FEMT, an open source library and tools for solving large sparse systems of equations in parallel

CIMAT

Miguel Vargas-Félix, Salvador Botello-Rionda

March, 2013

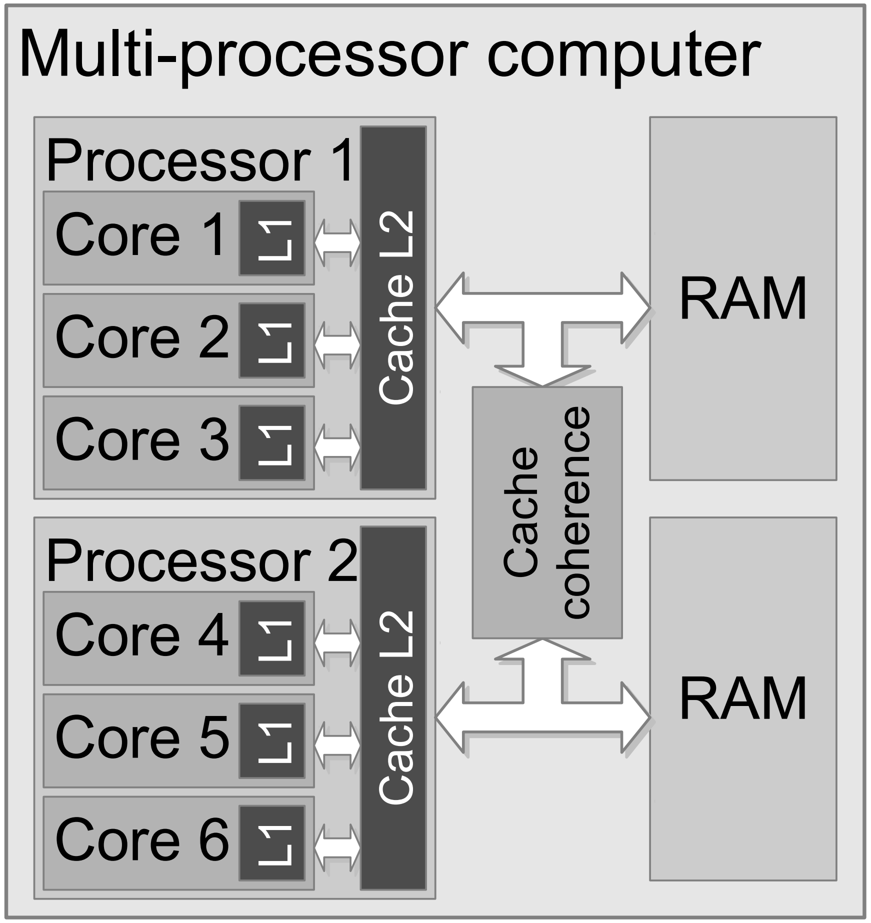
http://www.cimat.mx/~miguelvargas/FEMT/

FEMT (Finite Element Method Tools)

- FEMT is an open source multi-platform library and tools (Windows, GNU/Linux, Mac OS, BSD), released under the GNU Library General Public License.
- It contains routines to handle and solve large linear systems of equations resulting from finite element and finite volume discretizations.
- It includes several solvers that run in parallel in multi-core computers using OpenMP, or in clusters of computers with MPI.
- These solvers can be used in stationary or dynamic problems.
- A sparse matrix of $1'000,000 \times 1'000,000$ can be factorized with Cholesky in 60 seconds in a 8-core computer.
- Using a cluster with 124 cores it can solve a system with 150’000,000 equations in 3.7 hours.
- It has been programmed in modern standard C++. It supports Microsoft Visual C++ >= 2008, GNU Compiler Collection >=4.3 or Intel C++ Compiler >=10.1.
- FEMT also includes several programs that allow access to library routines through pipes. This makes really easy to use FEMT from Fortran, C, Python, C#, etc.

Parallelization using multi-core computers

Several processing units (cores) that access the same memory. All the cores fight for the RAM. We must design our programs to reduce this conflict.



In modern computers the processor is a lot faster than the memory, between them a high speed memory is used to improve data access. The ***cache***.

The most important issue to achieve high performance is to use the cache efficiently.

Access to	Cycles
Register	≤ 1
L1	~ 3
L2	~ 14
Main memory	~ 240

- Work using continuous memory blocks.
- Access memory in sequence.
- Each core should work in an independent memory area.

Algorithms to solve system of equations should take care of this.

How important is it?

```c
#include <stdio.h>
#include <stdlib.h>

#define ROWS 10000000
#define COLS 200

int main(void)
{
   char* A = (char*)malloc(ROWS*COLS);

   for (int j = 0; j < COLS; ++j)
   {
      for (int i = 0; i < ROWS; ++i)
      {
         A[i*COLS + j] = 0;
      }
   }

   free(A);

   return 0;
}
```

126.760 seconds

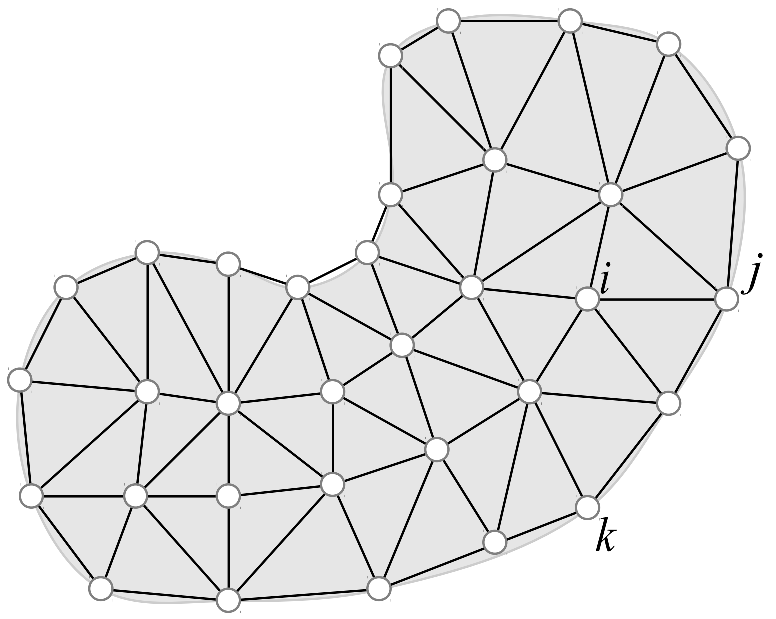
```c
#include <stdio.h>
#include <stdlib.h>

#define ROWS 10000000
#define COLS 200

int main(void)
{
   char* A = (char*)malloc(ROWS*COLS);

   for (int i = 0; i < ROWS; ++i)
   {
      for (int j = 0; j < COLS; ++j)
      {
         A[i*COLS + j] = 0;
      }
   }

   free(A);

   return 0;
}
```

6.757 seconds

The code on the right is 18.76 times faster than the code on the left

Sparse matrices

Relation between adjacent nodes is captured as entries in a matrix. Because a node has adjacency with only a few others, the resulting matrix has a very sparse structure.



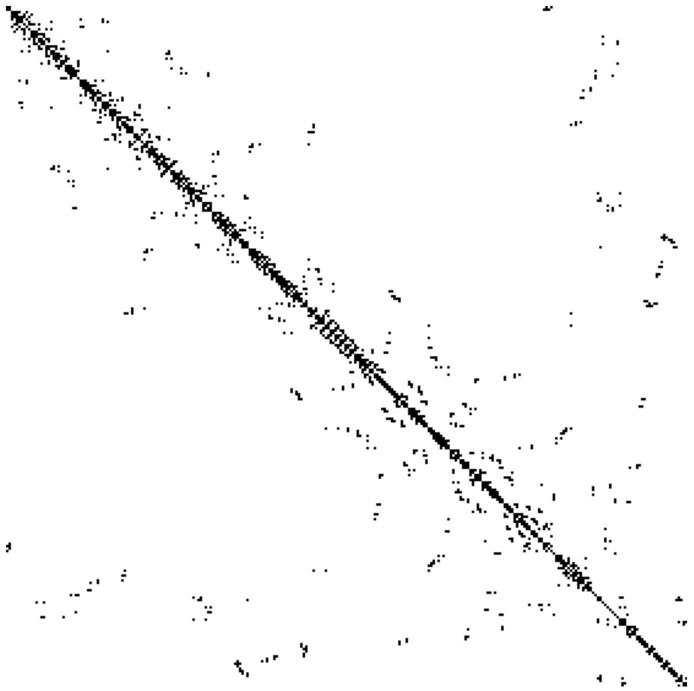
$$\mathbf{A} = \begin{pmatrix} \circ & \circ & \circ & \circ & \circ & \circ & \circ & \cdots \\ \circ & a_{ii} & \circ & a_{ij} & \circ & 0 & \circ & \cdots \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \cdots \\ \circ & a_{ji} & \circ & a_{jj} & \circ & 0 & \circ & \cdots \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \cdots \\ \circ & 0 & \circ & 0 & \circ & a_{kk} & \circ & \cdots \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

When assembled, we have to solve a linear system of equations $\mathbf{A}\,\mathbf{x} = \mathbf{b}$.

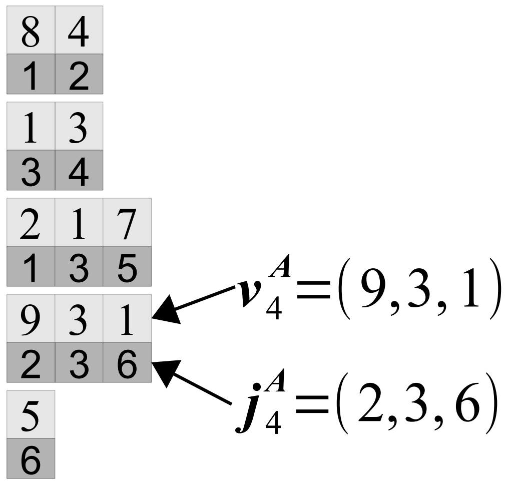
In finite element and finite volume all matrices have symmetric structure, and depending on the problem symmetric values or not.

Lets define the notation $\eta(\boldsymbol{A})$, it indicates the number of non-zero entries of $\boldsymbol{A}$.

For example, $\boldsymbol{A} \in \mathbb{R}^{556\times 556}$ has 309,136 entries, with $\eta(\boldsymbol{A}) = 1810$, this means that only the 0.58% of the entries are non zero.



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



8 4
1 2
1 3
3 4
2 1 7
1 3 5
9 3 1
2 3 6
5
6
$\boldsymbol{v}_4^{\boldsymbol{A}} = (9,3,1)$
$\boldsymbol{j}_4^{\boldsymbol{A}} = (2,3,6)$

Cholesky factorization for sparse matrices

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

To calculate entries of $\mathbf{L}$

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

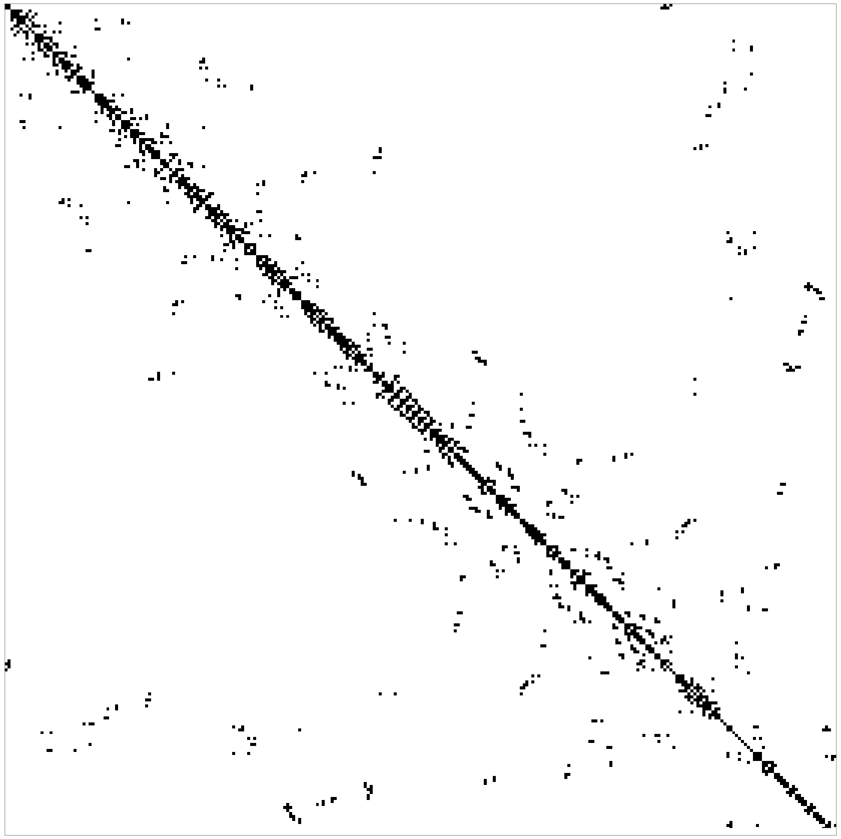
$$L_{jj}=\sqrt{A_{jj}-\sum_{k=1}^{j-1}L_{jk}^2}.$$

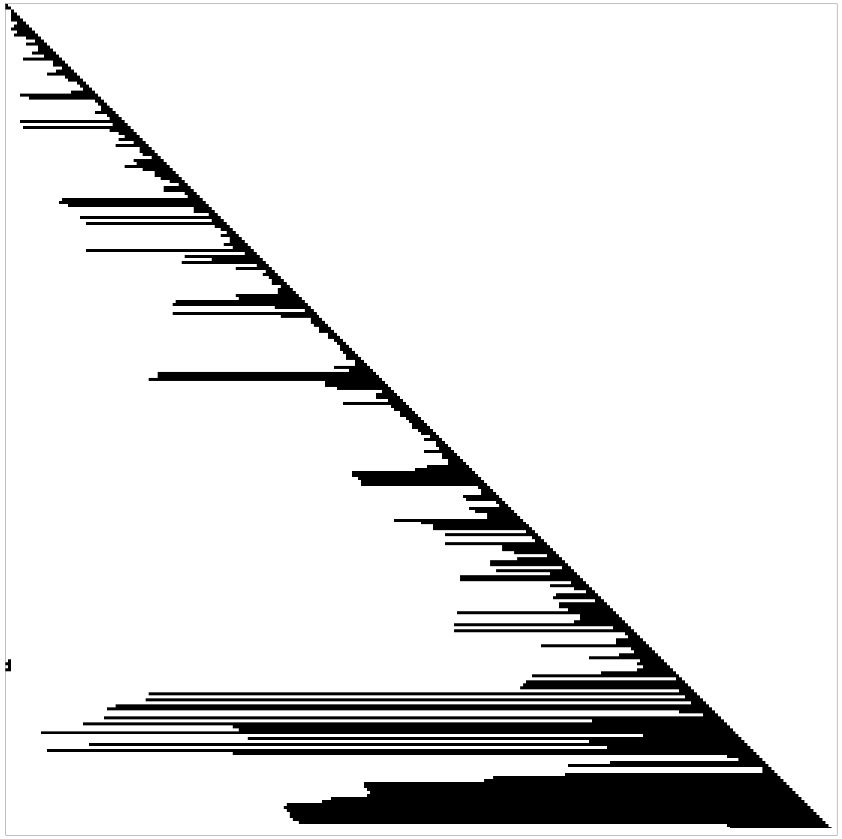
We use four strategies to reduce time and memory usage when performing this factorization on sparse matrices:

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 $(\boldsymbol{P}\,\boldsymbol{A}\,\boldsymbol{P}^{\mathrm{T}})(\boldsymbol{P}\,\boldsymbol{x})=(\boldsymbol{P}\,\boldsymbol{b})$.
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. $\eta\left(\boldsymbol{L}\right)$ indicates the number of non-zero entries of $\boldsymbol{L}$.

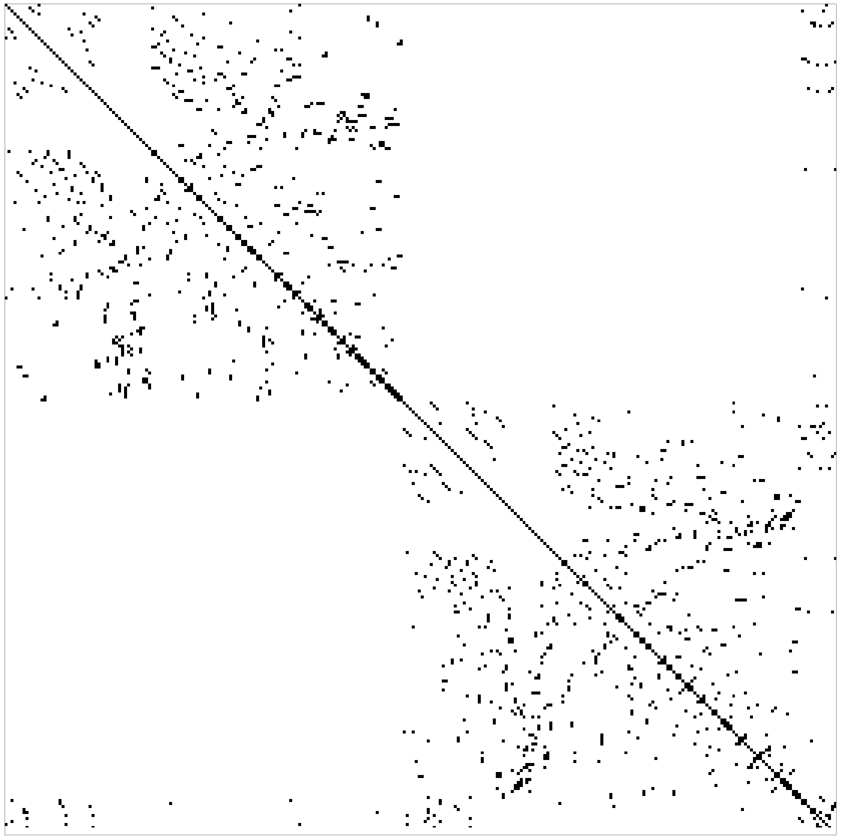
$\boldsymbol{A}=$ 

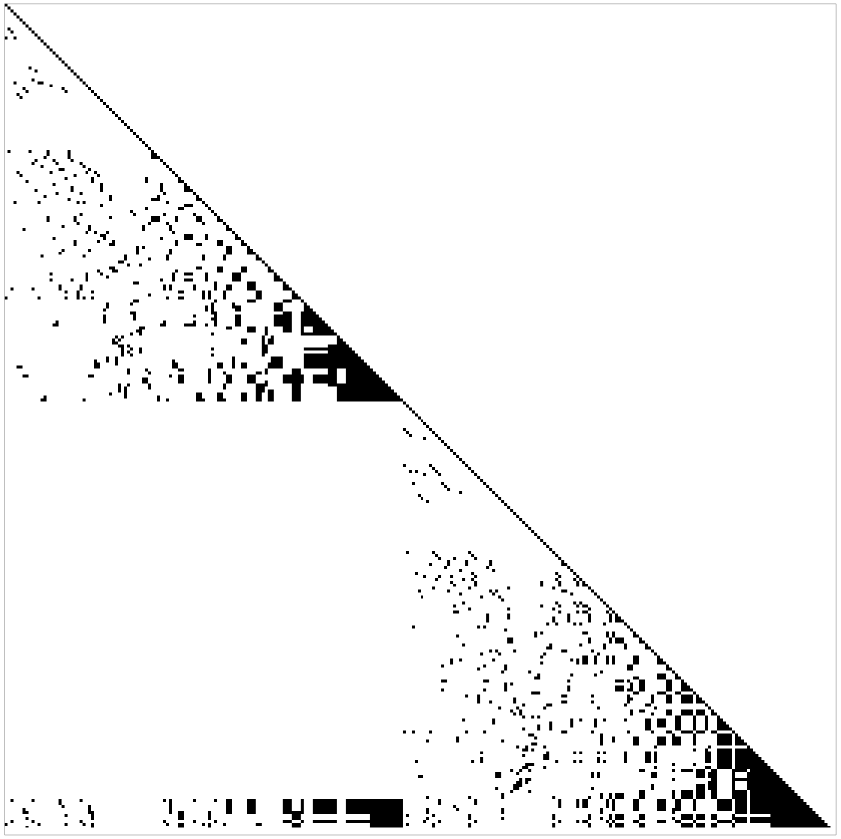
$\boldsymbol{L}=$ 

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

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 $j=1\dots n$,

$$\boldsymbol{a}_j \overset{\text{def}}{=} \{k>j \mid A_{kj}\neq 0\},$$
$$\boldsymbol{l}_j \overset{\text{def}}{=} \{k>j \mid L_{kj}\neq 0\}.$$

