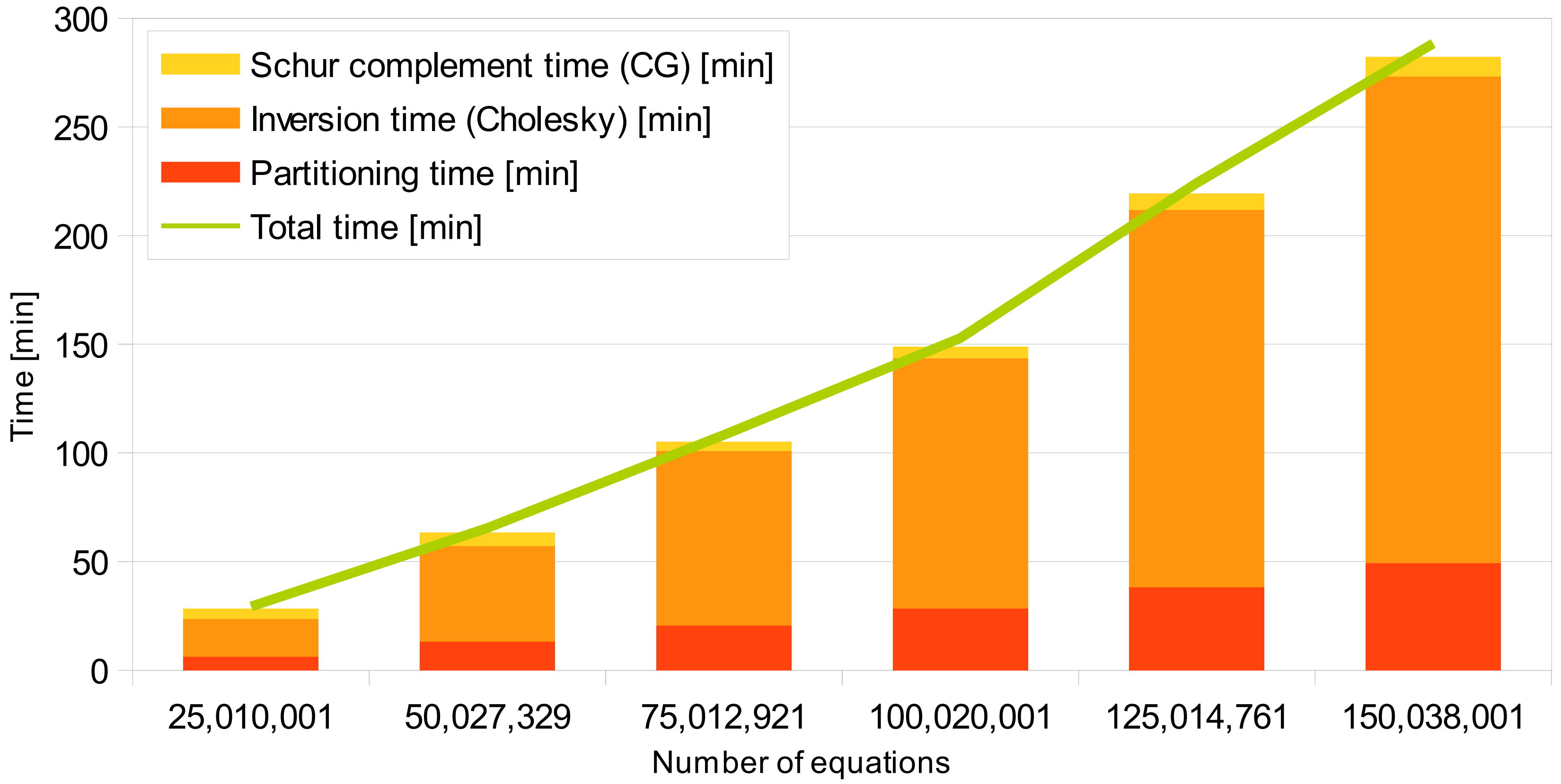
To test solution times in larger systems of equations we set a simple geometry. We calculated the temperature distribution of a metalic square with Dirichlet conditions on all boundaries.