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
    $p \leftarrow \begin{cases} \min\{i \in \boldsymbol{l}_j\} & \text{si } \boldsymbol{l}_j \neq \emptyset \\ j & \text{otro caso} \end{cases}$
    $\boldsymbol{r}_p \leftarrow \boldsymbol{r}_p \cup \{j\}$
end_for
```

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

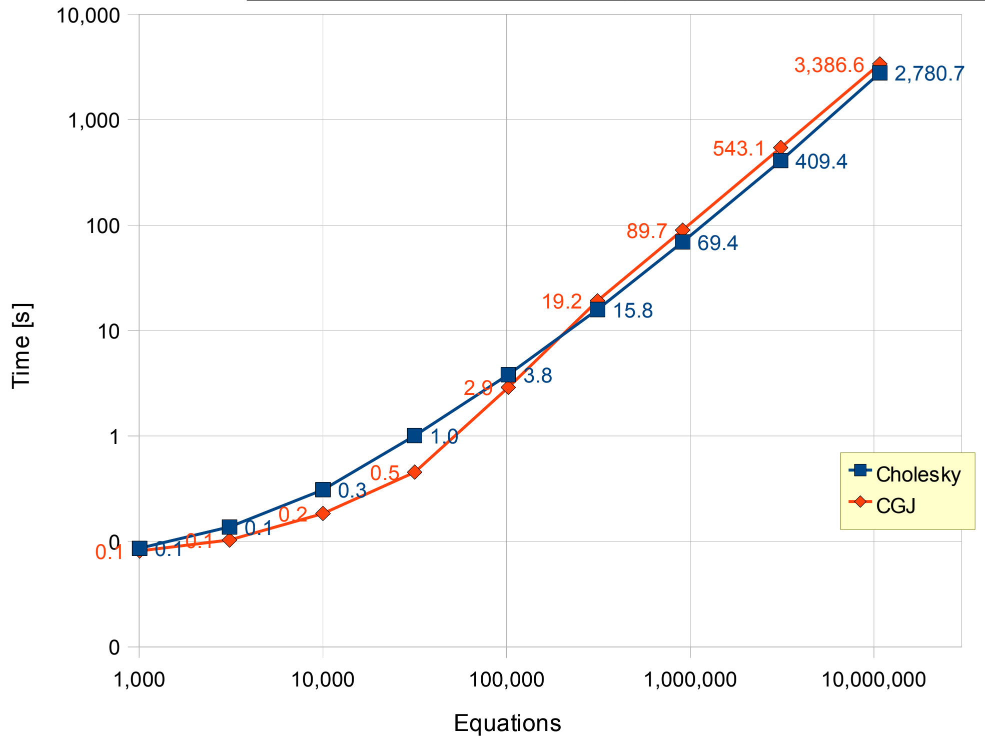
$$\boldsymbol{a}_2=\{3,4\} \qquad \boldsymbol{l}_2=\{3,4,6\}$$

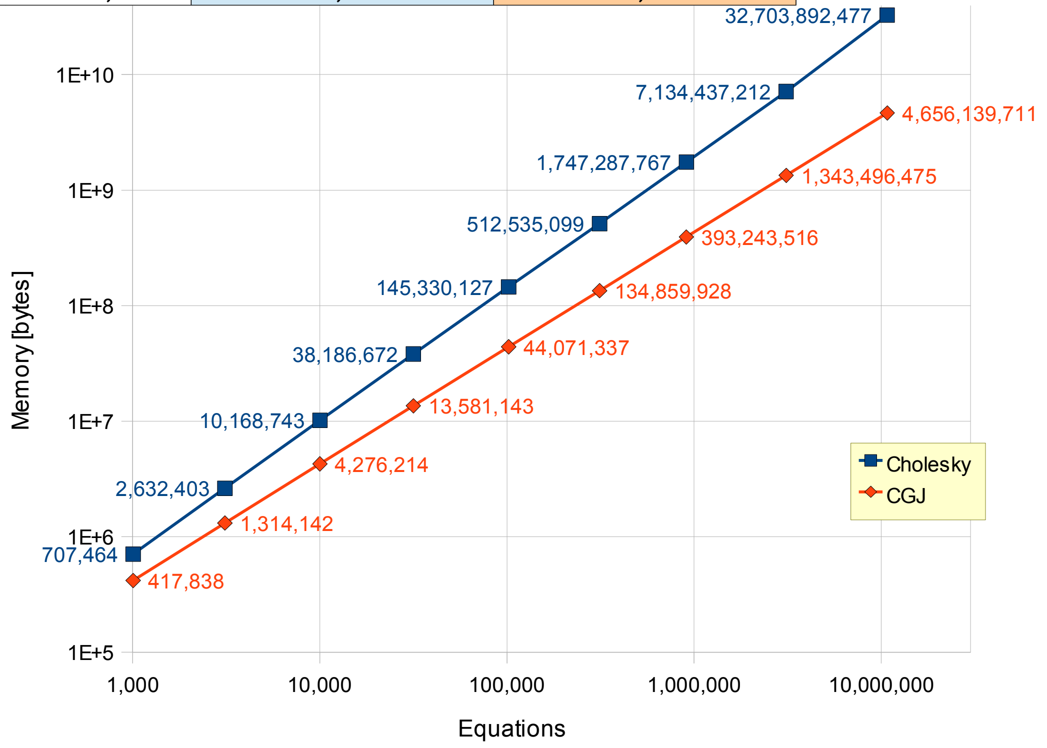
This algorithm is very efficient, its complexity in time and space has an order of $O(\eta(\boldsymbol{L}))$.

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





Parallelization of factorization

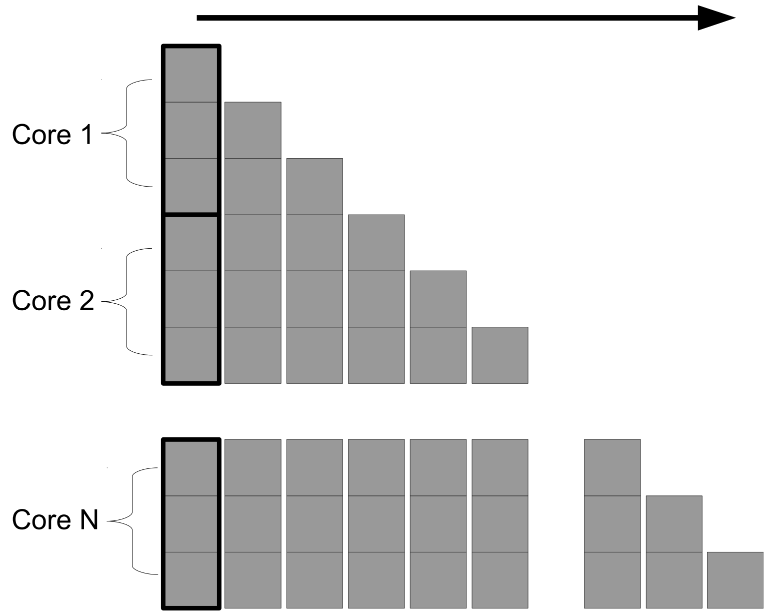
The calculation of the non-zero L_{ij} entries can be done in parallel if we fill $\boldsymbol{L}$ column by column [Heat91].

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

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

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

The paralellization was made using the OpenMP schema.



LU factorization for sparse matrices

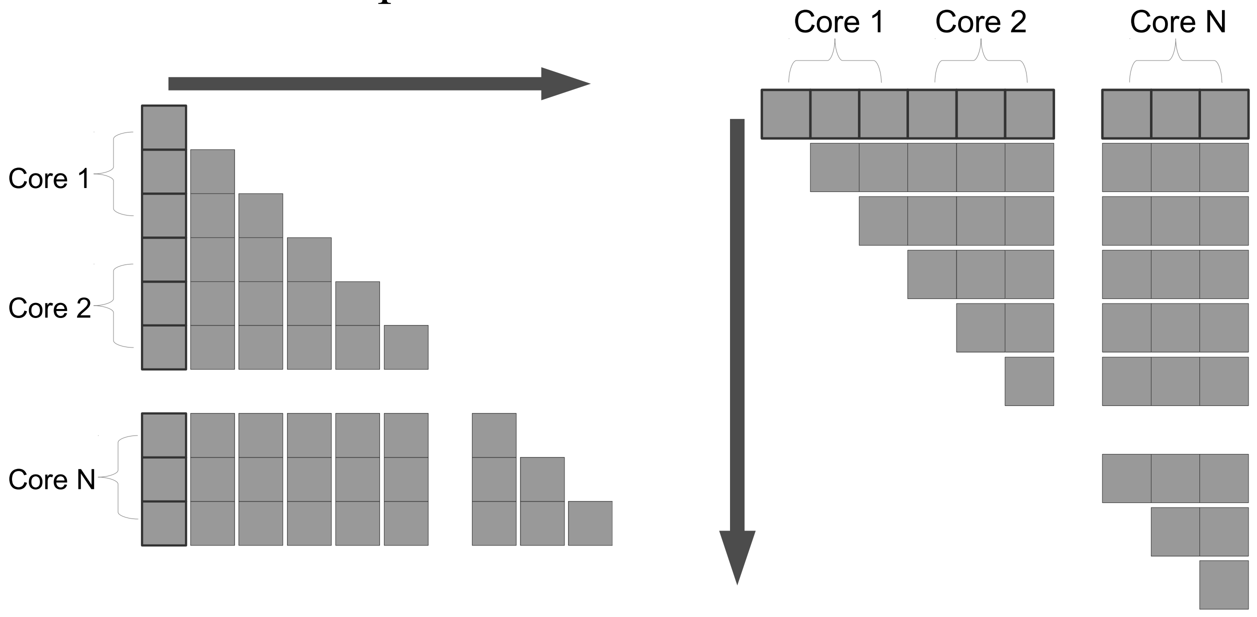
Symbolic Cholesky factorization could be use to determine the structure of the LU factorization if the matrix has symmetric structure,

$$U_{ij} = A_{ij} - \sum_{\substack{k \in (J(i) \cap J(j)) \\ k<j}} L_{ik} U_{jk} \quad \text{for } i > j,$$

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

$$U_{ii} = A_{ii} - \sum_{\substack{k \in J(i) \\ k<i}} L_{ik} U_{ik}, \; L_{ii} = 1.$$

Filling of $\boldsymbol{L}$ and $\boldsymbol{U}$ can also be done in parallel.



Preconditioning the parallel conjugate gradient

Instead of solving the problem

$$\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}=0,$$

we apply a preconditioner matrix $\boldsymbol{M}^{-1}$, it reduces the condition number of the system

$$\boldsymbol{M}^{-1}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})=0.$$

For large systems of equations, it is necessary to choose preconditioners that are also sparse.

```
$x_0$, initial coordinate
$g_0 \leftarrow A x_0 - b$, initial gradient
$q_0 \leftarrow M^{-1} g_0$
$p_0 \leftarrow -q_0$, initial descent direction
$\varepsilon$, tolerancia
$k \leftarrow 0$
while $\|g_k\| > \varepsilon$
    $w \leftarrow A p_k$
    $\alpha_k \leftarrow \frac{g_k^T q_k}{p_k^T w}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $g_{k+1} \leftarrow g_k + \alpha w$
    $q_{k+1} \leftarrow M^{-1} g$
    $\beta_k \leftarrow \frac{g_{k+1}^T q_{k+1}}{g_k^T q_k}$
    $p_{k+1} \leftarrow -q_{k+1} + \beta_{k+1} p_k$
    $k \leftarrow k+1$
```

Jacobi preconditioner

The $\boldsymbol{M}^{-1}$ is a diagonal matrix

$$\left(\boldsymbol{M}^{-1}\right)_{i\,j}=\begin{cases}\dfrac{1}{\boldsymbol{A}_{i\,i}} & \text{si } i=j\\ 0 & \text{si } i\neq j\end{cases}.$$

It is commonly stored as a vector.

Parallelization of this preconditioner is straightforward, because calculation of each entry of $\boldsymbol{q}_k$ is independent.

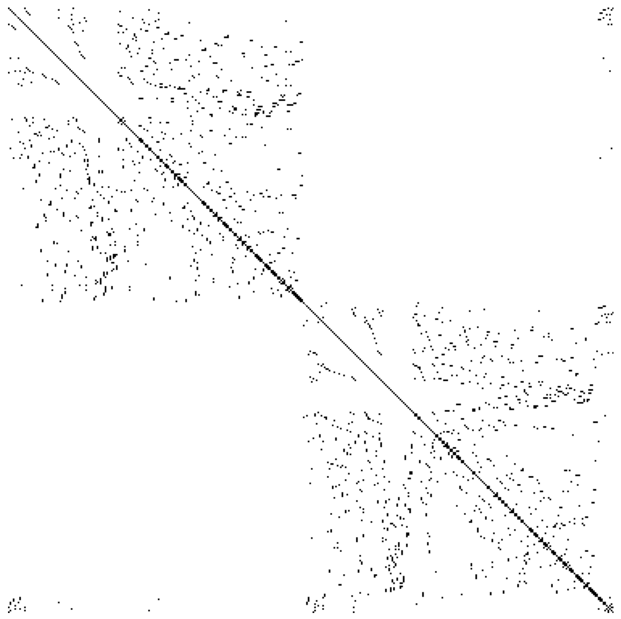
Incomplete Cholesky factorization preconditioner

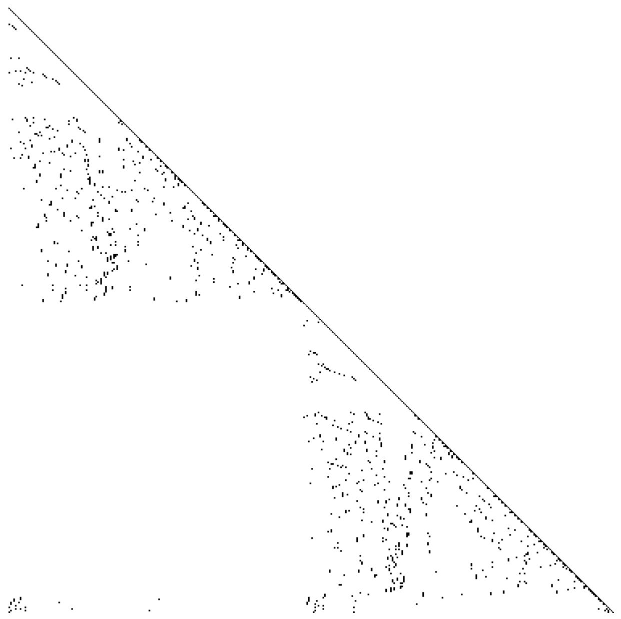
This preconditioner has the form

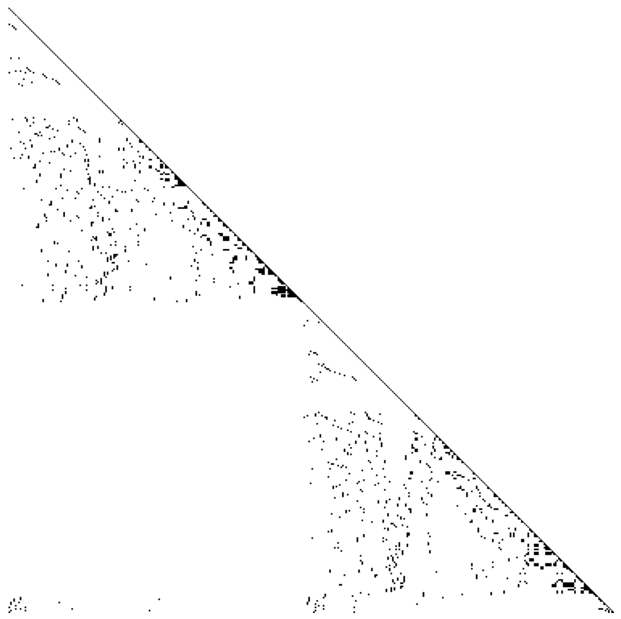
$$\boldsymbol{M}^{-1} = \boldsymbol{G}_l \boldsymbol{G}_l^{\mathrm{T}},$$

where $\boldsymbol{G}_l$ is a lower triangular sparse matrix that have a structure similar to the Cholesky factorization of $\boldsymbol{A}$.

- The structure of $\boldsymbol{G}_0$ is equal to the structure of the lower triangular form of $\boldsymbol{A}$.
- The structure of $\boldsymbol{G}_m$ is equal to the structure of $\boldsymbol{L}$ (complete Cholesky factorization of $\boldsymbol{A}$).
- For $0 < l < m$ the structure of $\boldsymbol{G}_l$ is creating having a number of entries between $\boldsymbol{L}$ and the lower triangular form of $\boldsymbol{A}$, making easy to control the sparsity of the preconditioner.


A


$\mathbf{G}_0$


$\mathbf{G}_{50}$

Entries are filled using

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

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

This preconditioner is not always SPD. To overcome this, we can use the algorithm of Munksgaard [Munk80], it consists in two strategies:

1. Perturbation of the diagonal of $\boldsymbol{A}$ by an α factor,

$$D_{jj} = \alpha A_{jj} - \sum_{k=1}^{j-1} H_{jk}^2 D_k.$$

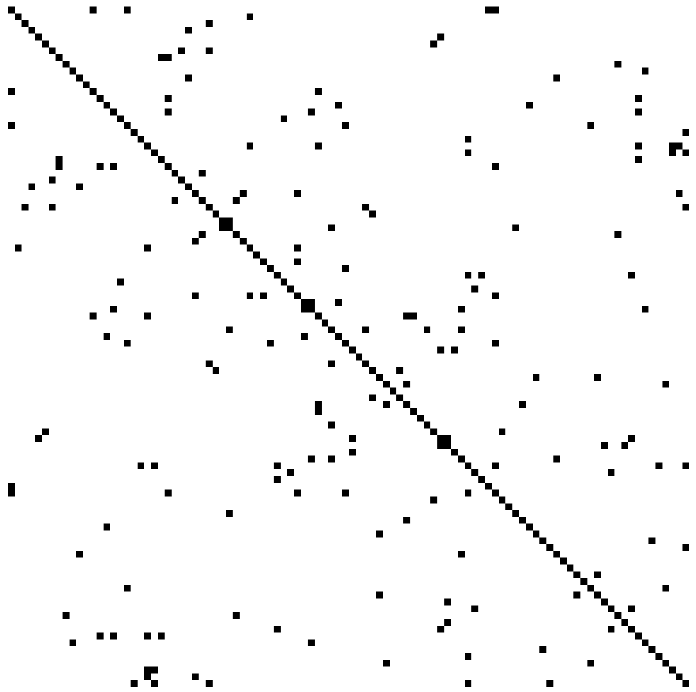
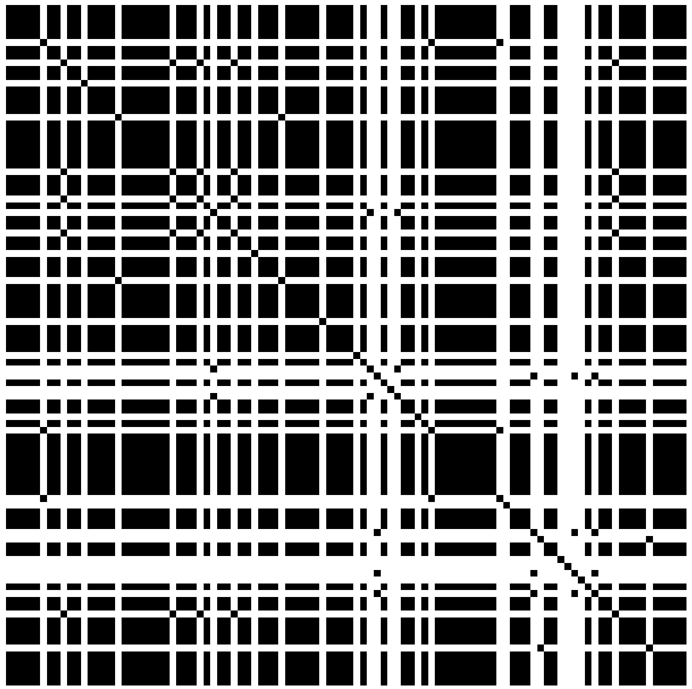
2. Perturbation of pivots, if they are negatives or near zero,

$$\text{if } D_{jj} \leq u\left(\sum_{k \neq j} |a_{jk}| \right), \text{ then } D_{jj} = \begin{cases} \sum_{k \neq j} |a_{jk}| & \text{si } \sum_{k \neq j} |a_{jk}| \neq 0 \\ 1 & \text{si } \sum_{k \neq j} |a_{jk}| = 0 \end{cases}.$$

The use of this preconditioner implies to solve a system of equations in each CG step using a backward and a forward substitution algorithm, this operations are fast given the sparsity of $\boldsymbol{G}_l$. Unfortunately the dependency of values makes these substitutions very hard to parallelize.

Factorized sparse approximate inverse preconditioner

The aim of this preconditioner is to construct $\boldsymbol{M}$ to be an approximation of the inverse of $\boldsymbol{A}$ with the property of being sparse. The inverse of a sparse matrix is not necessary sparse.

A way to create an approximate inverse is to minimize the Frobenius norm of the residual $\boldsymbol{I}-\boldsymbol{A}\boldsymbol{M}$,

$$F(\boldsymbol{M})=\|\boldsymbol{I}-\boldsymbol{A}\boldsymbol{M}\|_{\mathrm{F}}^2.$$

The Frobenius norm is defined as

$$\|\boldsymbol{A}\|_{\mathrm{F}}=\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}=\sqrt{\operatorname{tr}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})}.$$

It is possible to separate $F(\boldsymbol{M}) = \|\boldsymbol{I} - \boldsymbol{A}\boldsymbol{M}\|_{\mathrm{F}}^2$ into decoupled sums of 2-norms for each column [Chow98],

$$F(\boldsymbol{M}) = \|\boldsymbol{I} - \boldsymbol{A}\boldsymbol{M}\|_{\mathrm{F}}^2 = \sum_{j=1}^{n} \|\boldsymbol{e}_j - \boldsymbol{A}\boldsymbol{m}_j\|_2^2,$$

where $\boldsymbol{e}_j$ is the j-th column of $\boldsymbol{I}$ and $\boldsymbol{m}_j$ is the j-th column of $\boldsymbol{M}$. With this separation we can parallelize the construction of the preconditioner.

The factorized sparse approximate inverse preconditioner [Chow01] creates a preconditioner

$$\boldsymbol{M} = \boldsymbol{G}_l^{\mathrm{T}} \boldsymbol{G}_l,$$

where $\boldsymbol{G}$ is a lower triangular matrix such that

$$\boldsymbol{G}_l \approx \boldsymbol{L}^{-1},$$

where $\boldsymbol{L}$ is the Cholesky factor of $\boldsymbol{A}$. l is a positive integer that indicates a level of sparsity of the matrix.

Instead of minimizing $F(\boldsymbol{M}) = \|\boldsymbol{I} - \boldsymbol{A}\boldsymbol{M}\|_{\mathrm{F}}^2$, we minimize $\|\boldsymbol{I} - \boldsymbol{G}_l \boldsymbol{L}\|_{\mathrm{F}}^2$, it is noticeable that this can be done without knowing $\boldsymbol{L}$.

This preconditioner has these features:

- $\boldsymbol{M}$ is SPD if there are no zeroes in the diagonal of $\boldsymbol{G}_l$.
- The algorithm to construct the preconditioner is parallelizable.
- This algorithm is stable if $\boldsymbol{A}$ is SPD.

Parallel preconditionated biconjugated gradient

The algorithm is [Meie92]:

```
$\varepsilon$, tolerance
$\boldsymbol{x}_0$, initial coordinate
$\boldsymbol{g}_0 \leftarrow \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b}$, initial gradient
$\tilde{\boldsymbol{g}}_0 \leftarrow \boldsymbol{g}_0^{\mathrm{T}}$, initial pseudo-gradient
$\boldsymbol{q}_0 \leftarrow \boldsymbol{M}^{-1}\boldsymbol{g}_0$
$\tilde{\boldsymbol{q}}_0 \leftarrow \tilde{\boldsymbol{g}}_0\boldsymbol{M}^{-1}$
$\boldsymbol{p}_0 \leftarrow -\boldsymbol{q}_0$, initial descent direction
$\bar{\boldsymbol{p}}_0 \leftarrow -\tilde{\boldsymbol{q}}_0$, initial pseudo-direction of descent
$k \leftarrow 0$
while $\|\boldsymbol{g}_k\| > \varepsilon$
    $\boldsymbol{w} \leftarrow \boldsymbol{A}\boldsymbol{p}_k$
    $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{p}}_k\boldsymbol{A}$
    $\alpha_k \leftarrow -\dfrac{\tilde{\boldsymbol{q}}_k\boldsymbol{g}_k}{\tilde{\boldsymbol{p}}_k\boldsymbol{w}}$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$
    $\boldsymbol{g}_{k+1} \leftarrow \boldsymbol{g}_k + \alpha\boldsymbol{w}$
    $\tilde{\boldsymbol{g}}_{k+1} \leftarrow \tilde{\boldsymbol{g}}_k + \alpha\tilde{\boldsymbol{w}}$
    $\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{M}^{-1}\boldsymbol{g}_{k+1}$
    $\tilde{\boldsymbol{q}}_{k+1} \leftarrow \tilde{\boldsymbol{g}}_{k+1}\boldsymbol{M}^{-1}$
    $\beta_k \leftarrow \dfrac{\tilde{\boldsymbol{g}}_{k+1}\boldsymbol{q}_{k+1}}{\tilde{\boldsymbol{g}}_k\boldsymbol{q}_k}$
    $\boldsymbol{p}_{k+1} \leftarrow -\boldsymbol{q}_{k+1} + \beta_{k+1}\boldsymbol{p}_k$
    $\tilde{\boldsymbol{p}}_{k+1} \leftarrow -\tilde{\boldsymbol{q}}_{k+1} + \beta_{k+1}\tilde{\boldsymbol{p}}_k$
    $k \leftarrow k+1$
```

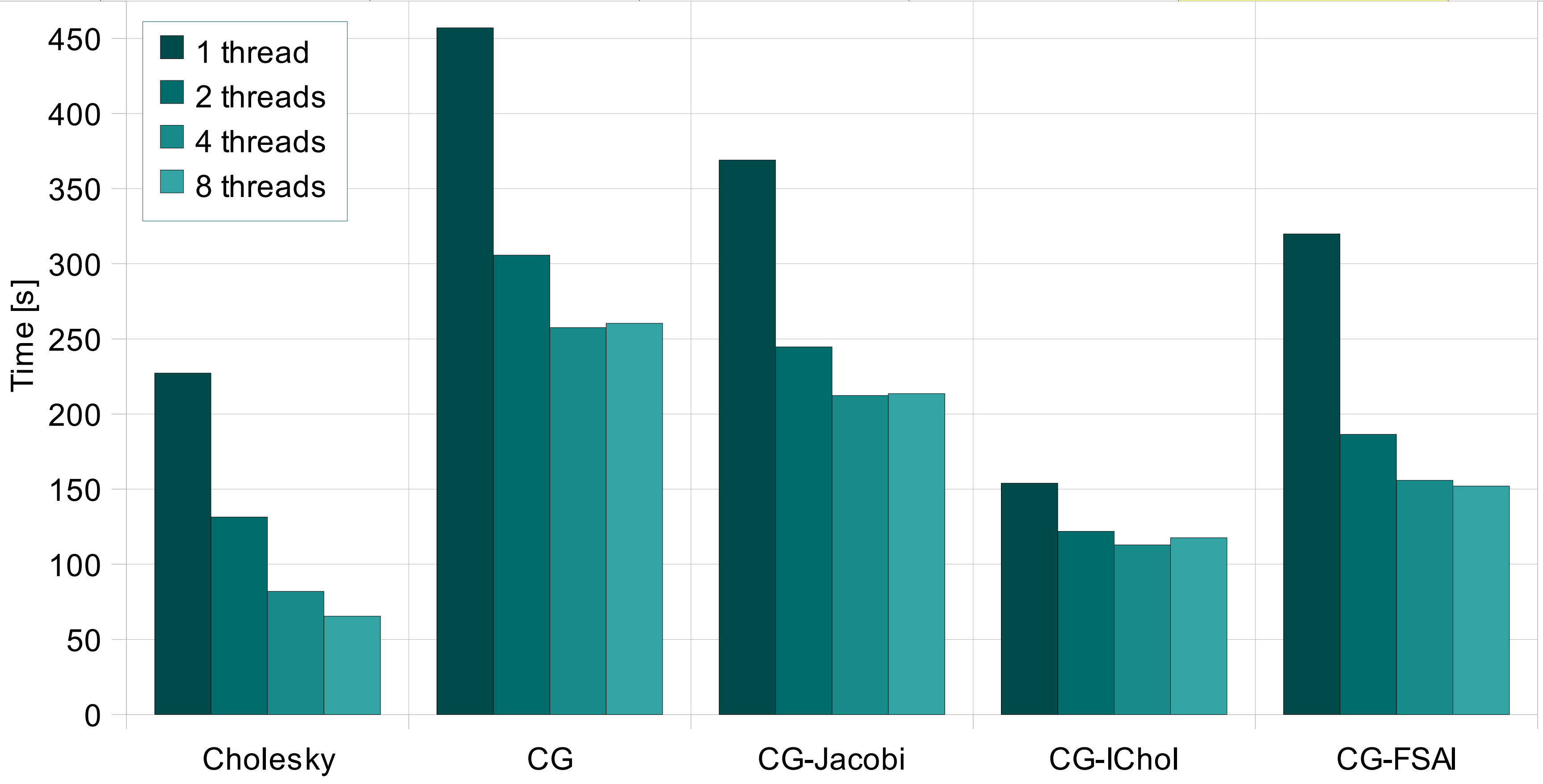
Preconditioners:

- Jacobi
- Incomplete LU factorization
- Sparse approximate inverse

Performance comparisons

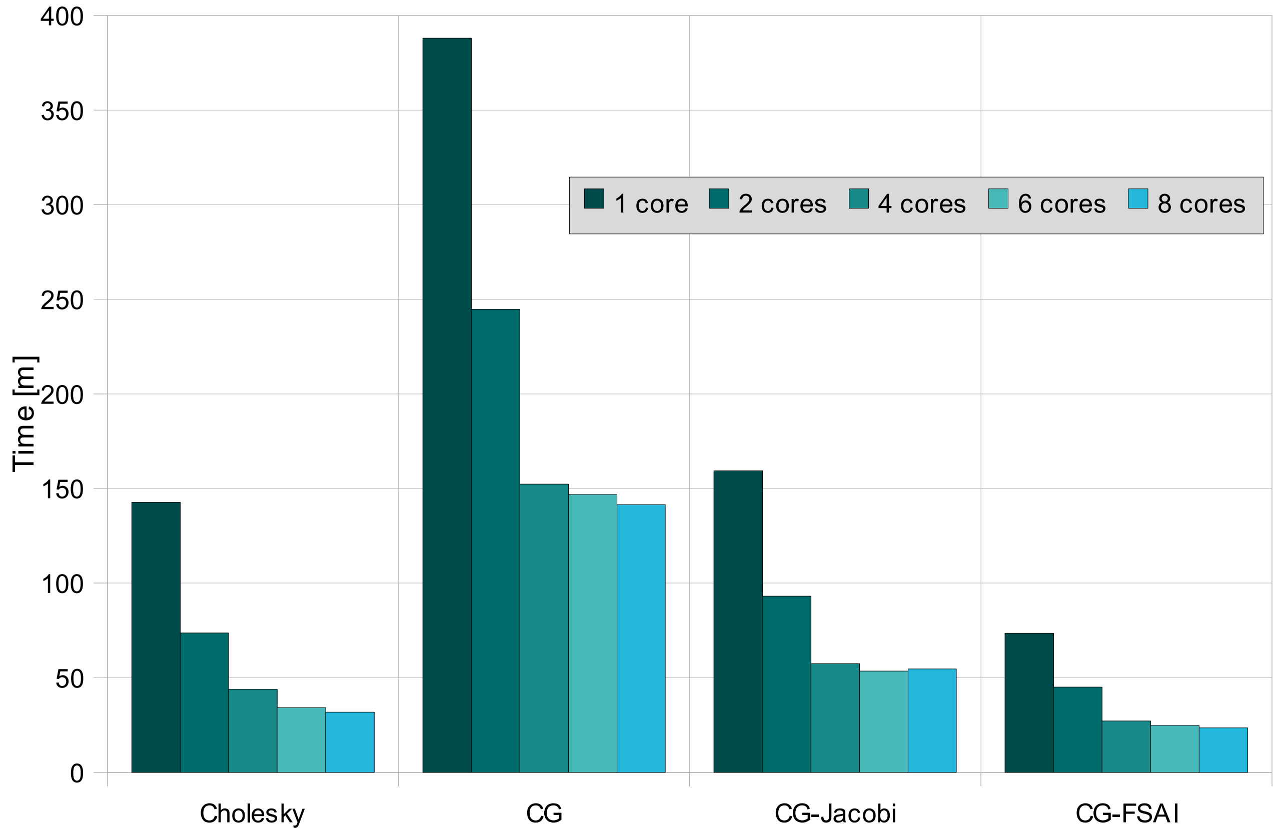
2D solid linear deformation, 501,264 elements, con 909,540 equations, $\eta(A) = 18'062{,}500$.

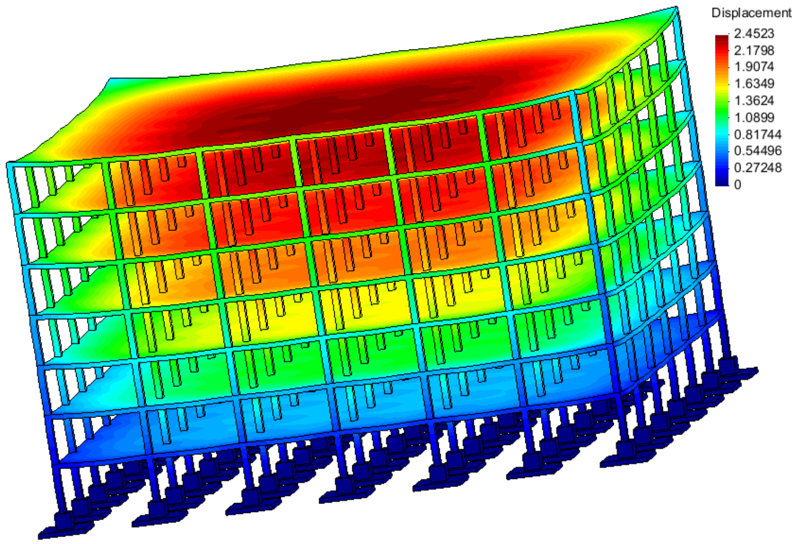
Solver	1 thread [s]	2 threads [s]	4 threads [s]	8 threads [s]	Steps	Memory
Cholesky	227.2	131.4	81.9	65.4		3,051,144,550
CG	457.0	305.8	257.5	260.4	9,251	317,929,450
CG-Jacobi	369.0	244.7	212.4	213.7	6,895	325,972,366
CG-IChol	153.9	121.9	112.8	117.6	1,384	586,380,322
CG-FSAI	319.8	186.5	155.7	152.1	3,953	430,291,930



3D solid deformed by self-weight

Elements:	264,250
Nodes:	326,228
Equations:	978,684
nnz(K):	69,255,522

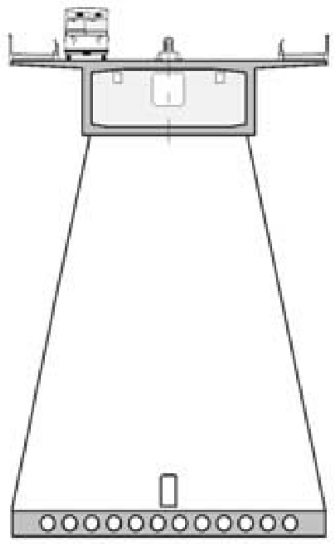




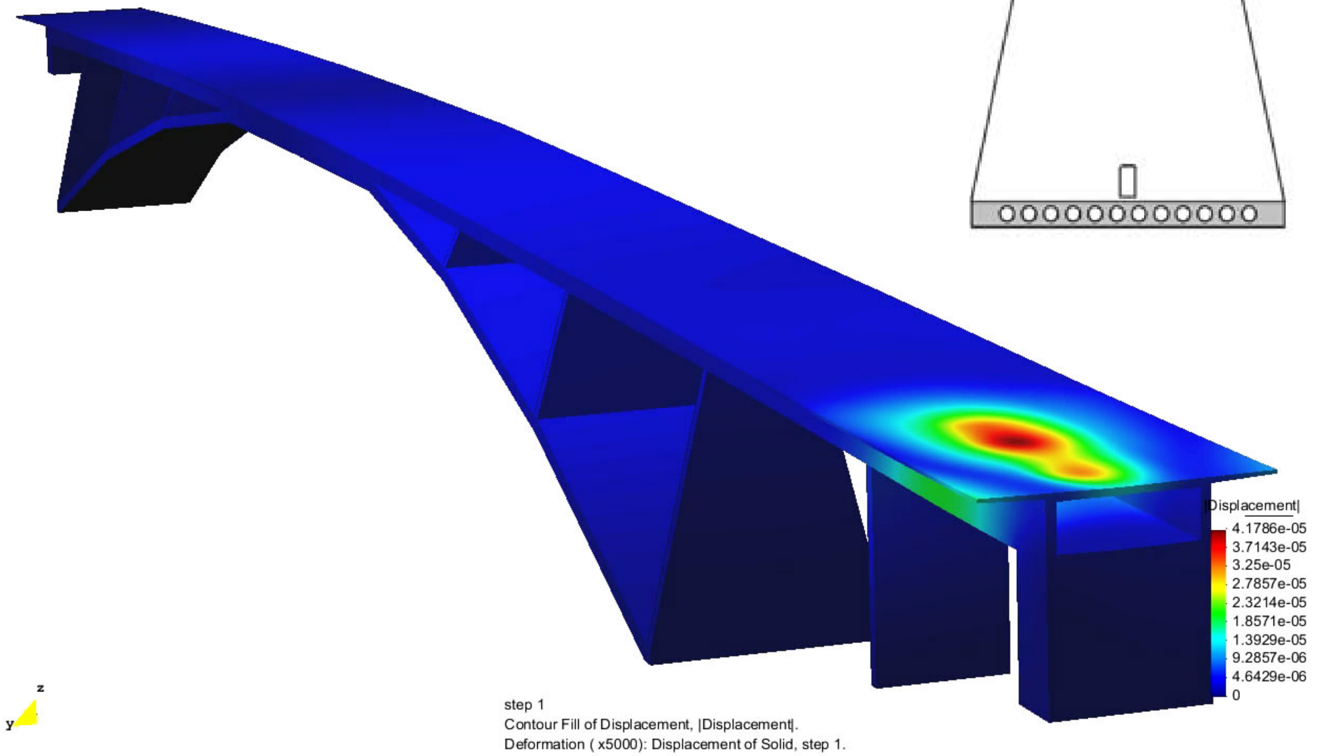
Solver	1 core [m]	2 cores [m]	4 cores [m]	6 cores [m]	8 cores [m]	Memory
Cholesky	142	73	43	34	31	19,864’132,056
CG	387	244	152	146	141	922’437,575
CG-Jacobi	159	93	57	53	54	923’360,936
CG-FSAI	73	45	27	24	23	1,440’239,572

Infante Henrique bridge over the Douro river, Portugal

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



Nodes	**337,195**
Elements	**1’413,279**
Element type	Tetrahedron
HHT alpha factor	0
Rayleigh damping a	0.5
Rayleigh damping b	0.5
Degrees of freedom	**1’011,585**
nnz(K)	38’104,965
Time to assemble K	4.5 s
Time to reorder K	32.4 s
Factorization time	178.8 s
Time steps	**372**
Time per step	2.6 s
Total time	**1205.1 s**

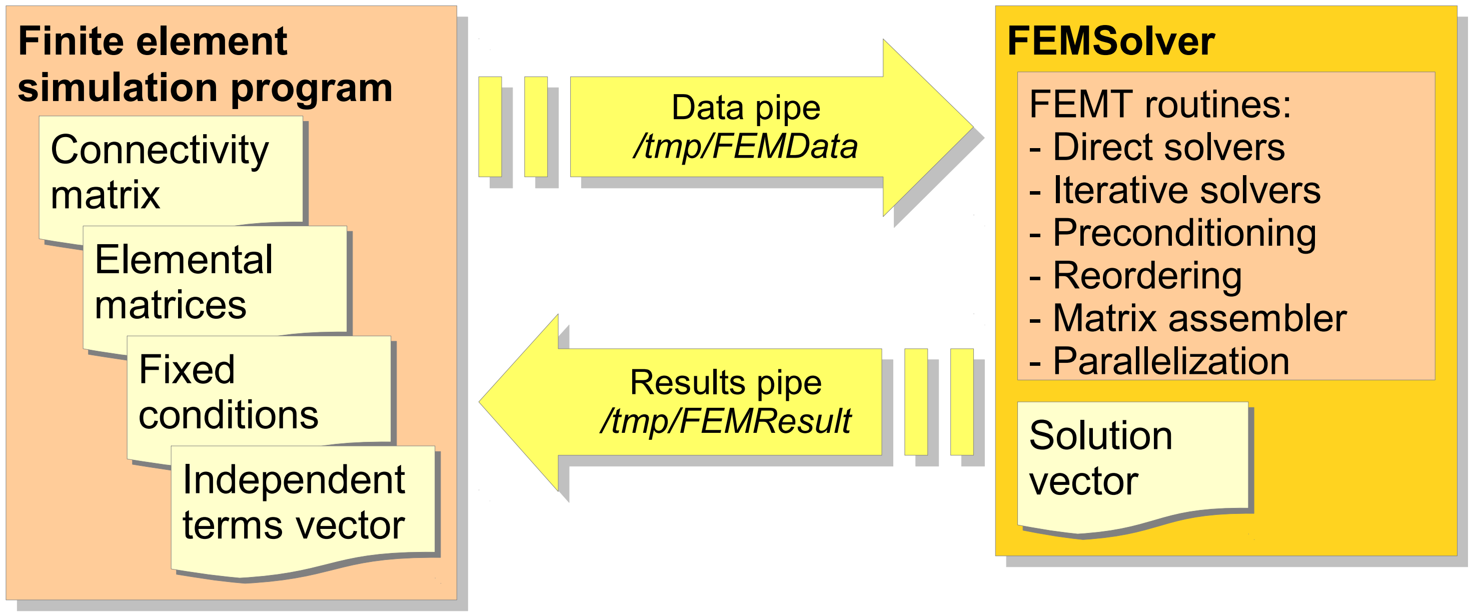


Peak allocated memory: 9,537’397,868 bytes
Computer: 2 x Intel(R) Xeon(R) CPU E5620, 8 cores, 12MB cache, 32 GB of RAM

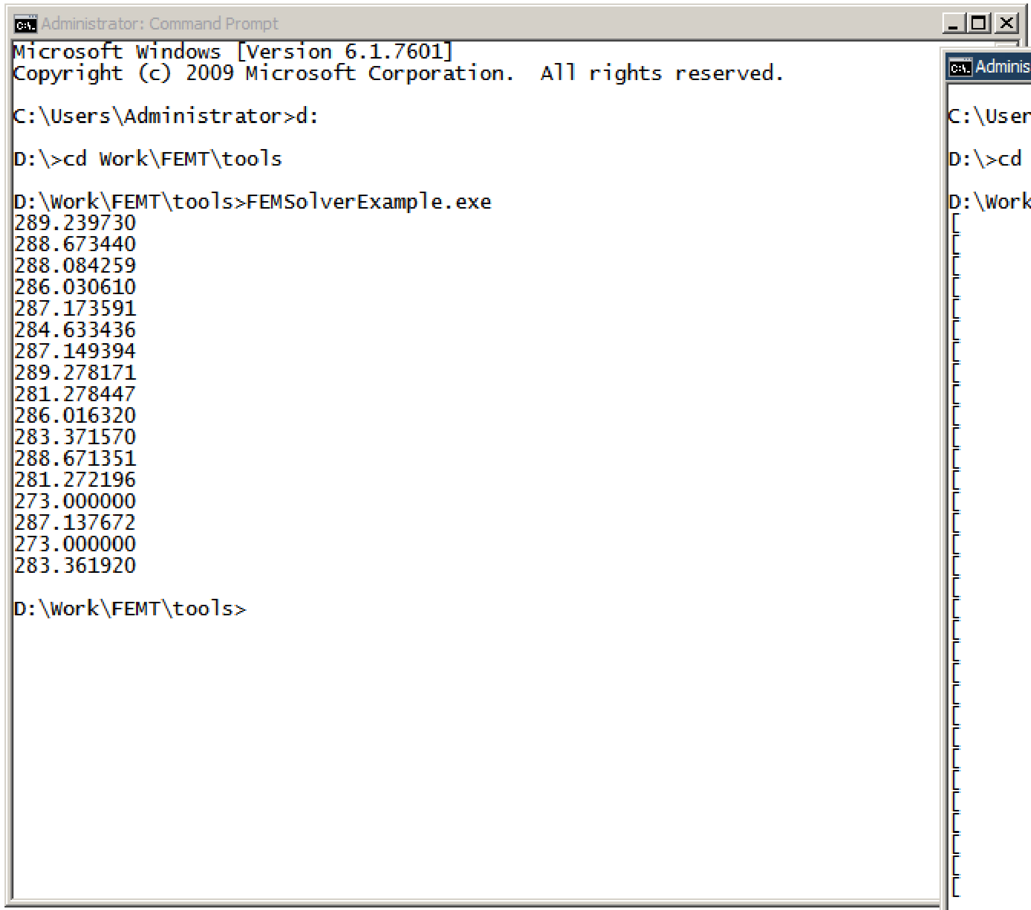
FEMSolver

FEMSolver is a program that solves finite element problems in parallel using the FEMT library on multi-core computers.

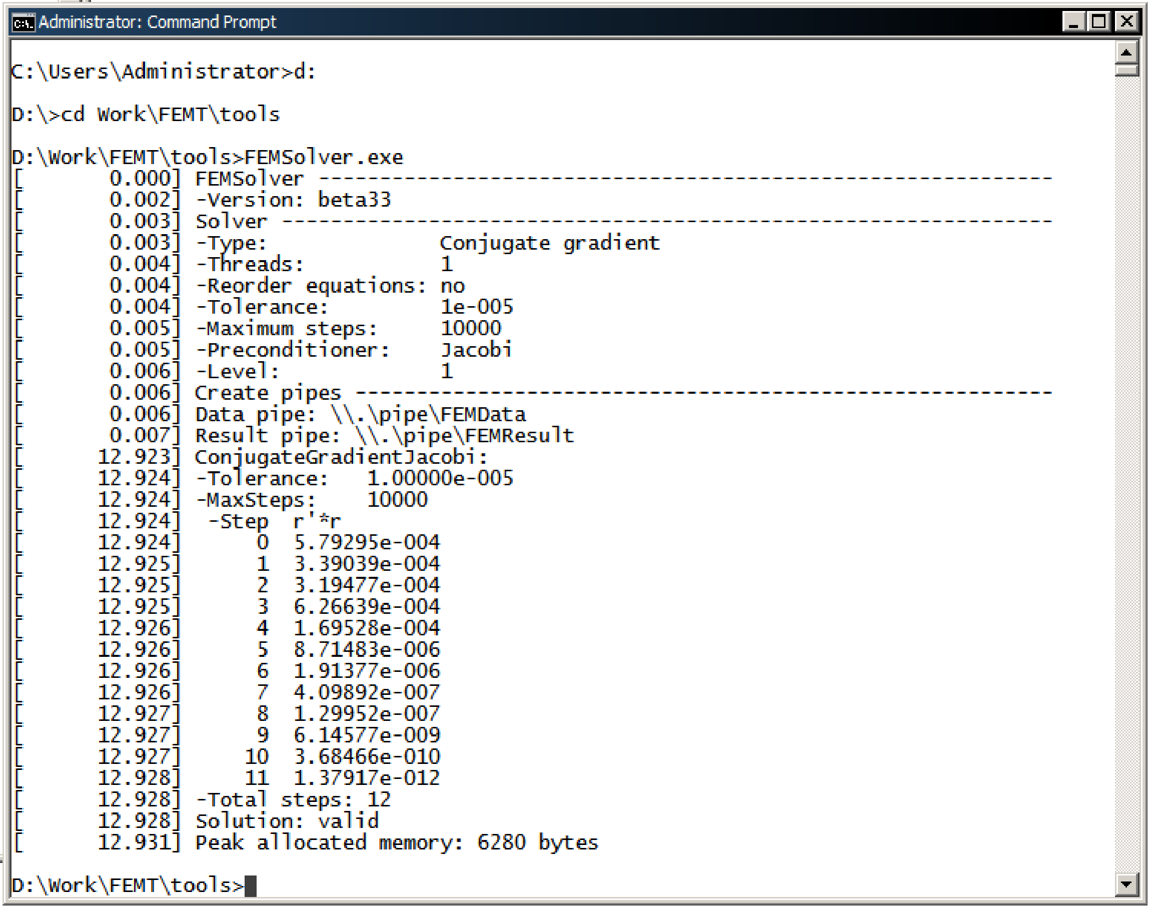
It uses a very simple interface using pipes. A pipe is an object of the operationg system that can be accessed like a file but does not write data to the disk, is a fast way to communicate running programs.



This flexible schema allows an used using any programming languaje (C/C++, Fortran, Python, C#, Java, etc.) to solve large systems of equations resulting from finite element discretizations. It can use any of the solvers of the FEMT library.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>d:

D:\>cd Work\FEMT\tools

D:\Work\FEMT\tools>FEMSolverExample.exe
289.239730
288.673440
288.084259
286.030610
287.173591
284.633436
287.149394
289.278171
281.278447
286.016320
283.371570
288.671351
281.272196
273.000000
287.137672
273.000000
283.361920

D:\Work\FEMT\tools>
```



```
C:\Users\Administrator>d:

D:\>cd Work\FEMT\tools

D:\Work\FEMT\tools>FEMSolver.exe
[        0.000] FEMSolver ------------------------------------------------------------
[        0.002] -Version: beta33
[        0.003] Solver ---------------------------------------------------------------
[        0.003] -Type:               Conjugate gradient
[        0.004] -Threads:            1
[        0.004] -Reorder equations:  no
[        0.004] -Tolerance:          1e-005
[        0.005] -Maximum steps:      10000
[        0.005] -Preconditioner:     Jacobi
[        0.006] -Level:              1
[        0.006] Create pipes ---------------------------------------------------------
[        0.006] Data pipe: \\.\pipe\FEMData
[        0.007] Result pipe: \\.\pipe\FEMResult
[       12.923] ConjugateGradientJacobi:
[       12.924] -Tolerance:    1.00000e-005
[       12.924] -MaxSteps:     10000
[       12.924]  -Step   r'*r
[       12.924]      0  5.79295e-004
[       12.925]      1  3.39039e-004
[       12.925]      2  3.19477e-004
[       12.925]      3  6.26639e-004
[       12.926]      4  1.69528e-004
[       12.926]      5  8.71483e-006
[       12.926]      6  1.91377e-006
[       12.926]      7  4.09892e-007
[       12.927]      8  1.29952e-007
[       12.927]      9  6.14577e-009
[       12.927]     10  3.68466e-010
[       12.928]     11  1.37917e-012
[       12.928] -Total steps: 12
[       12.928] Solution: valid
[       12.931] Peak allocated memory: 6280 bytes

D:\Work\FEMT\tools>
```