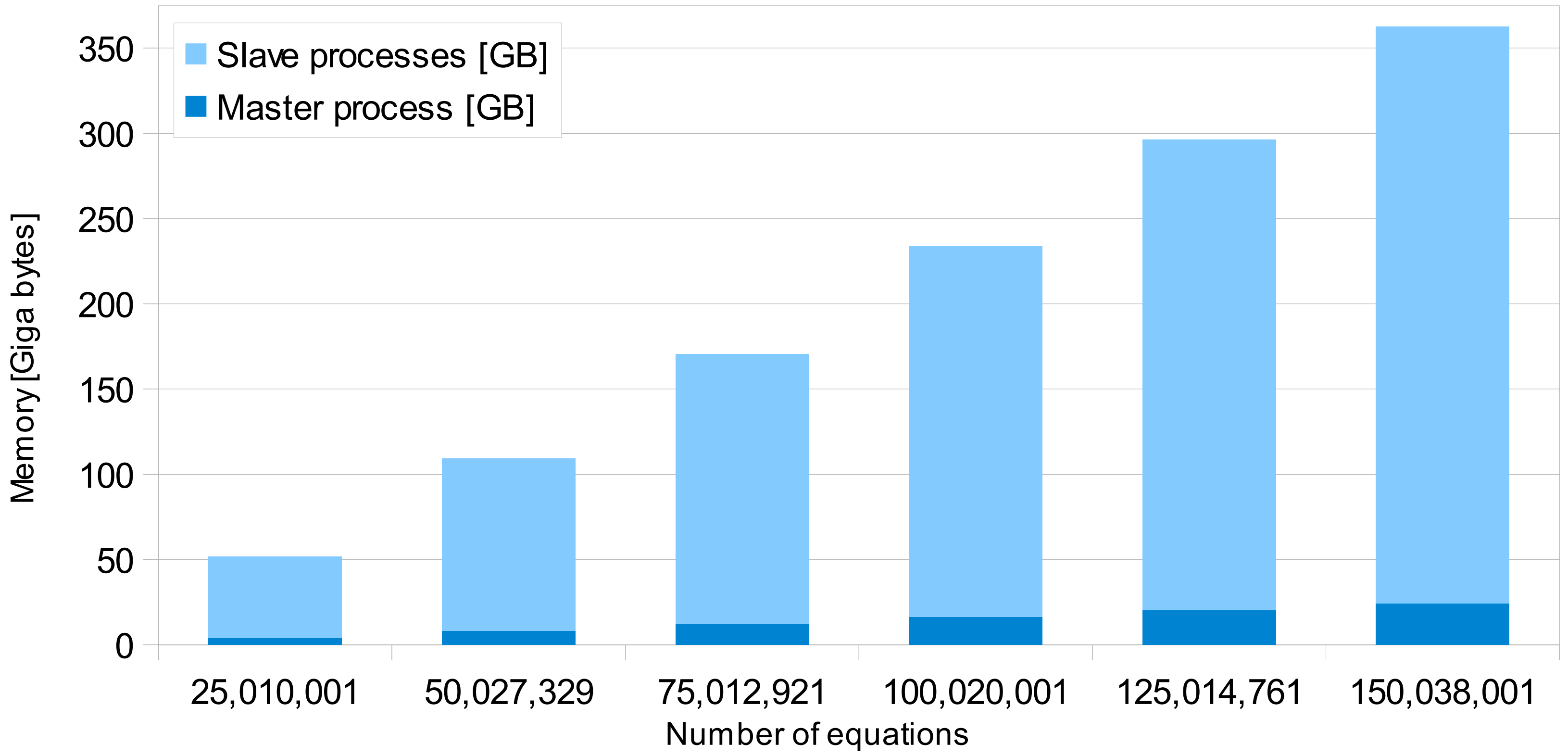


The domain was discretized using quadrilaterals with nine nodes, the discretization made was from 25 million nodes up to 150 million nodes.

In all cases we divided the domain into 116 partitions, each partition is solved in one core.