If the matrix remains constant, FEMSolver can be used to efficiently solve multi-step problems, like dynamic deformations, transient heat diffusion, etc.

```fortran
      PROGRAM FEMSolverExample
      IMPLICIT NONE

C Commands
      INTEGER command_end, command_set_connectivity, command_fill_A, command_se
      PARAMETER (
     . command_end            = 0,
     . command_set_connectivity = 1,
     . command_fill_A         = 2,
     . command_set_Ae         = 3,
     . command_set_all_Ae = 4,
     . command_set_x          = 5,
     . command_set_b          = 6,
     . command_set_fixed    = 7,
     . command_solver_init = 8,
     . command_solver_run = 9)
C Number of elements
      INTEGER*4 E /21/
C Number of nodes
      INTEGER*4 M /17/
C Element type (2=Triangle, 3=Quadrilateral, 4=Tetrahedra, 5=Hexahedra)
      INTEGER*4 T /2/
C Nodes per element
      INTEGER*4 N /3/
C Degrees of freedom
      INTEGER*4 D /1/
C Connectivity
      INTEGER*4 connectivity(3*21) /
     . 3, 8, 5,                  3, 5, 1,
     . 5, 8, 12,                 7, 2, 4,
     . 7, 4, 11,                  4, 2, 1,
     . 17, 16, 13,               17, 13, 15,
     . 13, 16, 14,               12, 15, 10,
     . 10, 15, 13,               12, 10, 5,
     . 14, 11, 9,                9, 11, 4,
     . 14, 9, 13,                10, 13, 6,
     . 6, 13, 9,                 10, 6, 5,
     . 6, 9, 4,                  5, 6, 4,
     . 1, 5, 4/
```

```c
#include <stdio.h>
int main()
{
// Commands
    enum Command
    {
        command_end             = 0,
        command_set_connectivity = 1,
        command_fill_A          = 2,
        command_set_Ae          = 3,
        command_set_all_Ae  = 4,
        command_set_x           = 5,
        command_set_b           = 6,
        command_set_fixed     = 7,
        command_solver_init  = 8,
        command_solver_run  = 9};
// Number of elements
    int E = 21;
// Number of nodes
    int M = 17;
// Element type (2=Triangle, 3=Quadrilateral, 4=Tetrahedra, 5=Hexahedra)
    int T = 2;
// Nodes per element
    int N = 3;
// Degrees of freedom
    int D = 1;
// Connectivity
    int connectivity[21*3] = {
        3, 8, 5,                    3, 5, 1,
        5, 8, 12,                   7, 2, 4,
        7, 4, 11,                   4, 2, 1,
        17, 16, 13,                 17, 13, 15,
        13, 16, 14,                 12, 15, 10,
        10, 15, 13,                 12, 10, 5,
        14, 11, 9,                  9, 11, 4,
        14, 9, 13,                  10, 13, 6,
        6, 13, 9,                   10, 6, 5,
        6, 9, 4,                    5, 6, 4,
        1, 5, 4};
```

```fortran
C Elemental matrices
    REAL*8 Ke(3*3,21) /
   . 6.47e-5, -2.62e-5, -3.85e-5, -2.62e-5, 6.47e-5, -3.85e-5, -3.85e-5, -3.85e-5, 7.70e-5
   . 1.19e-4, -6.49e-5, -5.46e-5, -6.49e-5, 6.45e-5, 3.33e-7, -5.46e-5, 3.33e-7, 5.42e-5
   . 6.45e-5, -6.49e-5, 3.33e-7, -6.49e-5, 1.19e-4, -5.46e-5, 3.33e-7, -5.46e-5, 5.42e-5
   . 6.47e-5, -2.62e-5, -3.85e-5, -2.62e-5, 6.47e-5, -3.85e-5, -3.85e-5, -3.85e-5, 7.70e-5
   . 1.19e-4, -6.49e-5, -5.46e-5, -6.49e-5, 6.45e-5, 3.33e-7, -5.46e-5, 3.33e-7, 5.42e-5
   . 6.45e-5, -6.49e-5, 3.33e-7, -6.49e-5, 1.19e-4, -5.46e-5, 3.33e-7, -5.46e-5, 5.42e-5
   . 6.47e-5, -2.62e-5, -3.85e-5, -2.62e-5, 6.47e-5, -3.85e-5, -3.85e-5, -3.85e-5, 7.70e-5
   . 1.19e-4, -6.49e-5, -5.46e-5, -6.49e-5, 6.45e-5, 3.33e-7, -5.46e-5, 3.33e-7, 5.42e-5
   . 6.45e-5, -6.49e-5, 3.33e-7, -6.49e-5, 1.19e-4, -5.46e-5, 3.33e-7, -5.46e-5, 5.42e-5
   . 6.47e-5, -2.62e-5, -3.85e-5, -2.62e-5, 6.47e-5, -3.85e-5, -3.85e-5, -3.85e-5, 7.70e-5
   . 1.18e-4, -6.49e-5, -5.33e-5, -6.49e-5, 6.52e-5, -3.33e-7, -5.33e-5, -3.33e-7, 5.37e-5
   . 6.52e-5, -6.49e-5, -3.33e-7, -6.49e-5, 1.18e-4, -5.33e-5, -3.33e-7, -5.33e-5, 5.37e-5
   . 6.47e-5, -2.62e-5, -3.85e-5, -2.62e-5, 6.47e-5, -3.85e-5, -3.85e-5, -3.85e-5, 7.70e-5
   . 1.18e-4, -6.49e-5, -5.33e-5, -6.49e-5, 6.52e-5, -3.33e-7, -5.33e-5, -3.33e-7, 5.37e-5
   . 6.52e-5, -6.49e-5, -3.33e-7, -6.49e-5, 1.18e-4, -5.33e-5, -3.33e-7, -5.33e-5, 5.37e-5
   . 6.44e-5, -2.56e-5, -3.89e-5, -2.56e-5, 6.44e-5, -3.89e-5, -3.89e-5, -3.89e-5, 7.78e-5
   . 7.67e-5, -3.96e-5, -3.72e-5, -3.96e-5, 6.60e-5, -2.64e-5, -3.72e-5, -2.64e-5, 6.36e-5
   . 6.60e-5, -3.96e-5, -2.64e-5, -3.96e-5, 7.67e-5, -3.72e-5, -2.64e-5, -3.72e-5, 6.36e-5
   . 7.49e-5, -3.85e-5, -3.65e-5, -3.85e-5, 6.64e-5, -2.80e-5, -3.65e-5, -2.80e-5, 6.45e-5
   . 6.05e-5, -6.76e-5, 7.08e-6, -6.76e-5, 1.33e-4, -6.58e-5, 7.08e-6, -6.58e-5, 5.87e-5
   . 7.18e-5, -3.59e-5, -3.59e-5, -3.59e-5, 6.67e-5, -3.07e-5, -3.59e-5, -3.07e-5, 6.67e-5/
C Vector with constrain values
    REAL*8 x(17) /0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 273, 0, 273, 0/
C Vector of independent terms
    REAL*8 b(17) /2.82e-4, 2.82e-4, 0, 0, 0, 0, 2.82e-4, 2.82e-4, 0, 0, 2.82e-4, 2.82e-4, 0,
C Vector that indicates where the constrains are
    INTEGER*1 fixed(17) /0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0/
C Result vector
    REAL*8 r(17)
C Variable to indicate command
    INTEGER*1 command
C Data pipe
    INTEGER*4 FEMData /100/
C Result pipe
    INTEGER*4 FEMResult /101/
C Indexes
    INTEGER*4 i, j
```

```c
// Elemental matrices
    double Ke[21][3*3] ={
    {6.47e-5,-2.62e-5,-3.85e-5,-2.62e-5,6.47e-5,-3.85e-5,-3.85e-5,-3.85e-5,7.70e-5},
    {1.19e-4,-6.49e-5,-5.46e-5,-6.49e-5,6.45e-5,3.33e-7,-5.46e-5,3.33e-7,5.42e-5},
    {6.45e-5,-6.49e-5,3.33e-7,-6.49e-5,1.19e-4,-5.46e-5,3.33e-7,-5.46e-5,5.42e-5},
    {6.47e-5,-2.62e-5,-3.85e-5,-2.62e-5,6.47e-5,-3.85e-5,-3.85e-5,-3.85e-5,7.70e-5},
    {1.19e-4,-6.49e-5,-5.46e-5,-6.49e-5,6.45e-5,3.33e-7,-5.46e-5,3.33e-7,5.42e-5},
    {6.45e-5,-6.49e-5,3.33e-7,-6.49e-5,1.19e-4,-5.46e-5,3.33e-7,-5.46e-5,5.42e-5},
    {6.47e-5,-2.62e-5,-3.85e-5,-2.62e-5,6.47e-5,-3.85e-5,-3.85e-5,-3.85e-5,7.70e-5},
    {1.19e-4,-6.49e-5,-5.46e-5,-6.49e-5,6.45e-5,3.33e-7,-5.46e-5,3.33e-7,5.42e-5},
    {6.45e-5,-6.49e-5,3.33e-7,-6.49e-5,1.19e-4,-5.46e-5,3.33e-7,-5.46e-5,5.42e-5},
    {6.47e-5,-2.62e-5,-3.85e-5,-2.62e-5,6.47e-5,-3.85e-5,-3.85e-5,-3.85e-5,7.70e-5},
    {1.18e-4,-6.49e-5,-5.33e-5,-6.49e-5,6.52e-5,-3.33e-7,-5.33e-5,-3.33e-7,5.37e-5},
    {6.52e-5,-6.49e-5,-3.33e-7,-6.49e-5,1.18e-4,-5.33e-5,-3.33e-7,-5.33e-5,5.37e-5},
    {6.47e-5,-2.62e-5,-3.85e-5,-2.62e-5,6.47e-5,-3.85e-5,-3.85e-5,-3.85e-5,7.70e-5},
    {1.18e-4,-6.49e-5,-5.33e-5,-6.49e-5,6.52e-5,-3.33e-7,-5.33e-5,-3.33e-7,5.37e-5},
    {6.52e-5,-6.49e-5,-3.33e-7,-6.49e-5,1.18e-4,-5.33e-5,-3.33e-7,-5.33e-5,5.37e-5},
    {6.44e-5,-2.56e-5,-3.89e-5,-2.56e-5,6.44e-5,-3.89e-5,-3.89e-5,-3.89e-5,7.78e-5},
    {7.67e-5,-3.96e-5,-3.72e-5,-3.96e-5,6.60e-5,-2.64e-5,-3.72e-5,-2.64e-5,6.36e-5},
    {6.60e-5,-3.96e-5,-2.64e-5,-3.96e-5,7.67e-5,-3.72e-5,-2.64e-5,-3.72e-5,6.36e-5},
    {7.49e-5,-3.85e-5,-3.65e-5,-3.85e-5,6.64e-5,-2.80e-5,-3.65e-5,-2.80e-5,6.45e-5},
    {6.05e-5,-6.76e-5,7.08e-6,-6.76e-5,1.33e-4,-6.58e-5,7.08e-6,-6.58e-5,5.87e-5},
    {7.18e-5,-3.59e-5,-3.59e-5,-3.59e-5,6.67e-5,-3.07e-5,-3.59e-5,-3.07e-5,6.67e-5}};
// Vector with constrain values
    double x[17] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 273, 0, 273, 0};
// Vector of independent terms
    double b[17] = {2.82e-4, 2.82e-4, 0, 0, 0, 0, 2.82e-4, 2.82e-4, 0, 0, 2.82e-4, 2.82e-4,
// Vector that indicates where the constrains are
    bool fixed[17] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0};
// Result vector
    double r[17];
// Variable to indicate command
    char command;
// Data pipe
    FILE* FEMData;
// Result pipe
    FILE* FEMResult;
// Indexes
    int i;
```

```fortran
C Paths for pipes
#ifdef WIN32
    CHARACTER*128 dataname /'\\.\pipe\FEMData'/
    CHARACTER*128 resultname /'\\.\pipe\FEMResult'/
#else
    CHARACTER*128 dataname /'/tmp/FEMData'/
    CHARACTER*128 resultname /'/tmp/FEMResult'/
#endif

C Open data and result pipes
    OPEN(FEMData, FILE=dataname, ACCESS='STREAM')
    OPEN(FEMResult, FILE=resultname, ACCESS='STREAM')

C Send mesh data
    command = command_set_connectivity
    WRITE(FEMData) command
C Send number of nodes
    WRITE(FEMData) M
C Send number of elements
    WRITE(FEMData) E
C Send element type
    WRITE(FEMData) T
C Send nodes per element
    WRITE(FEMData) N
C Send degrees of freedom
    WRITE(FEMData) D
C Send connectivity
    WRITE(FEMData) (connectivity(i), i = 1, E*N)

C Send elemental matrices
    command = command_set_Ae;
    DO i = 1, E
     WRITE(FEMData) command
     WRITE(FEMData) i
     WRITE(FEMData) (Ke(j, i), j = 1, N*N)
    ENDDO

C Send vector with constrain values
    command = command_set_x
```

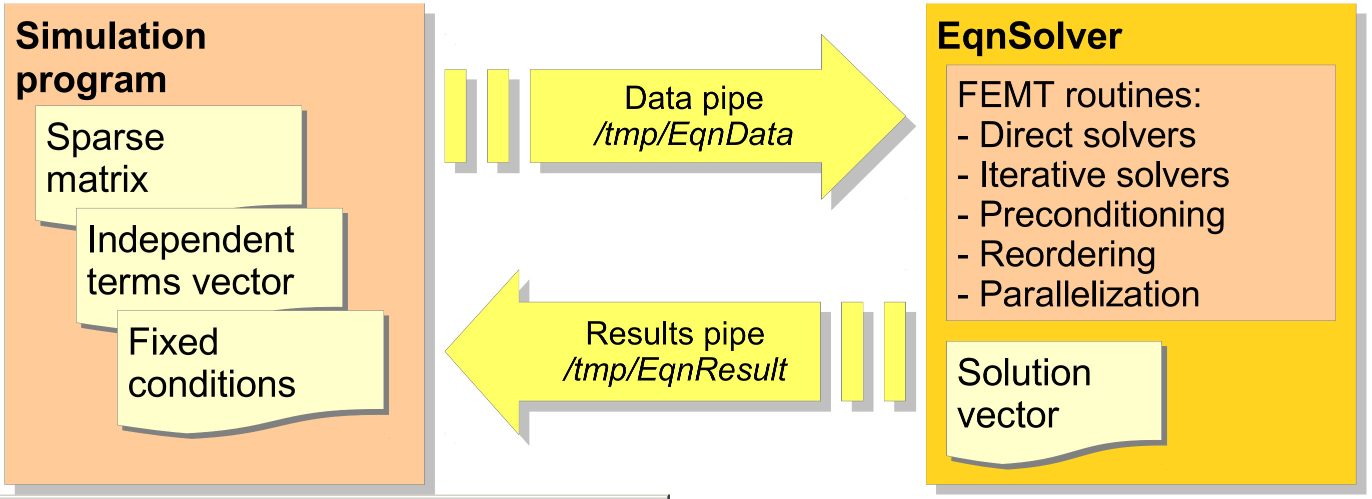
```c
// Names for the pipes
#ifdef WIN32
    const char* dataname = "\\\\.\\pipe\\FEMData";
    const char* resultname = "\\\\.\\pipe\\FEMResult";
#else
    const char* dataname = "/tmp/FEMData";
    const char* resultname = "/tmp/FEMResult";
#endif

// Open data and result pipes
    FEMData = fopen(dataname, "wb");
    FEMResult = fopen(resultname, "rb");

// Send mesh data
    command = command_set_connectivity;
    fwrite(&command, 1, 1, FEMData);
    // Send number of nodes
    fwrite(&M, sizeof(int), 1, FEMData);
    // Send number of elements
    fwrite(&E, sizeof(int), 1, FEMData);
    // Send element type
    fwrite(&T, sizeof(int), 1, FEMData);
    // Send nodes per element
    fwrite(&N, sizeof(int), 1, FEMData);
    // Send degrees of freedom
    fwrite(&D, sizeof(int), 1, FEMData);
    // Send connectivity
    fwrite(connectivity, sizeof(int), E*N, FEMData);

// Send elemental matrices
    command = command_set_Ae;
    for (i = 1; i <= E; ++i) {
        fwrite(&command, 1, 1, FEMData);
        fwrite(&i, sizeof(int), 1, FEMData);
        fwrite(Ke[i - 1], sizeof(double), N*N, FEMData);
    }

// Send vector with constrain values
    command = command_set_x;
```

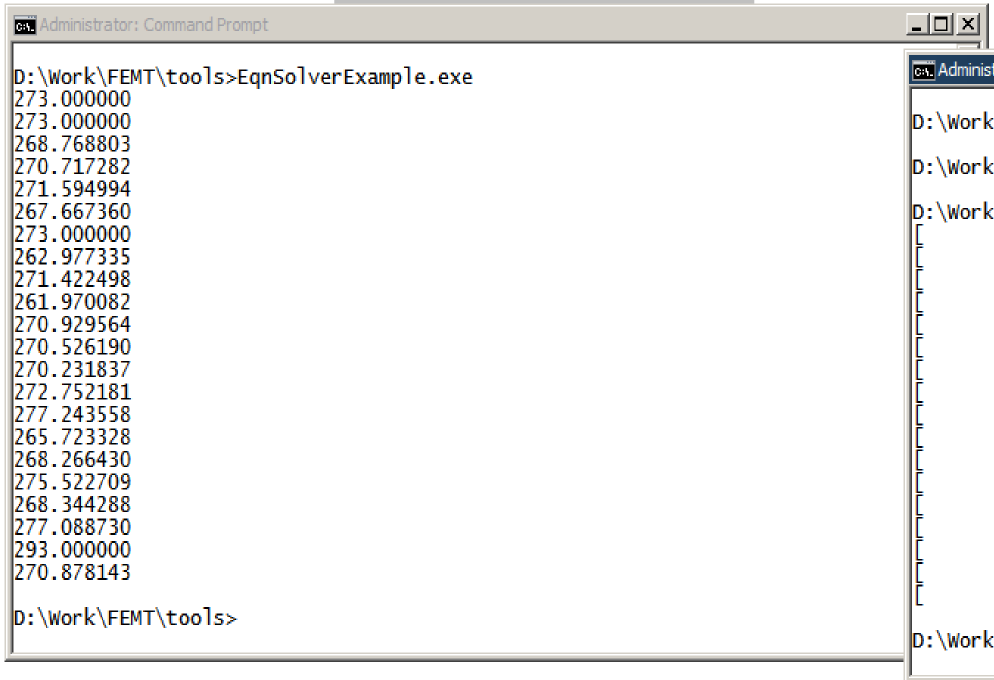
```fortran
      WRITE(FEMData) command
      WRITE(FEMData) (x(j), j = 1, M)

C Send vector of independent terms
      command = command_set_b
      WRITE(FEMData) command, (b(j), j = 1, M)


C Send vector that indicates where the constrains are
      command = command_set_fixed
      WRITE(FEMData) command, (fixed(j), j = 1, M)


C Initialize solver
      command = command_solver_init
      WRITE(FEMData) command

C Run solver and read result
      command = command_solver_run
      WRITE(FEMData) command
      FLUSH(FEMData)
      READ(FEMResult) (r(j), j = 1, M)

C Display result
      DO i = 1, M
       WRITE(6, *) r(i)
      ENDDO

C Send end session command
      command = command_end
      WRITE(FEMData) command
      FLUSH(FEMData)

C Close pipes
      CLOSE(FEMResult)
      CLOSE(FEMData)

      END
```

```c
    fwrite(&command, 1, 1, FEMData);
    fwrite(x, sizeof(double), M, FEMData);

// Send vector of independent terms
    command = command_set_b;
    fwrite(&command, 1, 1, FEMData);
    fwrite(b, sizeof(double), M, FEMData);

// Send vector that indicates where the constrains are
    command = command_set_fixed;
    fwrite(&command, 1, 1, FEMData);
    fwrite(fixed, sizeof(bool), M, FEMData);

// Initialize solver
    command = command_solver_init;
    fwrite(&command, 1, 1, FEMData);

// Run solver and read result
    command = command_solver_run;
    fwrite(&command, 1, 1, FEMData);
    fflush(FEMData);
    fread(r, sizeof(double), M, FEMResult);

// Display result
    for (i = 0; i < M; ++i) {
        printf("%f\n", r[i]);
    }

// Send end session command
    command = command_end;
    fwrite(&command, 1, 1, FEMData);
    fflush(FEMData);

// Close pipes
    fclose(FEMResult);
    fclose(FEMData);

    return 0;
}
```

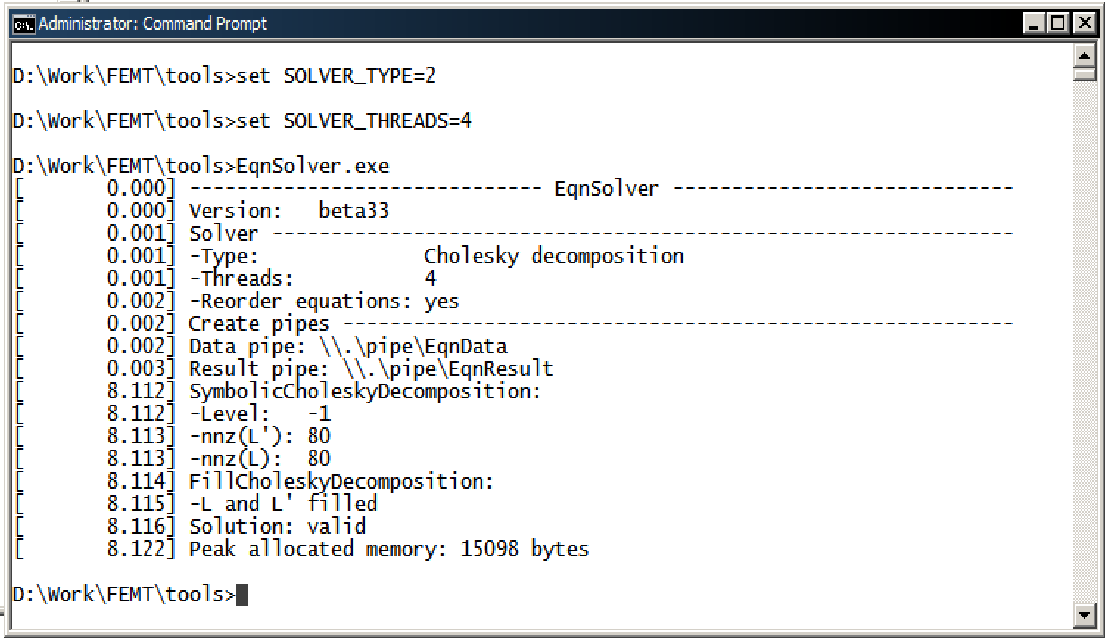
EqnSolver

EqnSolver works in a similar way than FEMSolver, but it takes as input a sparse matrix.





```
D:\Work\FEMT\tools>EqnSolverExample.exe
273.000000
273.000000
268.768803
270.717282
271.594994
267.667360
273.000000
262.977335
271.422498
261.970082
270.929564
270.526190
270.231837
272.752181
277.243558
265.723328
268.266430
275.522709
268.344288
277.088730
293.000000
270.878143

D:\Work\FEMT\tools>
```



```
D:\Work\FEMT\tools>set SOLVER_TYPE=2

D:\Work\FEMT\tools>set SOLVER_THREADS=4

D:\Work\FEMT\tools>EqnSolver.exe
[       0.000] ------------------------------ EqnSolver ----------------------------
[       0.000] Version:   beta33
[       0.001] Solver ---------------------------------------------------------------
[       0.001] -Type:             Cholesky decomposition
[       0.001] -Threads:          4
[       0.002] -Reorder equations: yes
[       0.002] Create pipes ---------------------------------------------------------
[       0.002] Data pipe: \\.\pipe\EqnData
[       0.003] Result pipe: \\.\pipe\EqnResult
[       8.112] SymbolicCholeskyDecomposition:
[       8.112] -Level:   -1
[       8.113] -nnz(L'): 80
[       8.113] -nnz(L):  80
[       8.114] FillCholeskyDecomposition:
[       8.115] -L and L' filled
[       8.116] Solution: valid
[       8.122] Peak allocated memory: 15098 bytes

D:\Work\FEMT\tools>
```

```
      PROGRAM EqnSolverExample

      IMPLICIT NONE
C Commands
      INTEGER command_end, command_set_size, command_set_row, command_set_x, c
      PARAMETER (
     . command_end       = 0,
     . command_set_size   = 1,
     . command_set_row    = 2,
     . command_set_x      = 3,
     . command_set_b      = 4,
     . command_set_fixed  = 5,
     . command_solver_init = 6,
     . command_solver_run  = 7)
C Number of equations
      INTEGER*4 M /22/
C Matrix values
      REAL*8 values(120) /
     . 1.2326e-4, -3.6051e-5, -3.3997e-5, -5.3215e-5,
     . -3.6051e-5, 1.7051e-4, -4.3495e-5, -6.0878e-5, -3.0087e-5,
     . -3.3997e-5, 1.7032e-4, -5.5748e-5, -4.5665e-5, -3.4909e-5,
     . -5.3215e-5, -4.3495e-5, -5.5748e-5, 3.4166e-4, -7.1143e-5, -6.8663e-5, -4.9398e-5,
     . -6.0878e-5, -7.1143e-5, 3.4267e-4, -5.8604e-5, -4.4280e-5, -4.3102e-5, -6.4661e-5,
     . -4.5665e-5, -6.8663e-5, 3.4155e-4, -5.6230e-5, -4.4270e-5, -5.8971e-5, -6.7748e-5,
     . -3.0087e-5, -5.8604e-5, 1.2279e-4, -3.4102e-5,
     . -3.4909e-5, -5.6230e-5, 1.2292e-4, -3.1777e-5,
     . -4.9398e-5, -4.4280e-5, -4.4270e-5, 3.2961e-4, -4.7225e-5, -5.2776e-5, -5.2312e-5, -
     . -5.8971e-5, -3.1777e-5, 1.7029e-4, -4.4635e-5, -3.4909e-5,
     . -4.3102e-5, -3.4102e-5, 1.7088e-4, -6.6465e-5, -2.7215e-5,
     . -6.4661e-5, -4.7225e-5, -6.6465e-5, 3.4234e-4, -5.0918e-5, -6.2797e-5, -5.0270e-5,
     . -6.7748e-5, -5.2776e-5, -4.4635e-5, 3.4104e-4, -5.8173e-5, -5.6230e-5, -6.1480e-5,
     . -5.2312e-5, -5.0918e-5, 3.4252e-4, -7.1496e-5, -5.5712e-5, -5.1108e-5, -6.0978e-5,
     . -3.9348e-5, -5.8173e-5, -7.1496e-5, 3.4468e-4, -4.5355e-5, -6.3924e-5, -6.6386e-5,
     . -3.4909e-5, -5.6230e-5, 1.2292e-4, -3.1777e-5,
     . -2.7215e-5, -6.2797e-5, 1.2287e-4, -3.2859e-5,
     . -6.1480e-5, -4.5355e-5, -3.1777e-5, 1.7057e-4, -3.1963e-5,
     . -5.0270e-5, -5.5712e-5, -3.2859e-5, 1.7088e-4, -3.2040e-5,
     . -5.1108e-5, -6.3924e-5, 1.6950e-4, -2.4862e-5, -2.9603e-5,
     . -6.6386e-5, -3.1963e-5, -2.4862e-5, 1.2321e-4,
```

```
#include <stdio.h>

int main()
{
// Commands
  enum Command {
    command_end       = 0,
    command_set_size   = 1,
    command_set_row    = 2,
    command_set_x      = 3,
    command_set_b      = 4,
    command_set_fixed  = 5,
    command_solver_init = 6,
    command_solver_run  = 7};
// Number of equations
  int M = 22;
// Matrix values
  double values[120] = {
    1.2326e-4, -3.6051e-5, -3.3997e-5, -5.3215e-5,
    -3.6051e-5, 1.7051e-4, -4.3495e-5, -6.0878e-5, -3.0087e-5,
    -3.3997e-5, 1.7032e-4, -5.5748e-5, -4.5665e-5, -3.4909e-5,
    -5.3215e-5, -4.3495e-5, -5.5748e-5, 3.4166e-4, -7.1143e-5, -6.8663e-5, -4.9398e-5,
    -6.0878e-5, -7.1143e-5, 3.4267e-4, -5.8604e-5, -4.4280e-5, -4.3102e-5, -6.4661e-5,
    -4.5665e-5, -6.8663e-5, 3.4155e-4, -5.6230e-5, -4.4270e-5, -5.8971e-5, -6.7748e-5,
    -3.0087e-5, -5.8604e-5, 1.2279e-4, -3.4102e-5,
    -3.4909e-5, -5.6230e-5, 1.2292e-4, -3.1777e-5,
    -4.9398e-5, -4.4280e-5, -4.4270e-5, 3.2961e-4, -4.7225e-5, -5.2776e-5, -5.2312e-5, -
    -5.8971e-5, -3.1777e-5, 1.7029e-4, -4.4635e-5, -3.4909e-5,
    -4.3102e-5, -3.4102e-5, 1.7088e-4, -6.6465e-5, -2.7215e-5,
    -6.4661e-5, -4.7225e-5, -6.6465e-5, 3.4234e-4, -5.0918e-5, -6.2797e-5, -5.0270e-5,
    -6.7748e-5, -5.2776e-5, -4.4635e-5, 3.4104e-4, -5.8173e-5, -5.6230e-5, -6.1480e-5,
    -5.2312e-5, -5.0918e-5, 3.4252e-4, -7.1496e-5, -5.5712e-5, -5.1108e-5, -6.0978e-5,
    -3.9348e-5, -5.8173e-5, -7.1496e-5, 3.4468e-4, -4.5355e-5, -6.3924e-5, -6.6386e-5,
    -3.4909e-5, -5.6230e-5, 1.2292e-4, -3.1777e-5,
    -2.7215e-5, -6.2797e-5, 1.2287e-4, -3.2859e-5,
    -6.1480e-5, -4.5355e-5, -3.1777e-5, 1.7057e-4, -3.1963e-5,
    -5.0270e-5, -5.5712e-5, -3.2859e-5, 1.7088e-4, -3.2040e-5,
    -5.1108e-5, -6.3924e-5, 1.6950e-4, -2.4862e-5, -2.9603e-5,
    -6.6386e-5, -3.1963e-5, -2.4862e-5, 1.2321e-4,
```

```fortran
     . -6.0978e-5, -3.2040e-5, -2.9603e-5, 1.2262e-4/
C Matrix indexes
      INTEGER*4 indexes(120) /
     . 1, 2, 3, 4,
     . 1, 2, 4, 5, 7,
     . 1, 3, 4, 6, 8,
     . 1, 2, 3, 4, 5, 6, 9,
     . 2, 4, 5, 7, 9, 11, 12,
     . 3, 4, 6, 8, 9, 10, 13,
     . 2, 5, 7, 11,
     . 3, 6, 8, 10,
     . 4, 5, 6, 9, 12, 13, 14, 15,
     . 6, 8, 10, 13, 16,
     . 5, 7, 11, 12, 17,
     . 5, 9, 11, 12, 14, 17, 19,
     . 6, 9, 10, 13, 15, 16, 18,
     . 9, 12, 14, 15, 19, 20, 22,
     . 9, 13, 14, 15, 18, 20, 21,
     . 10, 13, 16, 18,
     . 11, 12, 17, 19,
     . 13, 15, 16, 18, 21,
     . 12, 14, 17, 19, 22,
     . 14, 15, 20, 21, 22,
     . 15, 18, 20, 21,
     . 14, 19, 20, 22/
C Row sizes
      INTEGER*4 count(22) /4, 5, 5, 7, 7, 7, 4, 4, 8, 5, 5, 7, 7, 7, 7, 4, 4, 5, 5, 5, 4, 4/
C Vector with constrain values
      REAL*8 x(22) /273, 273, 0, 0, 0, 0, 273, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 293, 0/
C Vector of independent terms
      REAL*8 b(22) /0, 0, 0, 0, 0, 0, 0, -4.3388e-4, 0, -8.6776e-4, 0, 0, 0, 0, 0, -4.3388e-4, -2.
C Vector that indicates where the constrains are
      INTEGER*1 fixed(22) /1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0/
C Result vector
      REAL*8 r(22)
C Variable to indicate command
      INTEGER*1 command
C Data pipe
      INTEGER*4 EqnData /100/
```

```c
    -6.0978e-5, -3.2040e-5, -2.9603e-5, 1.2262e-4};
// Matrix indexes
  int indexes[120]= {
    1, 2, 3, 4,
    1, 2, 4, 5, 7,
    1, 3, 4, 6, 8,
    1, 2, 3, 4, 5, 6, 9,
    2, 4, 5, 7, 9, 11, 12,
    3, 4, 6, 8, 9, 10, 13,
    2, 5, 7, 11,
    3, 6, 8, 10,
    4, 5, 6, 9, 12, 13, 14, 15,
    6, 8, 10, 13, 16,
    5, 7, 11, 12, 17,
    5, 9, 11, 12, 14, 17, 19,
    6, 9, 10, 13, 15, 16, 18,
    9, 12, 14, 15, 19, 20, 22,
    9, 13, 14, 15, 18, 20, 21,
    10, 13, 16, 18,
    11, 12, 17, 19,
    13, 15, 16, 18, 21,
    12, 14, 17, 19, 22,
    14, 15, 20, 21, 22,
    15, 18, 20, 21,
    14, 19, 20, 22};
// Row sizes
  int count[22] = {4, 5, 5, 7, 7, 7, 4, 4, 8, 5, 5, 7, 7, 7, 7, 4, 4, 5, 5, 5, 4, 4};
// Vector with constrain values
  double x[22] = {273, 273, 0, 0, 0, 0, 273, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 293, 0};
// Vector of independent terms
  double b[22] = {0, 0, 0, 0, 0, 0, 0, -4.3388e-4, 0, -8.6776e-4, 0, 0, 0, 0, 0, -4.3388e-4, -2.
// Vector that indicates where the constrains are
  bool fixed[22] = {1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};
// Result vector
  double r[22];
// Variable to indicate command
  char command;
// Data pipe
  FILE* EqnData;
```

```fortran
C Result pipe
    INTEGER*4 EqnResult /101/
C Indexes
    INTEGER*4 i, j, j1, jn
C Names for the pipes
#ifdef WIN32
    CHARACTER*128 dataname /'\\.\pipe\EqnData'/
    CHARACTER*128 resultname /'\\.\pipe\EqnResult'/
#else
    CHARACTER*128 dataname /'/tmp/EqnData'/
    CHARACTER*128 resultname /'/tmp/EqnResult'/
#endif

C Open data and result pipes
    OPEN(EqnData, FILE=dataname, ACCESS='STREAM')
    OPEN(EqnResult, FILE=resultname, ACCESS='STREAM')

C Send number of equations
    command = command_set_size
    WRITE(EqnData) command
    WRITE(EqnData) M

C Send rows
    command = command_set_row
    j1 = 1
    DO i = 1, M
     jn = j1 + count(i) - 1
     WRITE(EqnData) command
     WRITE(EqnData) i,
     WRITE(EqnData) count(i)
     WRITE(EqnData) (indexes(j), j = j1, jn)
     WRITE(EqnData) (values(j), j = j1, jn)
     j1 = jn + 1
    ENDDO

C Send vector with constrain values
    command = command_set_x
    WRITE(EqnData) command
    WRITE(EqnData) (x(j), j = 1, M)
```

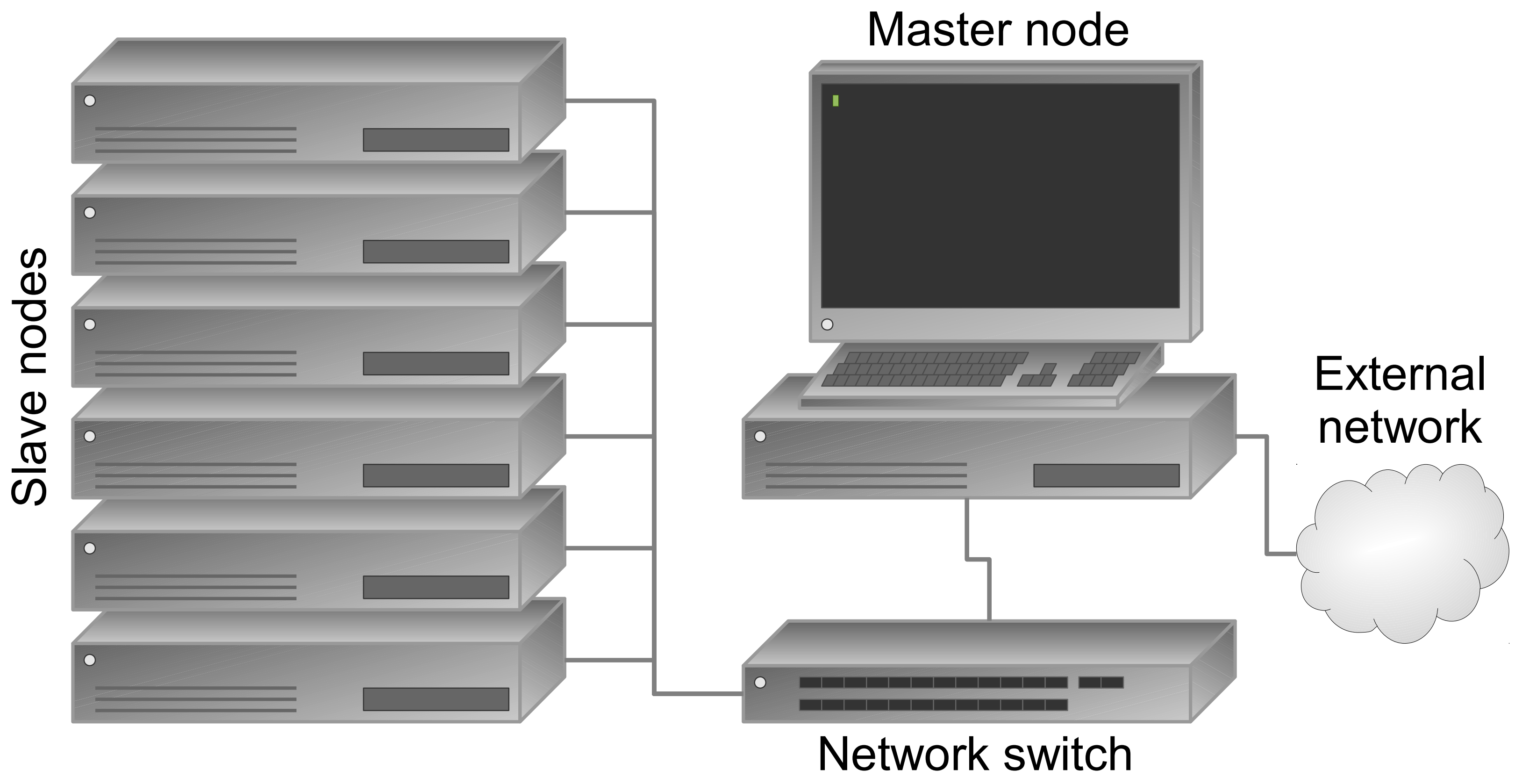
```c
// Result pipe
  FILE* EqnResult;
// Indexes
  int i, j1;
// Names for the pipes
#ifdef WIN32
  const char* dataname = "\\\\.\\pipe\\EqnData";
  const char* resultname = "\\\\.\\pipe\\EqnResult";
#else
  const char* dataname = "/tmp/EqnData";
  const char* resultname = "/tmp/EqnResult";
#endif

// Open data and result pipes
  EqnData = fopen(dataname, "wb");
  EqnResult = fopen(resultname, "rb");

// Send number of equations
  command = command_set_size;
  fwrite(&command, 1, 1, EqnData);
  fwrite(&M, sizeof(int), 1, EqnData);

// Send rows
  command = command_set_row;
  j1 = 0;
  for (i = 0; i < M; ++i) {
    int row = i + 1;
    fwrite(&command, 1, 1, EqnData);
    fwrite(&row, sizeof(int), 1, EqnData);
    fwrite(&count[i], sizeof(int), 1, EqnData);
    fwrite(indexes + j1, sizeof(int), count[i], EqnData);
    fwrite(values + j1, sizeof(double), count[i], EqnData);
    j1 += count[i];
  }

// Send vector with constrain values
  command = command_set_x;
  fwrite(&command, 1, 1, EqnData);
  fwrite(x, sizeof(double), M, EqnData);
```

```fortran
C Send vector of independent terms
      command = command_set_b
      WRITE(EqnData) command
      WRITE(EqnData) (b(j), j = 1, M)

C Send vector that indicates where the constrains are
      command = command_set_fixed
      WRITE(EqnData) command
      WRITE(EqnData) (fixed(j), j = 1, M)

C Initialize solver
      command = command_solver_init
      WRITE(EqnData) command

C Run solver and read result
      command = command_solver_run
      WRITE(EqnData) command
      FLUSH(EqnData)
      READ(EqnResult) (r(j), j = 1, M)

C Display result
      DO i = 1, M
        WRITE(6, *) r(i)
      ENDDO

C Send end session command
      command = command_end
      WRITE(EqnData) command
      FLUSH(EqnData)

C Close pipes
      CLOSE(EqnResult)
      CLOSE(EqnData)

      END
```

```c
// Send vector of independent terms
  command = command_set_b;
  fwrite(&command, 1, 1, EqnData);
  fwrite(b, sizeof(double), M, EqnData);

// Send vector that indicates where the constrains are
  command = command_set_fixed;
  fwrite(&command, 1, 1, EqnData);
  fwrite(fixed, sizeof(bool), M, EqnData);

// Initialize solver
  command = command_solver_init;
  fwrite(&command, 1, 1, EqnData);

// Run solver and read result
  command = command_solver_run;
  fwrite(&command, 1, 1, EqnData);
  fflush(EqnData);
  fread(r, sizeof(double), M, EqnResult);

// Display result
  for (i = 0; i < M; ++i) {
    printf("%f\n", r[i]);
  }

// Send end session command
  command = command_end;
  fwrite(&command, 1, 1, EqnData);
  fflush(EqnData);

// Close pipes
  fclose(EqnResult);
  fclose(EqnData);

  return 0;
}
```

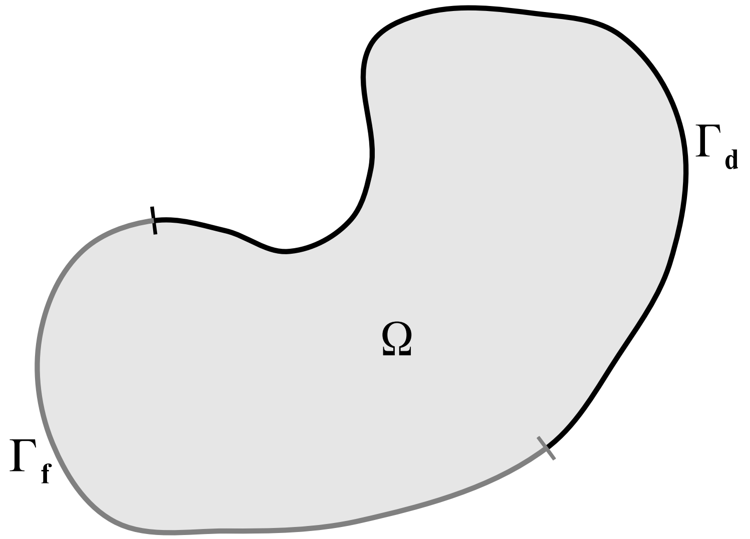
Parallelization in computer clusters

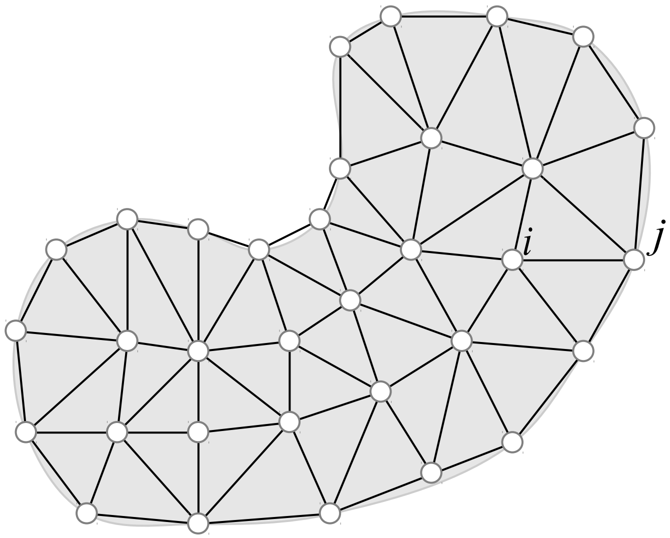
We developed a software program that runs in parallel in a Beowulf cluster [Ster95]. A Beowulf cluster consists of several multi-core computers (nodes) connected with a high speed network.