Equations	Partitioning time [min]	Inversion time (Cholesky) [min]	Schur complement time (CG) [min]	CG steps	Total time [min]
25,010,001	6.2	17.3	4.7	872.0	29.4
50,027,329	13.3	43.7	6.3	1012.0	65.4
75,012,921	20.6	80.2	4.3	1136.0	108.3
100,020,001	28.5	115.1	5.4	1225.0	152.9
125,014,761	38.3	173.5	7.5	1329.0	224.2
150,038,001	49.3	224.1	8.9	1362.0	288.5



Equations	Master process [GB]	Average slave processes [GB]	Slave processes [GB]	Total memory [GB]
25,010,001	4.05	0.41	47.74	51.79
50,027,329	8.10	0.87	101.21	109.31
75,012,921	12.15	1.37	158.54	170.68
100,020,001	16.20	1.88	217.51	233.71
125,014,761	20.25	2.38	276.04	296.29
150,038,001	24.30	2.92	338.29	362.60

Dynamic problem formulation

The Hilber-Hughes-Taylor (HHT) method [Hilb77] is used to solve a dynamic problem, equations of motion have the form

$$\mathbf{M}\,\ddot{\mathbf{u}}+\mathbf{C}\,\dot{\mathbf{u}}+\mathbf{K}\,\mathbf{u}=\mathbf{F}(t),$$

where **M**, **C** and **K** are the mass, damping and stiffness matrices of size $n\times n$, these are constant, **F** is the time depending force.

The integration formulas depend on two parameters β and γ,

$$\mathbf{u}_{i+1}=\mathbf{u}_i+h\,\dot{\mathbf{u}}_i+\frac{h^2}{2}\left[(1-2\beta)\,\ddot{\mathbf{u}}_i+2\beta\,\ddot{\mathbf{u}}_{i+1}\right],$$

$$\dot{\mathbf{u}}_{i+1}=\dot{\mathbf{u}}_i+h\left[(1-\gamma)\,\ddot{\mathbf{u}}_i+\gamma\,\ddot{\mathbf{u}}_{i+1}\right],$$

they are used to discretize the equations of motion at time t_{i+1}, h is the step size,

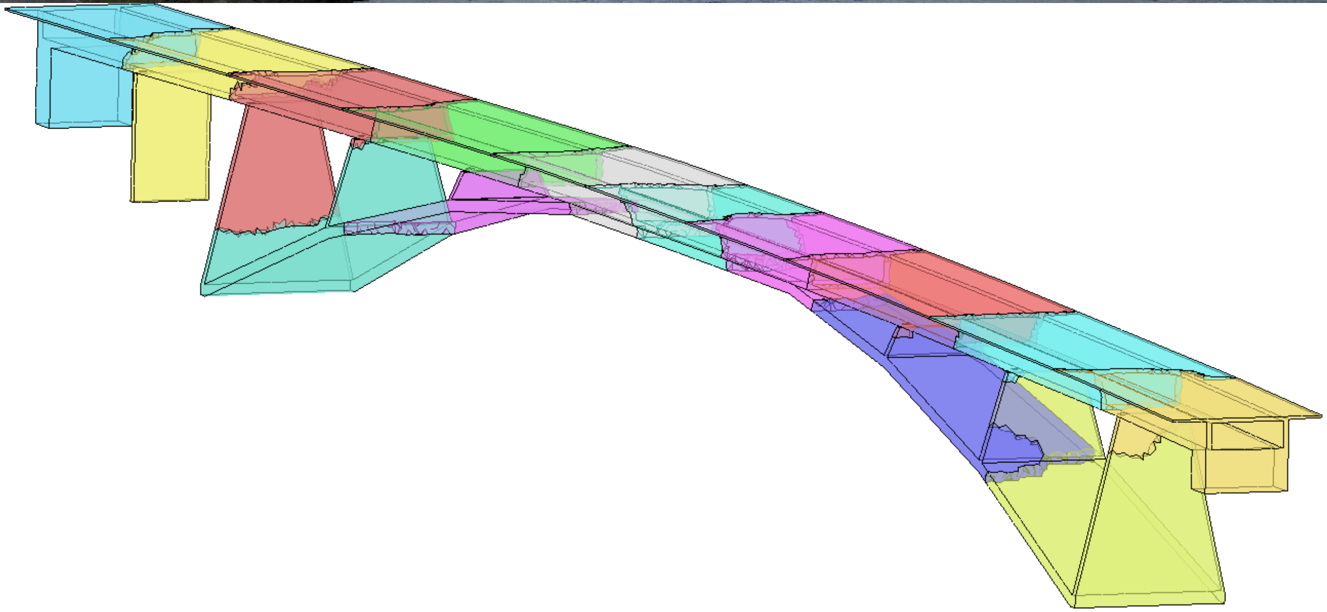
$$\mathbf{M}\,\ddot{\mathbf{u}}_{i+1}+(1+\alpha)\,\mathbf{C}\,\dot{\mathbf{u}}_{i+1}-\alpha\,\mathbf{C}\,\dot{\mathbf{u}}_i+(1+\alpha)\,\mathbf{K}\,\mathbf{u}_{i+1}-\alpha\,\mathbf{K}\,\mathbf{u}_i=\mathbf{F}(\tilde{t}_{i+1}),$$