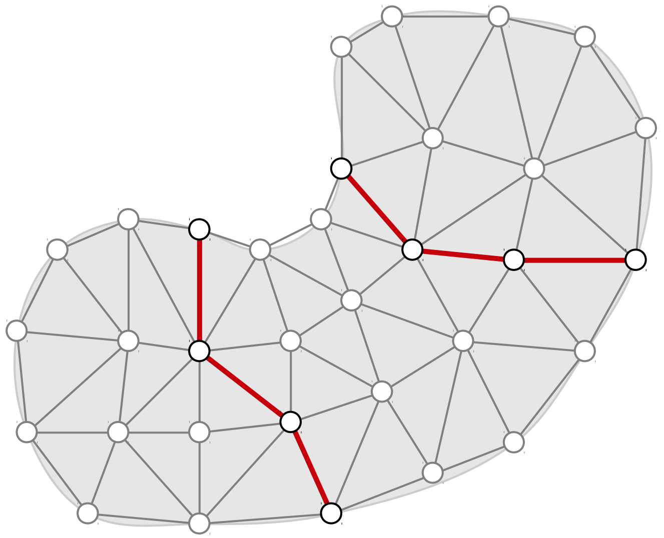


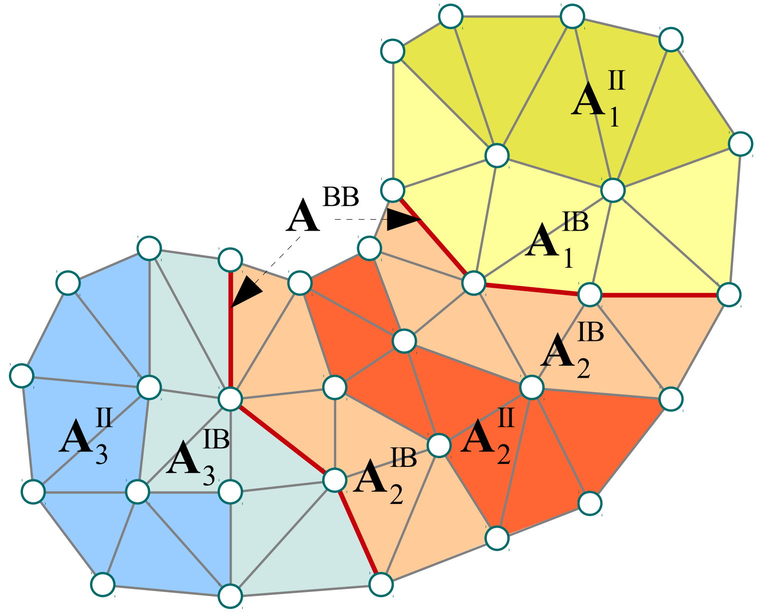
Schur substructuring method

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









$$\begin{pmatrix} \mathbf{A}_1^{II} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{IB} \\ \mathbf{0} & \mathbf{A}_2^{II} & \mathbf{0} & \mathbf{A}_2^{IB} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_3^{II} & \mathbf{A}_3^{IB} \\ \mathbf{A}_1^{BI} & \mathbf{A}_2^{BI} & \mathbf{A}_3^{BI} & \boxed{\mathbf{A}^{BB}} \end{pmatrix}$$

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

$$\begin{pmatrix} \mathbf{A}_1^{\mathrm{II}} & & \mathbf{0} & & & \mathbf{A}_1^{\mathrm{IB}} \\ & \mathbf{A}_2^{\mathrm{II}} & & & & \mathbf{A}_2^{\mathrm{IB}} \\ \mathbf{0} & & \mathbf{A}_3^{\mathrm{II}} & & & \mathbf{A}_3^{\mathrm{IB}} \\ \vdots & & & \ddots & & \vdots \\ & & & & \mathbf{A}_p^{\mathrm{II}} & \mathbf{A}_p^{\mathrm{IB}} \\ \mathbf{A}_1^{\mathrm{BI}} & \mathbf{A}_2^{\mathrm{BI}} & \mathbf{A}_3^{\mathrm{BI}} & \cdots & \mathbf{A}_p^{\mathrm{BI}} & \mathbf{A}^{\mathrm{BB}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1^{\mathrm{I}} \\ \mathbf{x}_2^{\mathrm{I}} \\ \mathbf{x}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{x}_p^{\mathrm{I}} \\ \mathbf{x}^{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1^{\mathrm{I}} \\ \mathbf{b}_2^{\mathrm{I}} \\ \mathbf{b}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{b}_p^{\mathrm{I}} \\ \mathbf{b}^{\mathrm{B}} \end{pmatrix}.$$

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,

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

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

$$\mathbf{x}_i^{\mathrm{I}} = \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \left(\mathbf{b}_i^{\mathrm{I}} - \mathbf{A}_i^{\mathrm{IB}} \mathbf{x}^{\mathrm{B}}\right).$$

By applying Gaussian elimination by blocks to eliminate terms in the last row,

$$\begin{pmatrix} \mathbf{A}_1^{\mathrm{II}} & \mathbf{0} & & & & \mathbf{A}_1^{\mathrm{IB}} \\ & \mathbf{A}_2^{\mathrm{II}} & & & & \mathbf{A}_2^{\mathrm{IB}} \\ \mathbf{0} & & \mathbf{A}_3^{\mathrm{II}} & & & \mathbf{A}_3^{\mathrm{IB}} \\ \vdots & & & \ddots & & \vdots \\ & & & & \mathbf{A}_p^{\mathrm{II}} & \mathbf{A}_p^{\mathrm{IB}} \\ 0 & \mathbf{A}_2^{\mathrm{BI}} & \mathbf{A}_3^{\mathrm{BI}} & \cdots & \mathbf{A}_p^{\mathrm{BI}} & \mathbf{A}^{\mathrm{BB}} - \mathbf{A}_1^{\mathrm{BI}} \left(\mathbf{A}_1^{\mathrm{II}}\right)^{-1} \mathbf{A}_1^{\mathrm{IB}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1^{\mathrm{I}} \\ \mathbf{x}_2^{\mathrm{I}} \\ \mathbf{x}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{x}_p^{\mathrm{I}} \\ \mathbf{x}^{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1^{\mathrm{I}} \\ \mathbf{b}_2^{\mathrm{I}} \\ \mathbf{b}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{b}_p^{\mathrm{I}} \\ \mathbf{b}^{\mathrm{B}} - \mathbf{A}_1^{\mathrm{BI}} \left(\mathbf{A}_1^{\mathrm{II}}\right)^{-1} \mathbf{b}_1^{\mathrm{I}} \end{pmatrix}$$

the reduced system of equations becomes

$$\left(\mathbf{A}^{\mathrm{BB}} - \sum_{i=1}^{p} \mathbf{A}_i^{\mathrm{BI}} \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \mathbf{A}_i^{\mathrm{IB}}\right) \mathbf{x}^{\mathrm{B}} = \mathbf{b}^{\mathrm{B}} - \sum_{i=1}^{p} \mathbf{A}_i^{\mathrm{BI}} \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \mathbf{b}_i^{\mathrm{I}}.$$

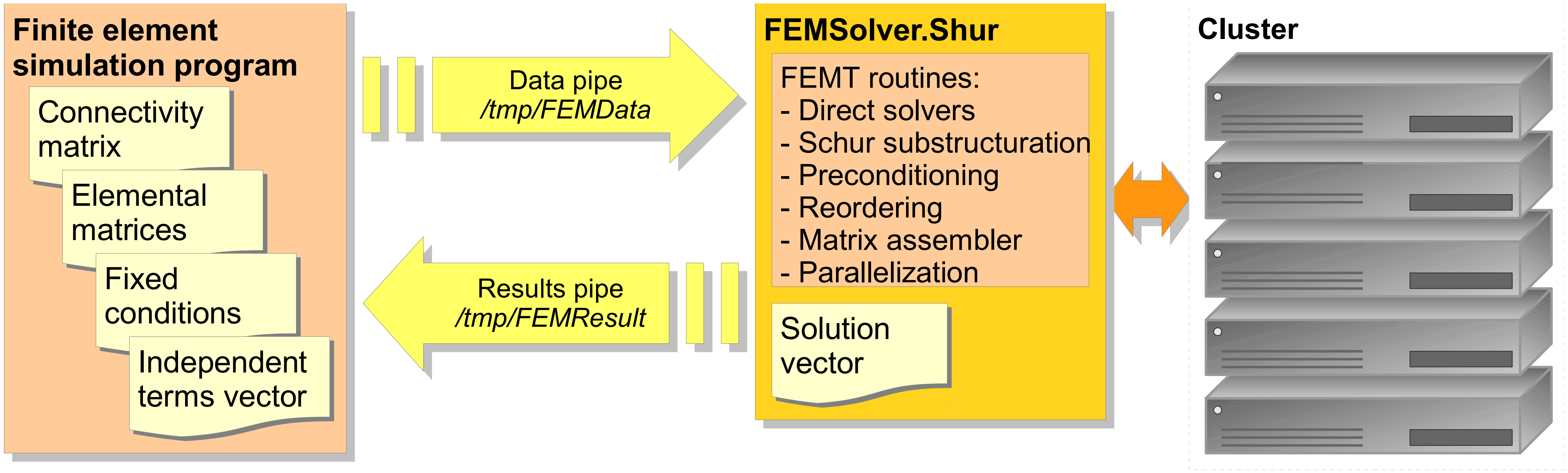
Each $\mathbf{A}_i^{\mathrm{BI}} \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \mathbf{A}_i^{\mathrm{IB}}$, and $\mathbf{A}_i^{\mathrm{BI}} \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \mathbf{b}_i^{\mathrm{I}}$ is calculated in a different processor.

Once the vector $\mathbf{x}^{\mathrm{B}}$ is computed, we can calculate the internal unknowns $\mathbf{x}_i^{\mathrm{I}}$ using

$$\mathbf{x}_i^{\mathrm{I}} = \left(\mathbf{A}_i^{\mathrm{II}}\right)^{-1} \left(\mathbf{b}_i^{\mathrm{I}} - \mathbf{A}_i^{\mathrm{IB}} \mathbf{x}^{\mathrm{B}}\right).$$

FEMSolver.Schur

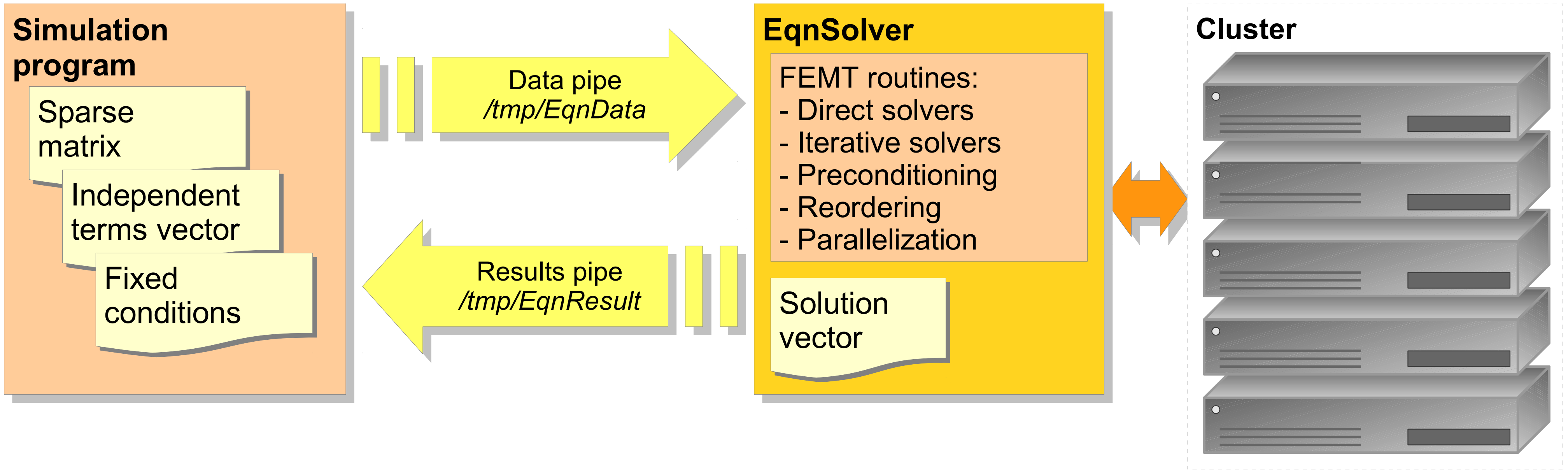
FEMSolver.Schur is a program similar to FEMSolver, but instead of solving the system of equations using a single computer, it can use a cluster of computers to distribute the workload and solve even larger systems of equations.



It uses the MPI technology to handle communication between nodes in the cluster. It makes high performance computing (HPC) easy to use.

EqnSolver.Schur

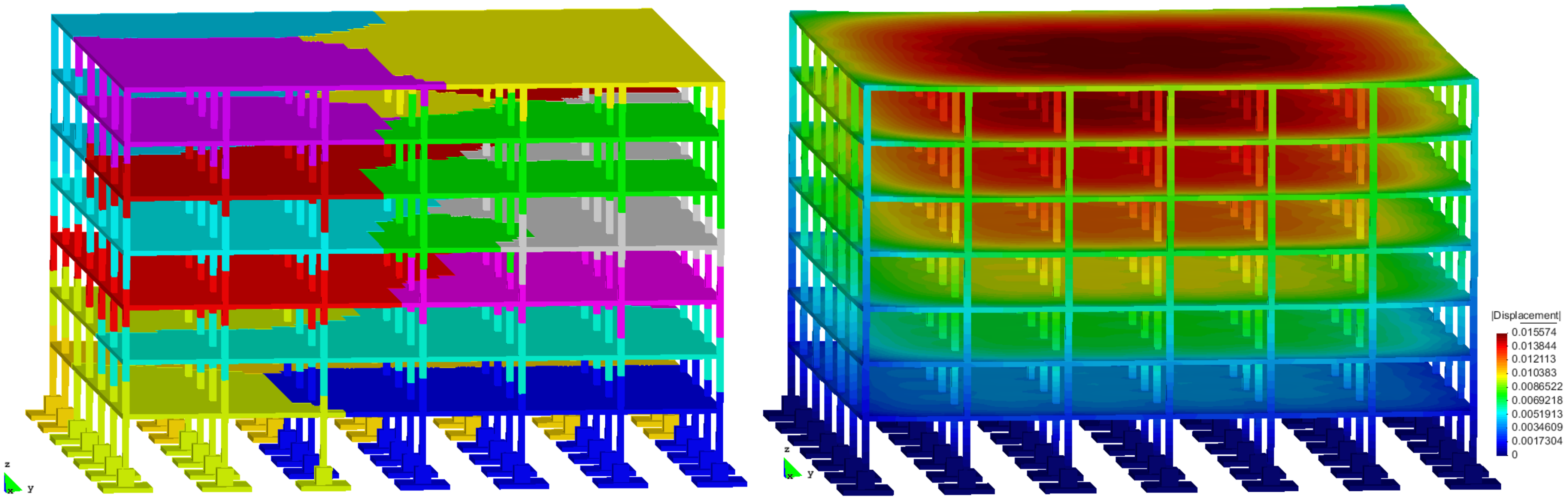
EqnSolver.Schur is a program similar to EqnSolver, but instead of solving the system of equations using a single computer, it can use a cluster of computers to distribute the workload and solve even larger systems of equations.



Building deformation

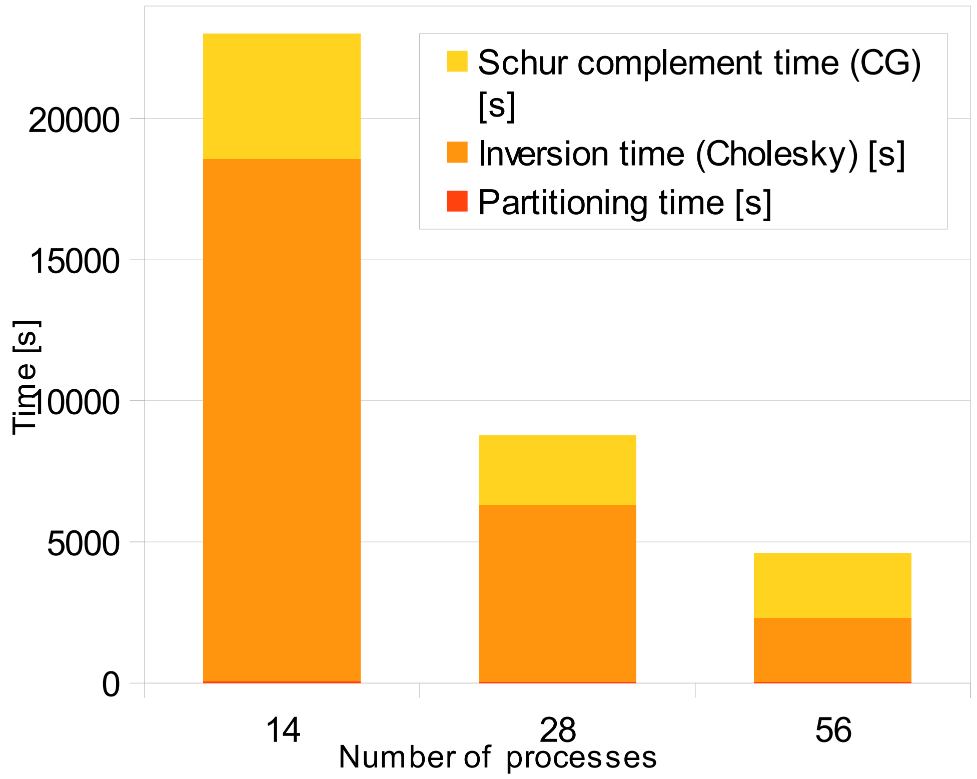
We used a cluster with 15 nodes, each one with two dual core Intel Xeon E5502 (1.87GHz) processors, a total of 60 cores.

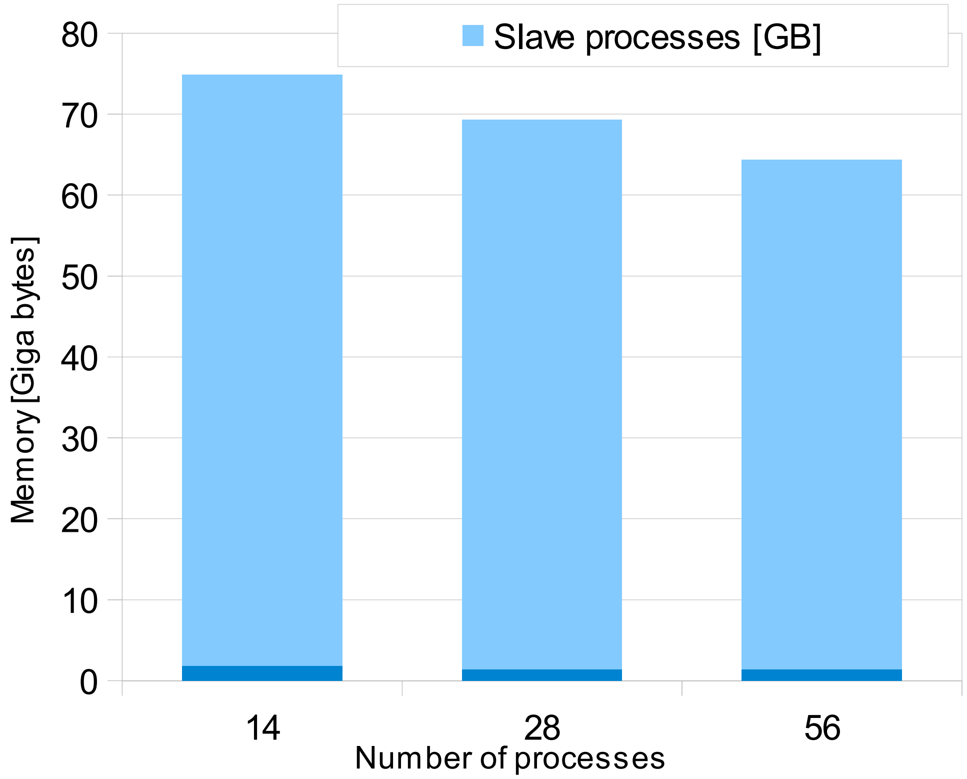
The problem tested is a 3D solid model of a building that is deformed due to self weight. The geometry is divided in 1’336,832 elements, with 1’708,273 nodes, with three degrees of freedom per node the resulting system of equations has 5’124,819 unknowns. Tolerance used is 1x10^{-10}.



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

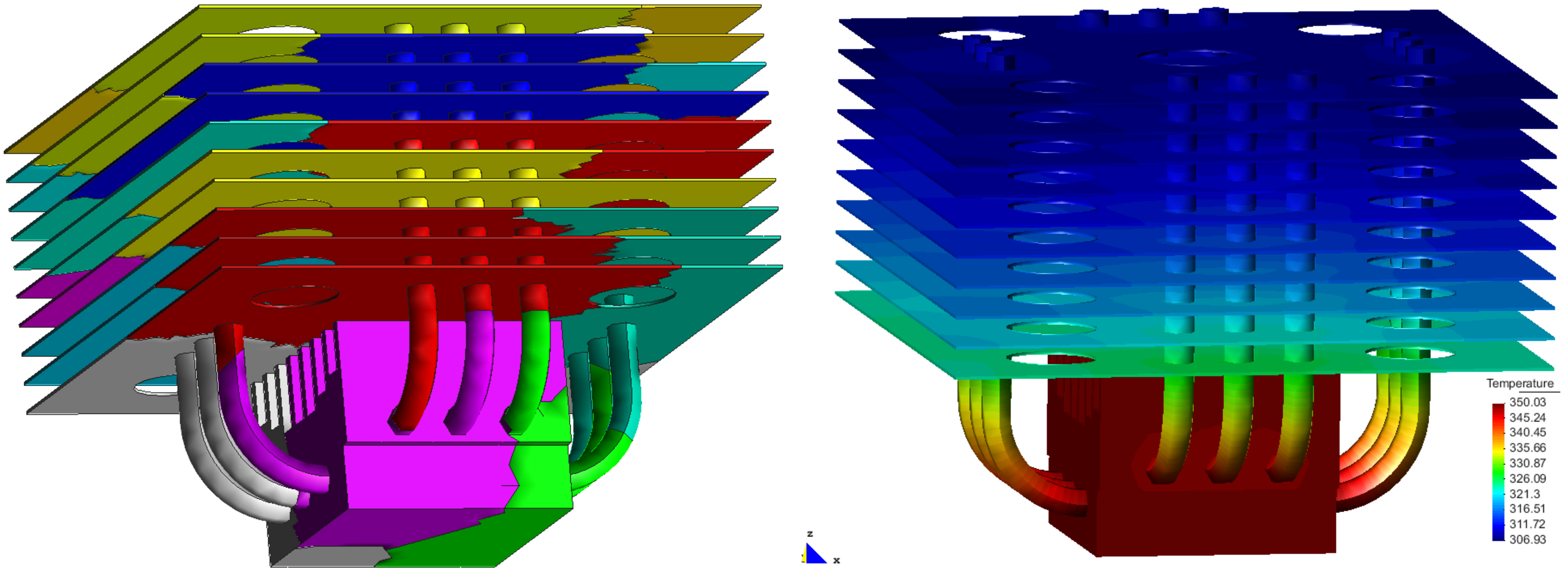




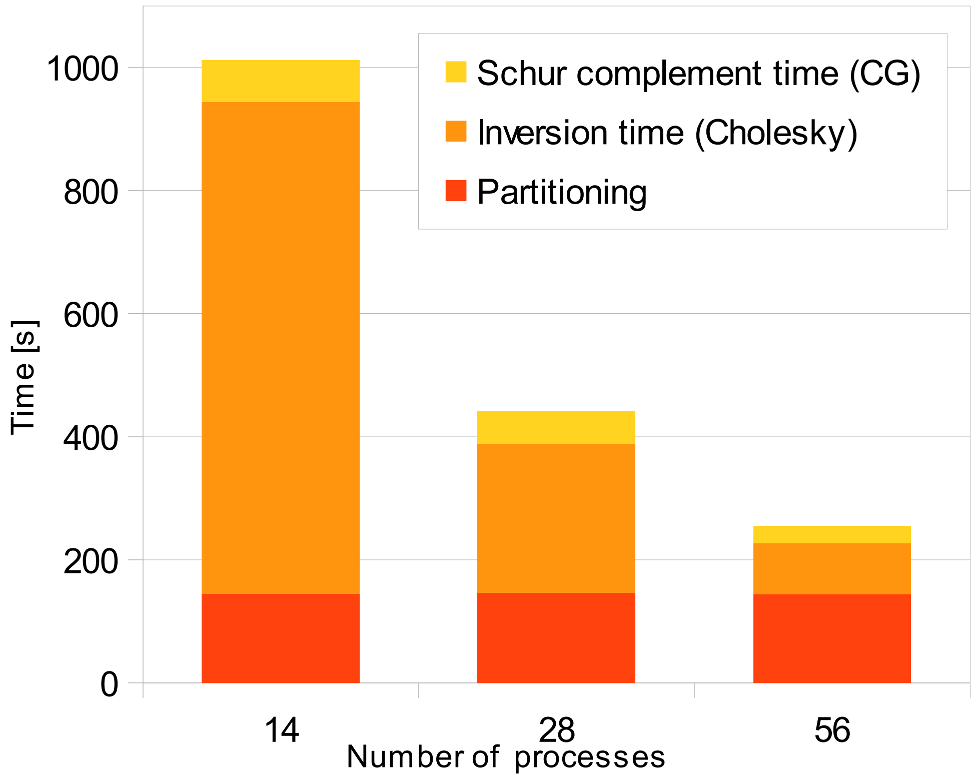
Number of processes	Master process [GB]	Slave processes [GB]	Total memory [GB]
14	1.89	73.00	74.89
28	1.43	67.88	69.32
56	1.43	62.97	64.41

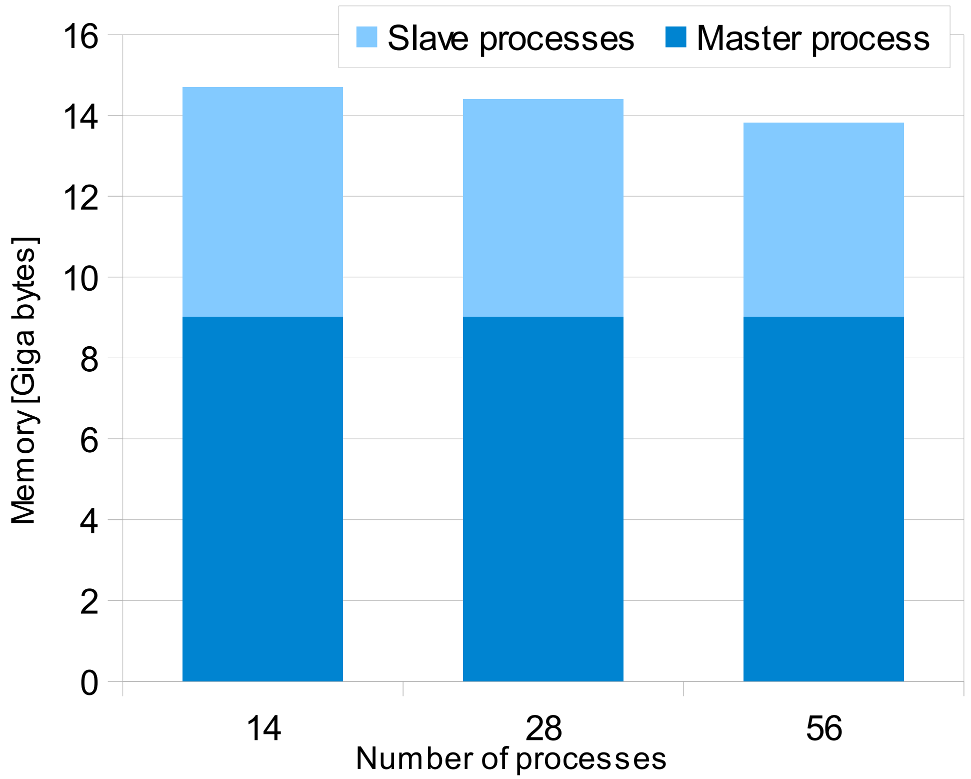
Heat diffusion

This is a 3D model of a heat sink, in this problem the base of the heat sink is set to a certain temperature and heat is lost in all the surfaces at a fixed rate. The geometry is divided in 4'493,232 elements, with 1'084,185 nodes. The system of equations solved had 1'084,185 unknowns.