where $\tilde{t}_{i+1}=t_n+(1+\alpha)h$.

The HTT method is second order accurate and unconditionally stable with

$$-1/3\leq\alpha\leq 0,\ \beta=(1-\alpha)^2/4 \text{ and } \gamma=(1-2\alpha)/2.$$

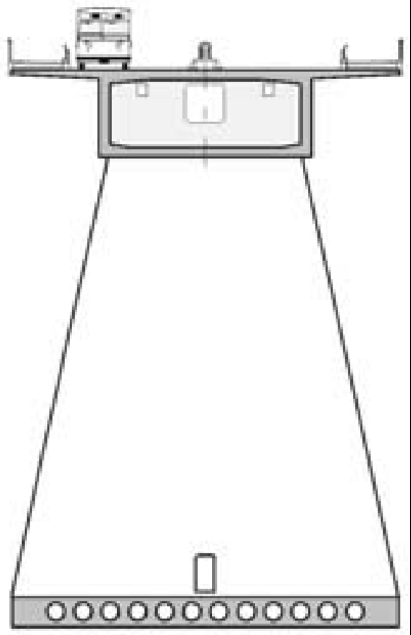
Infante Henrique bridge over the Douro river, Portugal



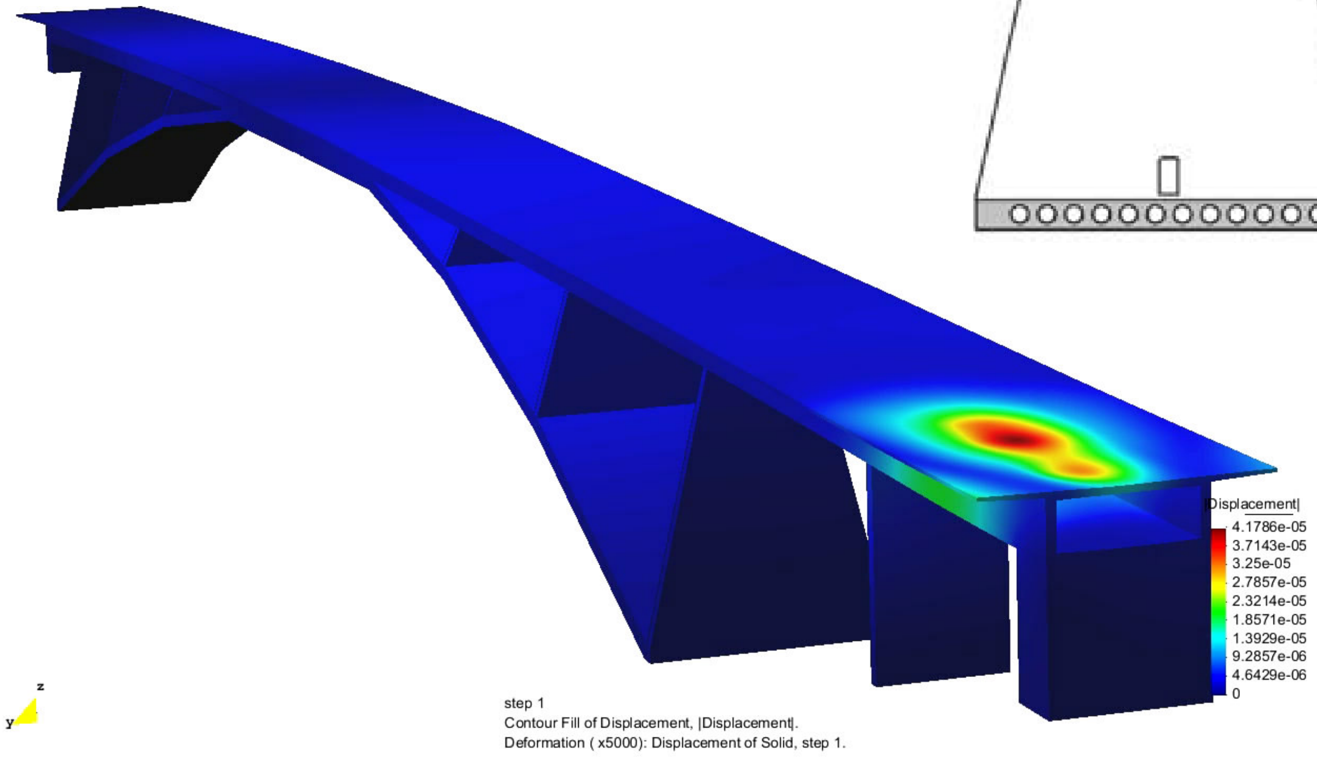
Simulation

Simulation of a 18 wheels 36 metric tons truck crossing the Infante D. Henrique Bridge.

Pre and post-process using GiD (http://gid.cimne.upc.es).



Nodes	332,462
Elements	1’381,944
Element type	Tetrahedron
Time steps	372
HHT alpha factor	0
Rayleigh damping a	0.5
Rayleigh damping b	0.5
Degrees of freedom	997,386
nnz(K)	38’302,119
Partitioning time	32.9 s
Factorization time	87.3 s
Time per step (CGJ)	132.9 s
Total time	13.7 h



Thank you!
Questions?

miguelvargas@cimat.mx
http://www.cimat.mx/~miguelvargas

References

[Gall90] K. A. Gallivan, M. T. Heath, E. Ng, J. M. Ortega, B. W. Peyton, R. J. Plemmons, C. H. Romine, A. H. Sameh, R. G. Voigt, Parallel Algorithms for Matrix Computations, SIAM, 1990.

[Geor81] A. George, J. W. H. Liu. Computer solution of large sparse positive definite systems. Prentice-Hall, 1981