Number of processes	Partitioning time [s]	Inversion time (Cholesky) [s]	Schur complement time (CG) [s]	CG steps	Total time [s]
14	144.9	798.5	68.1	307	1020.5
28	146.6	242.0	52.1	348	467.1
56	144.2	82.8	27.6	391	264.0



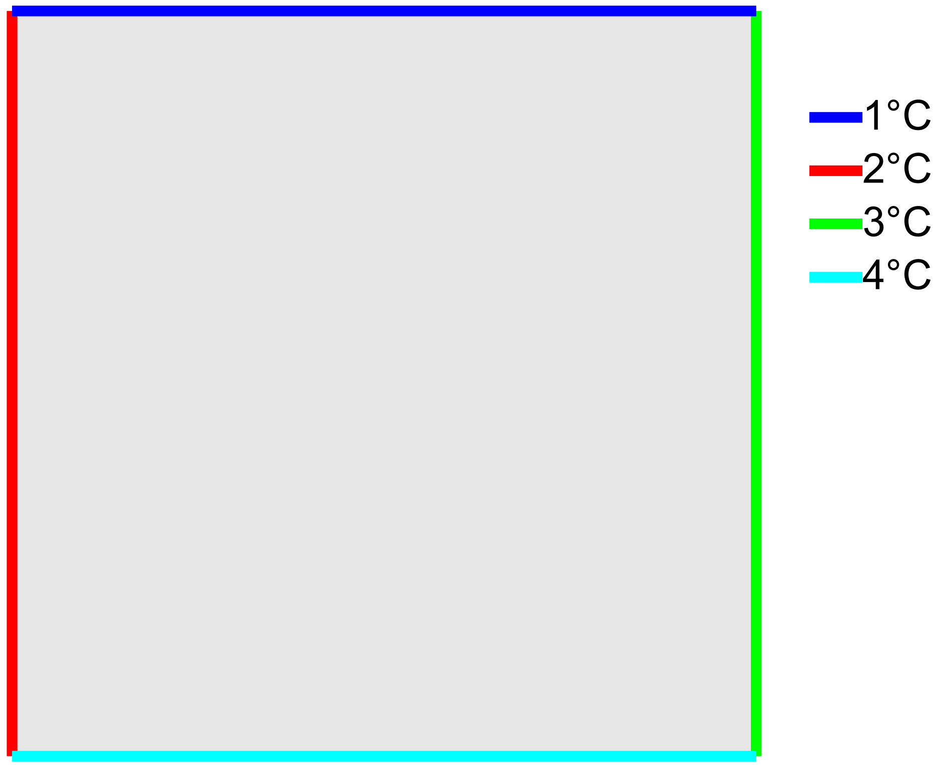


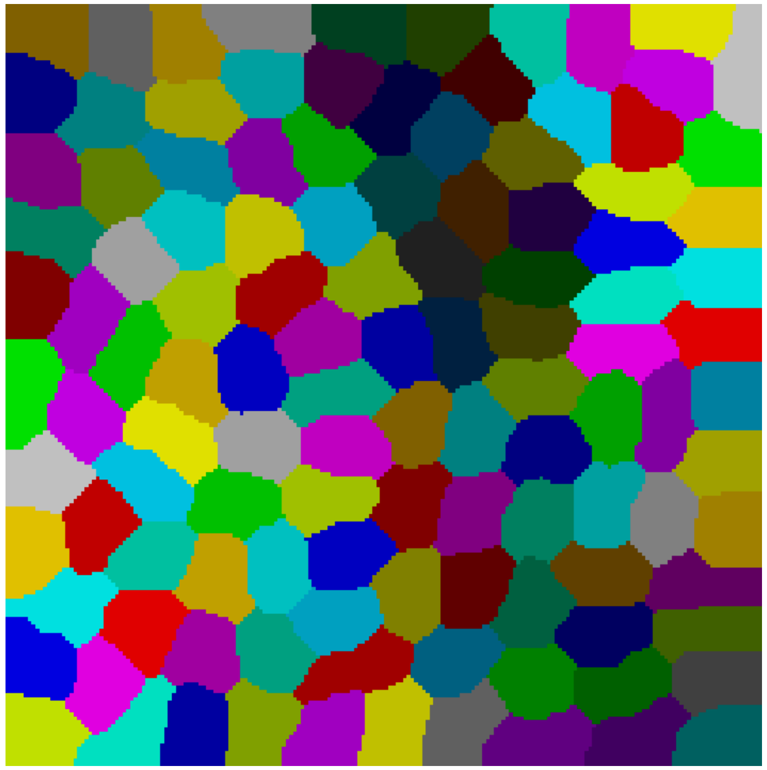
Number of processes	Master process [GB]	Slave processes [GB]	Total memory [GB]
14	9.03	5.67	14.70
28	9.03	5.38	14.41
56	9.03	4.80	13.82

Larger systems of equations

To test solution times in larger systems of equations we set a simple geometry.

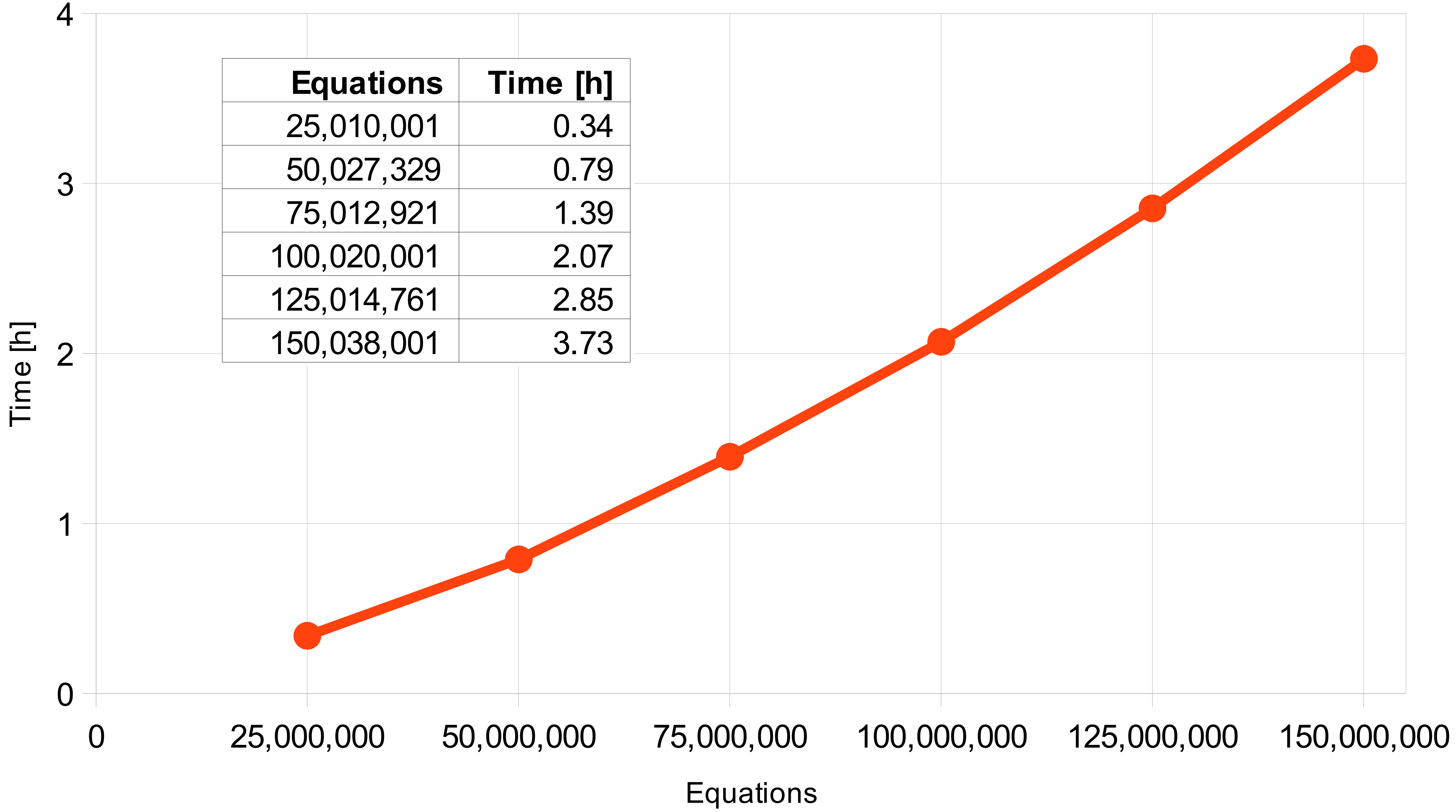
We calculated the temperature distribution of a metalic unit square with Dirichlet conditions on all boundaries.





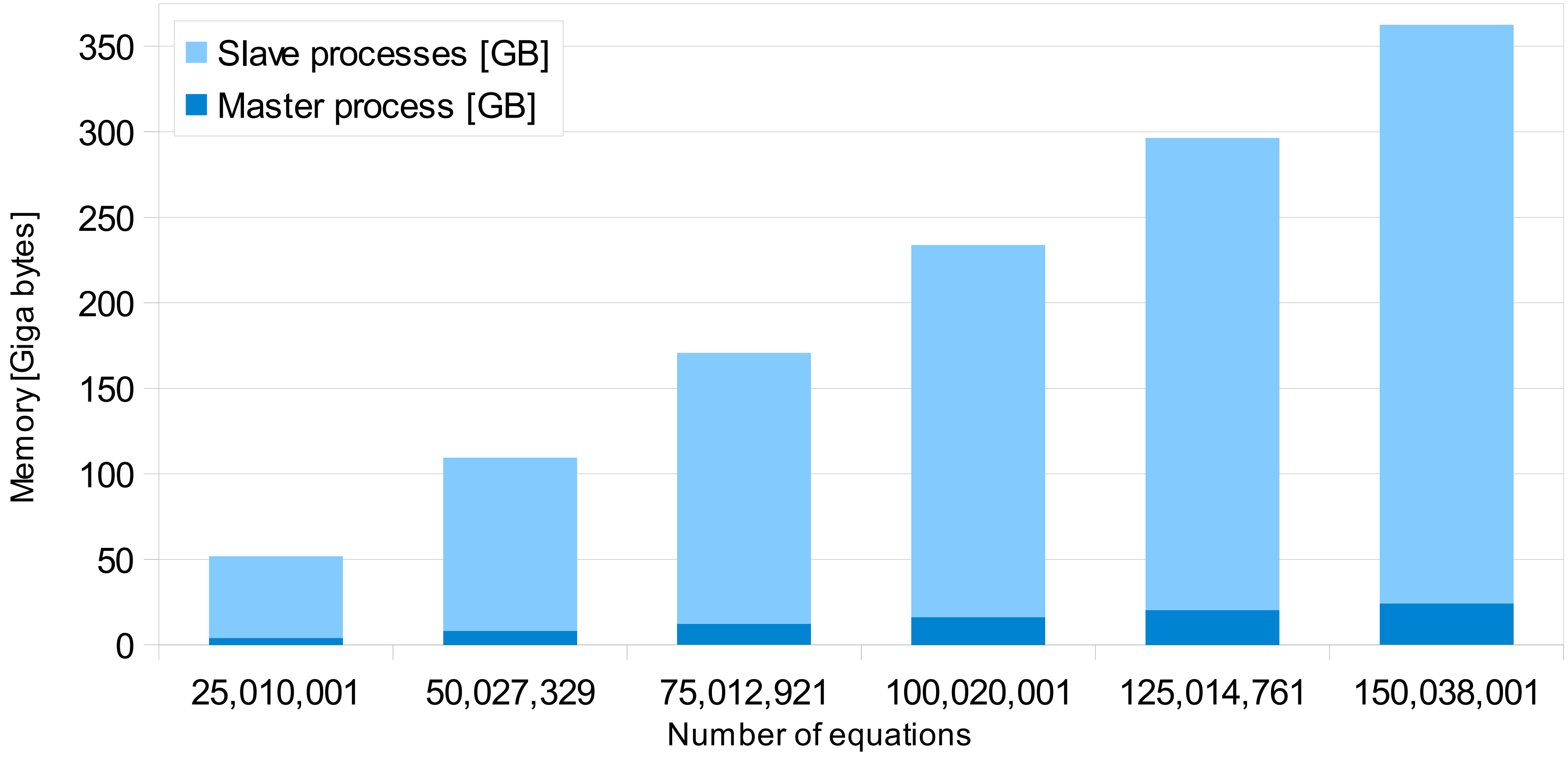
We divided the domain into 124 partitions, each partition is solved in one core.

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



Equations	Time [h]
25,010,001	0.34
50,027,329	0.79
75,012,921	1.39
100,020,001	2.07
125,014,761	2.85
150,038,001	3.73

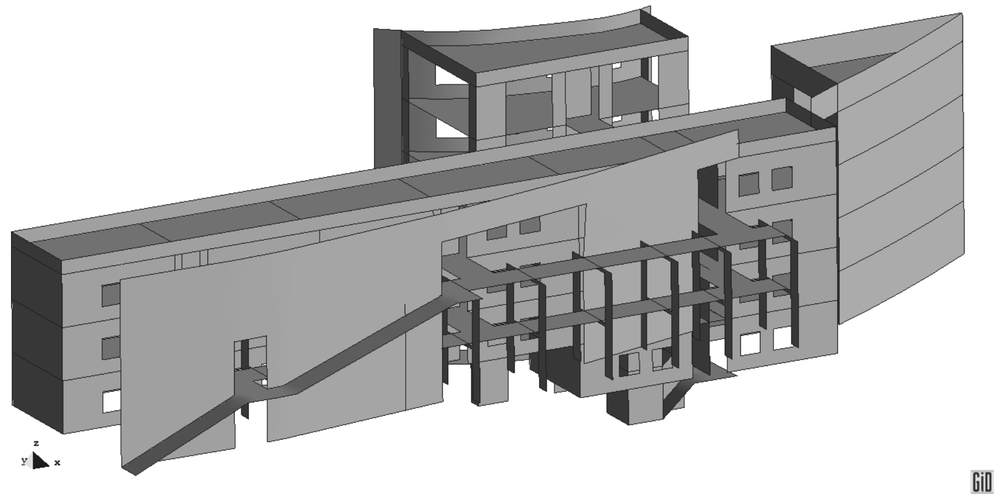
Equations	CG steps	Total time [h]	Master process [GB]	Slave processes (average) [GB]	Total memory [GB]
25,010,001	863	0.34	4.05	0.41	51.79
50,027,329	1036	0.79	8.10	0.87	109.31
75,012,921	1146	1.39	12.15	1.37	170.68
100,020,001	1236	2.07	16.20	1.88	233.71
125,014,761	1289	2.85	20.25	2.38	296.29
150,038,001	1342	3.73	24.30	2.92	362.60

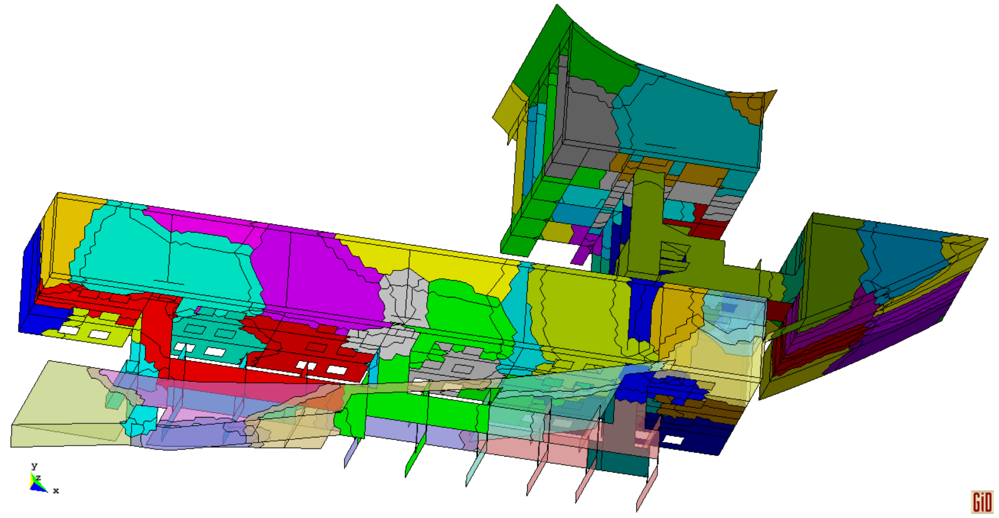


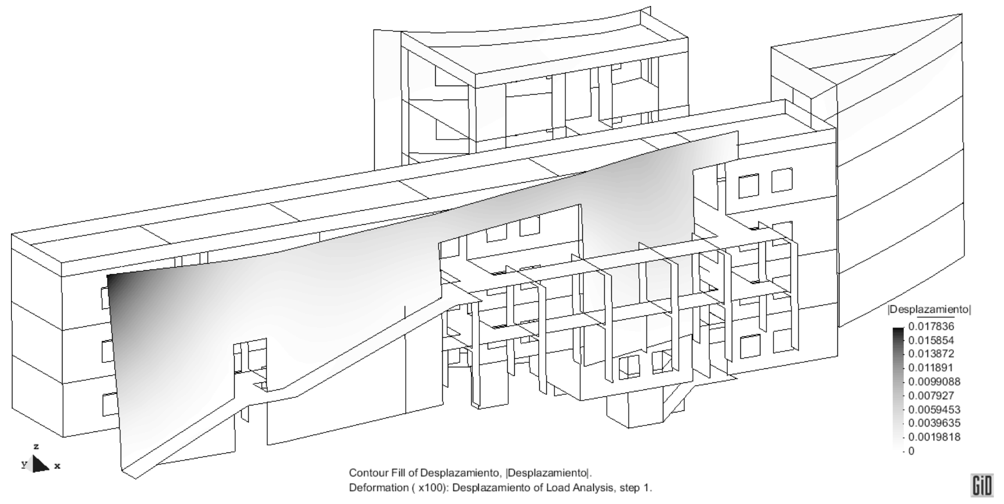
Calsef (Dr. Salvador Botello et al.)

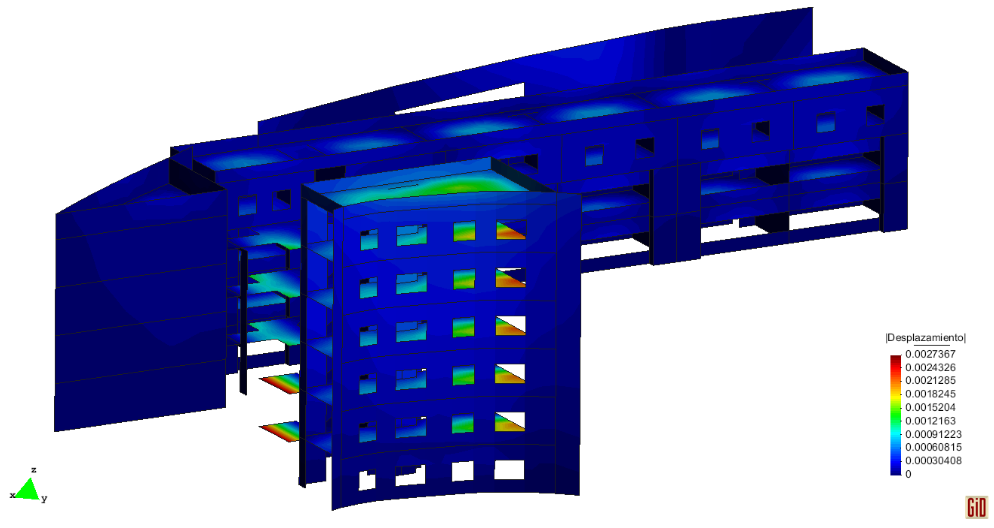
Calsef is a program to solve problems of structural mechanics. It was written in Fortran 77.

Simple functions were added to Calsef to interact to FEMSolver using pipes.