[Heat91] M T. Heath, E. Ng, B. W. Peyton. Parallel Algorithms for Sparse Linear Systems. SIAM Review, Vol. 33, No. 3, pp. 420-460, 1991.

[Hilb77] H. M. Hilber, T. J. R. Hughes, and R. L. Taylor. Improved numerical dissipation for time integration algorithms in structural dynamics. Earthquake Eng. and Struct. Dynamics, 5:283–292, 1977.

[Kary99] G. Karypis, V. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing, Vol. 20-1, pp. 359-392, 1999.

[Krui04] J. Kruis. “Domain Decomposition Methods on Parallel Computers”. Progress in Engineering Computational Technology, pp 299-322. Saxe-Coburg Publications. Stirling, Scotland, UK. 2004.

[MPIF08] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. University of Tennessee, 2008.

[Saad03] Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM, 2003.

[Sori00] M. Soria-Guerrero. Parallel multigrid algorithms for computational fluid dynamics and heat transfer. Universitat Politècnica de Catalunya. Departament de Màquines i Motors Tèrmics. 2000. http://www.tesisenred.net/handle/10803/6678

[Yann81] M. Yannakakis. Computing the minimum fill-in is NP-complete. SIAM Journal on Algebraic Discrete Methods, Volume 2, Issue 1, pp 77-79, March, 1981.