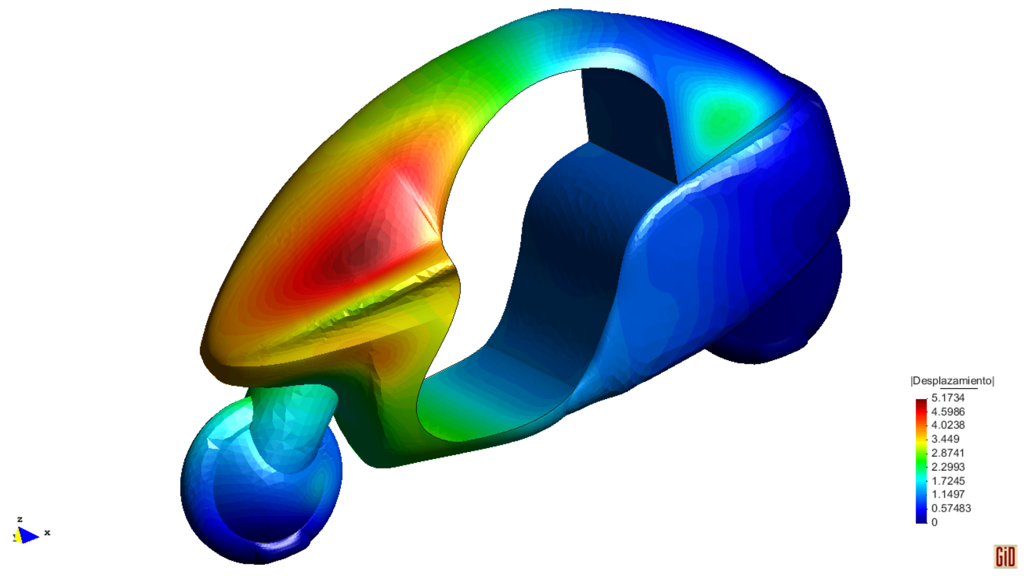


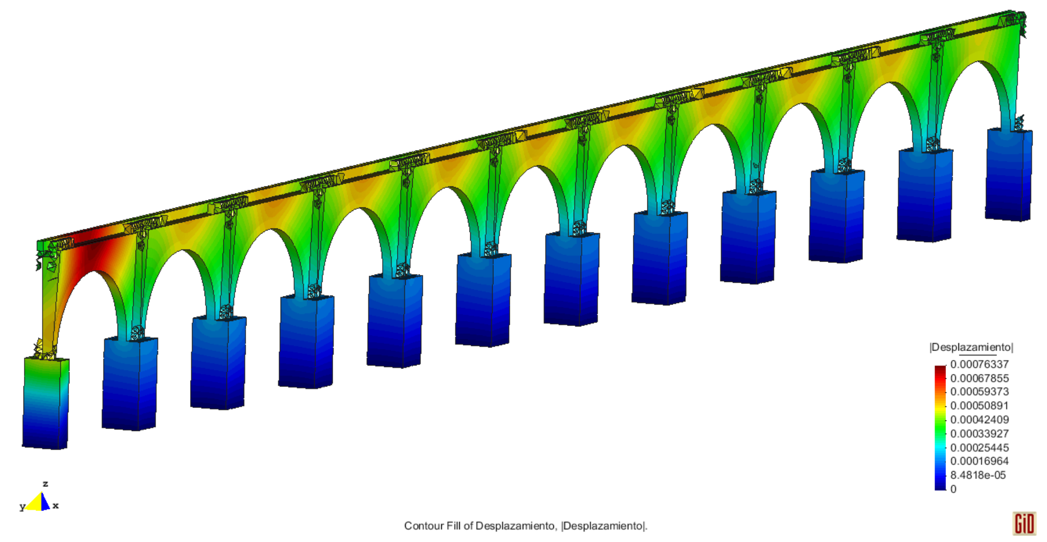


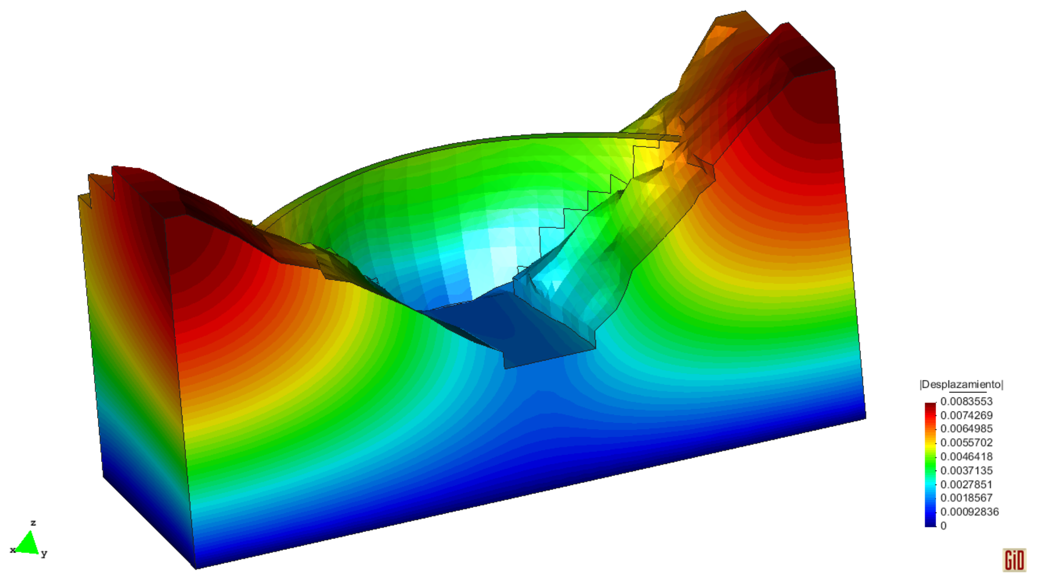

|Desplazamiento|
0.017836
0.015854
0.013872
0.011891
0.0099088
0.007927
0.0059453
0.0039635
0.0019818
0
Contour Fill of Desplazamiento, |Desplazamiento|.
Deformation (x100): Desplazamiento of Load Analysis, step 1.
GiD

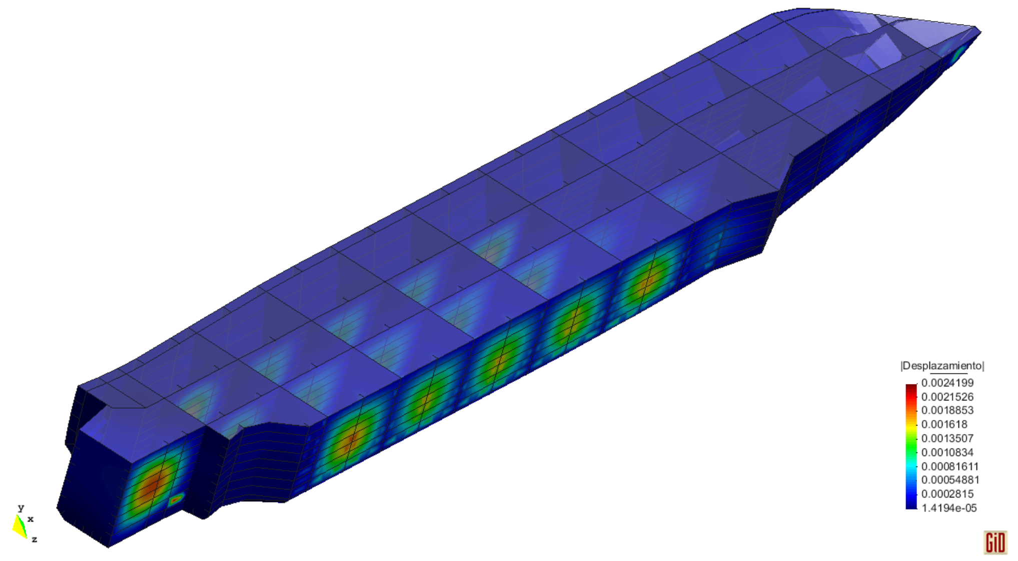


Otros ejemplos de problemas resueltos utilizando Calsef+FEMSolver









Modelo de subdifusión (Dr. Joaquín Peña et al.)

Resolvemos el problema transitorio

$$\begin{aligned} b(\mathbf{r})\tfrac{\partial p}{\partial t} - \nabla \cdot (a(\mathbf{r})\nabla p) &= f(\mathbf{r},t) && \mathbf{r} \in \Omega,\ t \geq 0. \\ \nabla p(\mathbf{r},t)\cdot \mathbf{n} &= \alpha && \mathbf{r} \in \partial\Omega,\ t \geq 0, \\ p(\mathbf{r},0) &= p_0(\mathbf{r}) && \mathbf{r} \in \Omega. \end{aligned}$$

donde

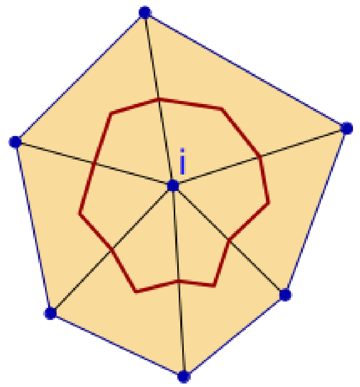
$$\begin{aligned} a(\mathbf{r}) &= \frac{\Gamma(\frac{D}{2})\, 2^{D-d-1}}{\Gamma(d)} \|\mathbf{r}-\mathbf{r}_0\|^{d-1}\,\rho\,\kappa, \\ b(\mathbf{r}) &= c_f \rho^0 \|\mathbf{r}-\mathbf{r}_0\|^{D-2}, \\ f(\mathbf{r},t) &= q(\mathbf{r},t)\|\mathbf{r}-\mathbf{r}_0\|^{D-2}. \end{aligned}$$

Los exponentes d y D, $1 \leq d < D \leq 2$, determinan el modo en que la difusión ocurre.

Solución usando MEFVC

Usamos el método de elementos finitos basados en volúmenes de control:

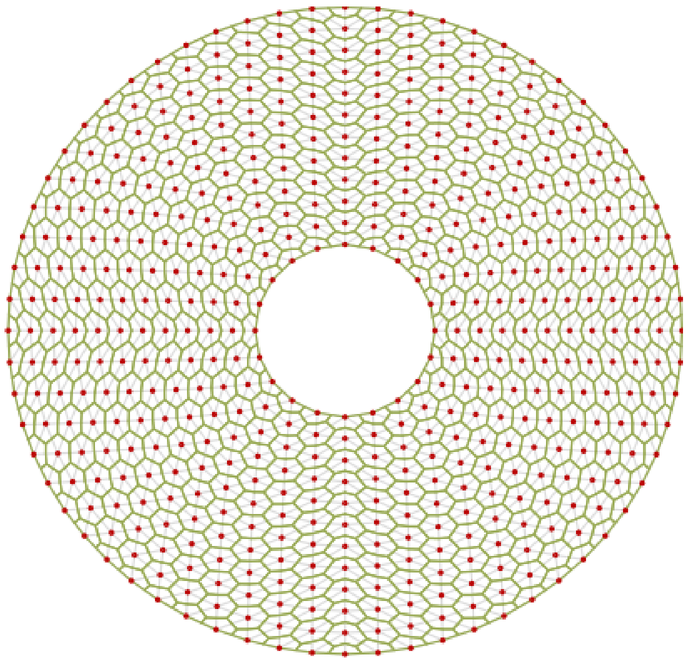
- Al nodo i de la discretización se le asocia el volumen de control V_i.
- Integramos la ecuación sobre cada V_i.
- Aproximamos la solución mediante funciones lineales y usamos Crank-Nicholson.

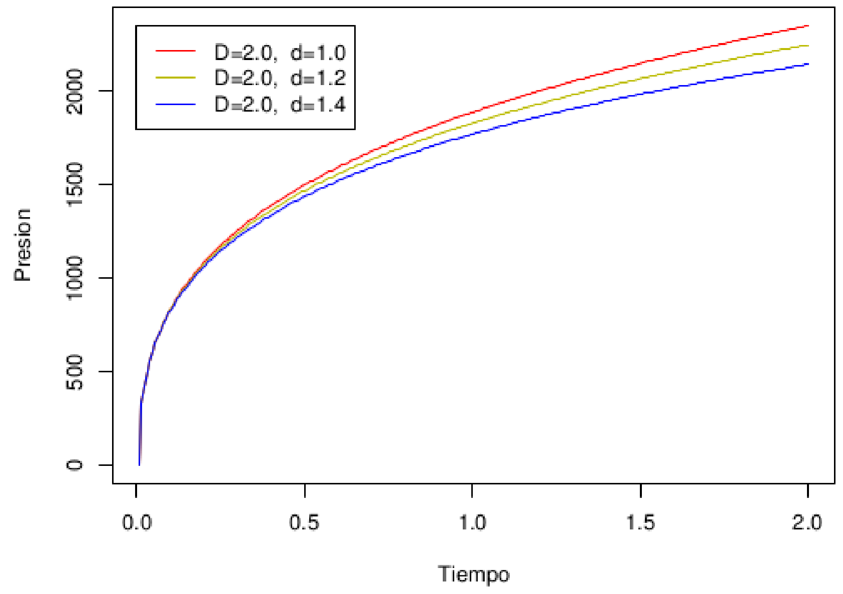


$$\overbrace{\frac{p_i^{k+1}-p_i^k}{\Delta t}\int_{V_i} b(\mathbf{r})\,d\mathbf{r}}^{\int_{V_i} b(\mathbf{r})p_t\,d\mathbf{r}} = \overbrace{\sum_{j\in S_i} T_{ij}(p_j^{k+1}-p_i^{k+1})}^{\int_{\partial V_i} a(\mathbf{r})\nabla p\cdot\mathbf{n}\,dl} + \overbrace{\int_{V_i} f(\mathbf{r},t_k)\,d\mathbf{r}}^{F_i^k}$$

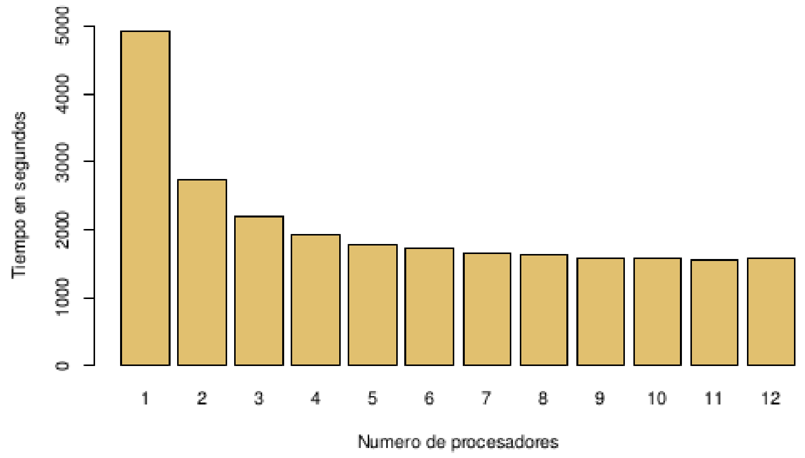
$$\mathbf{A}\mathbf{p}^{k+1} = \mathbf{B}\mathbf{p}^k + \frac{1}{2}\left[F_i^k + F_i^{k+1}\right].$$

Resultados usando EqnSolver

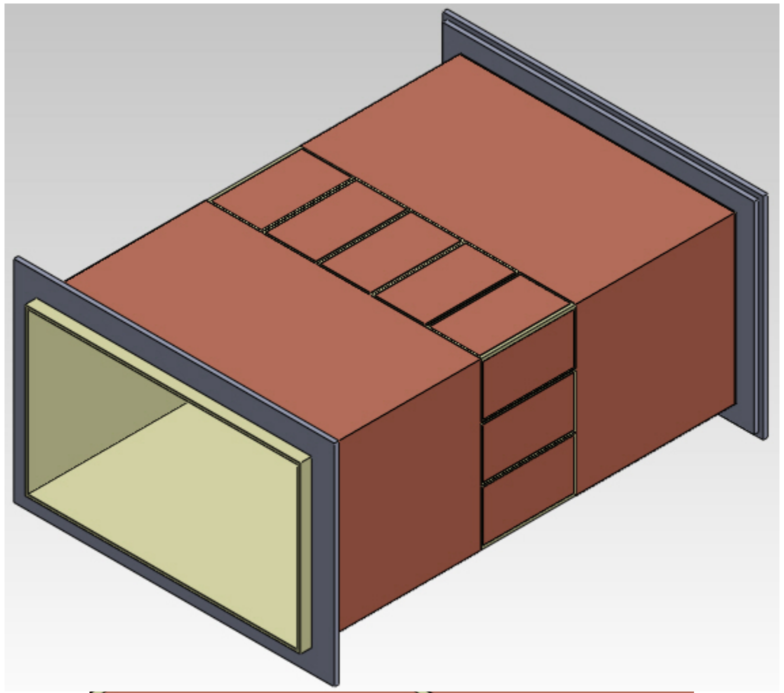


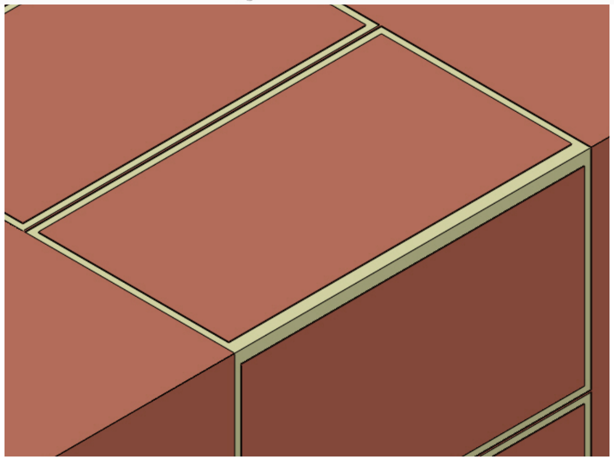


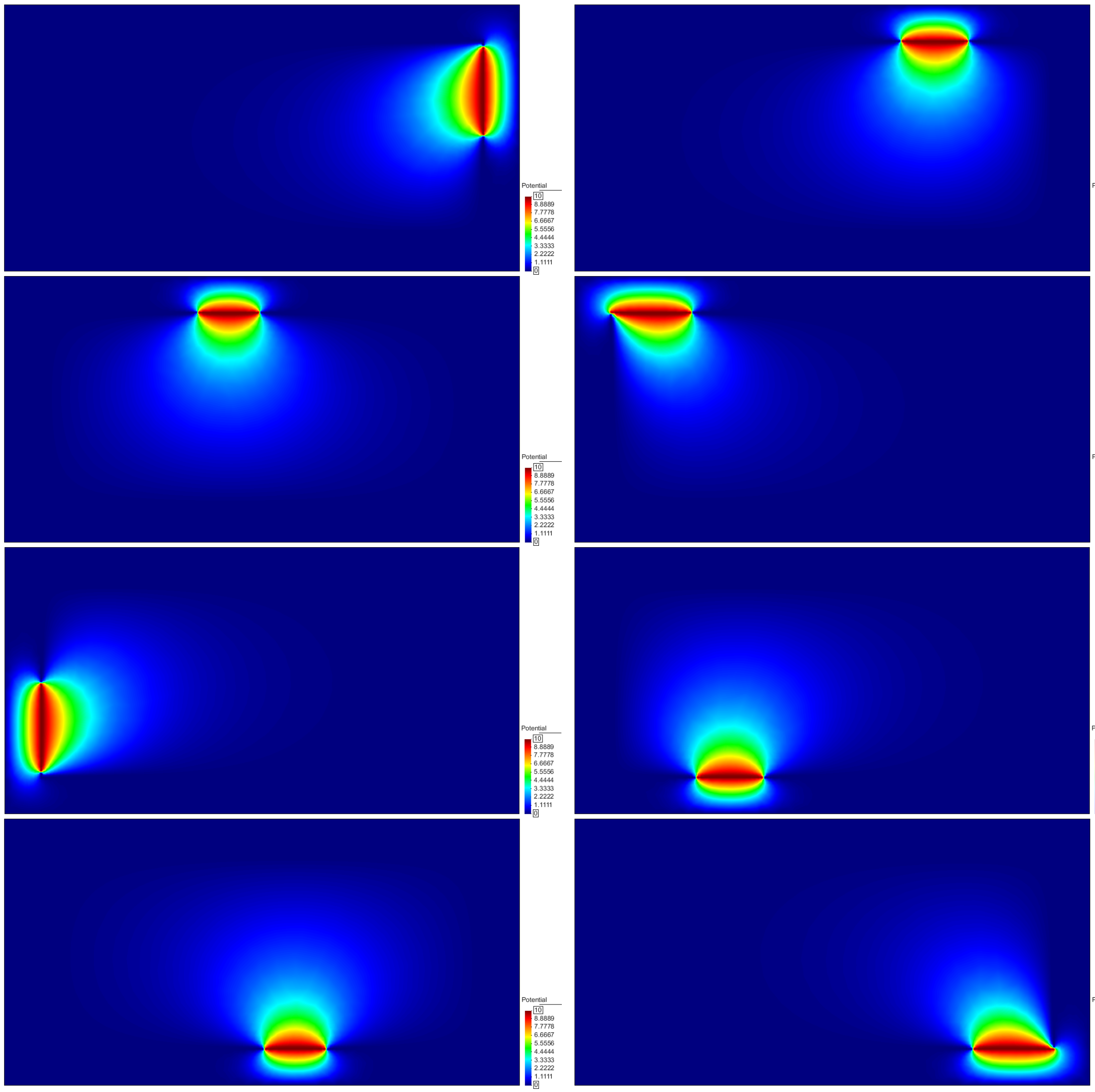
145912	nodos
290628	elementos triangulares
436540	aristas
1000	pasos de tiempo



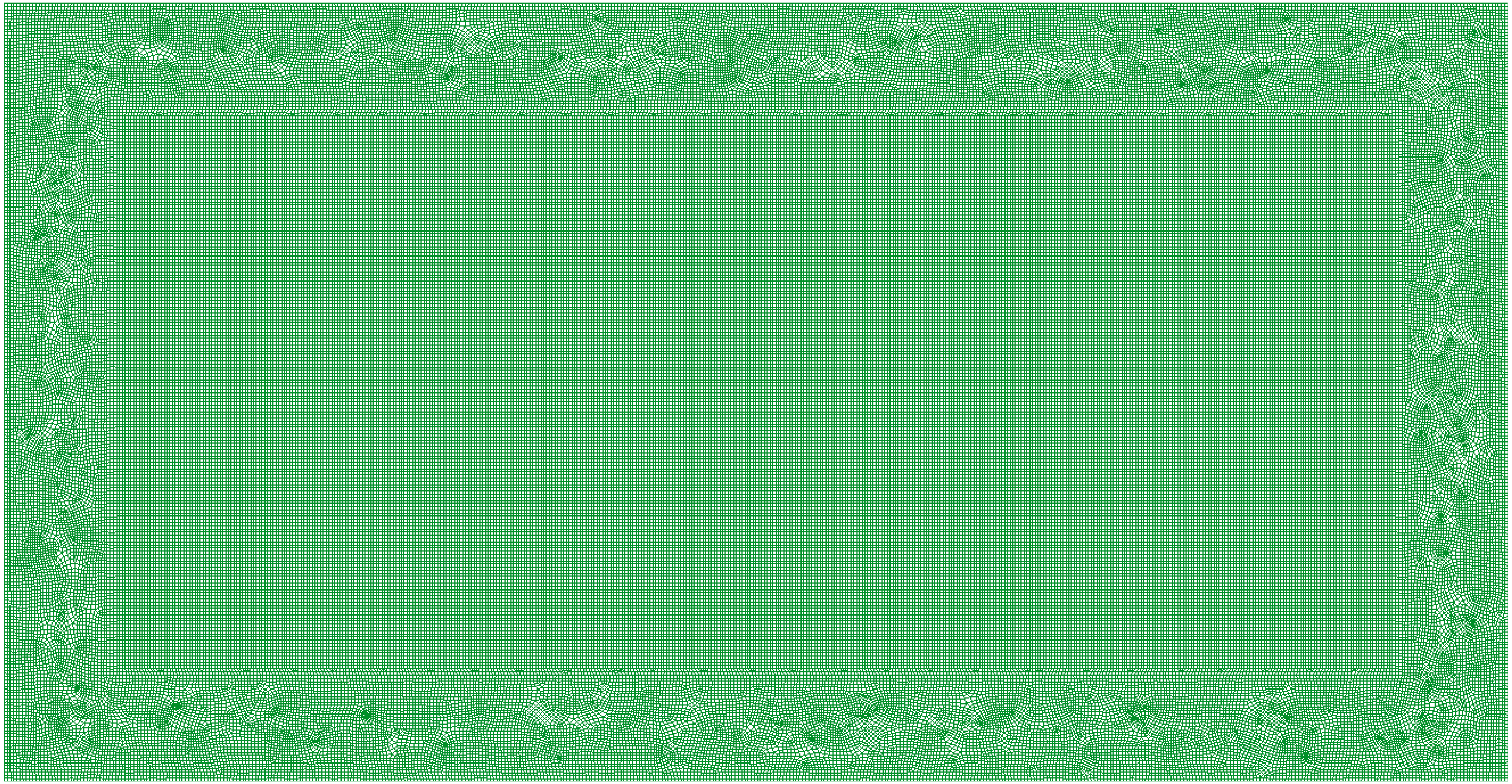
Tomograph by capacitance (Dr Norberto Flores et al.)







The mesh has 131,530 elements and 132,249 nodes.

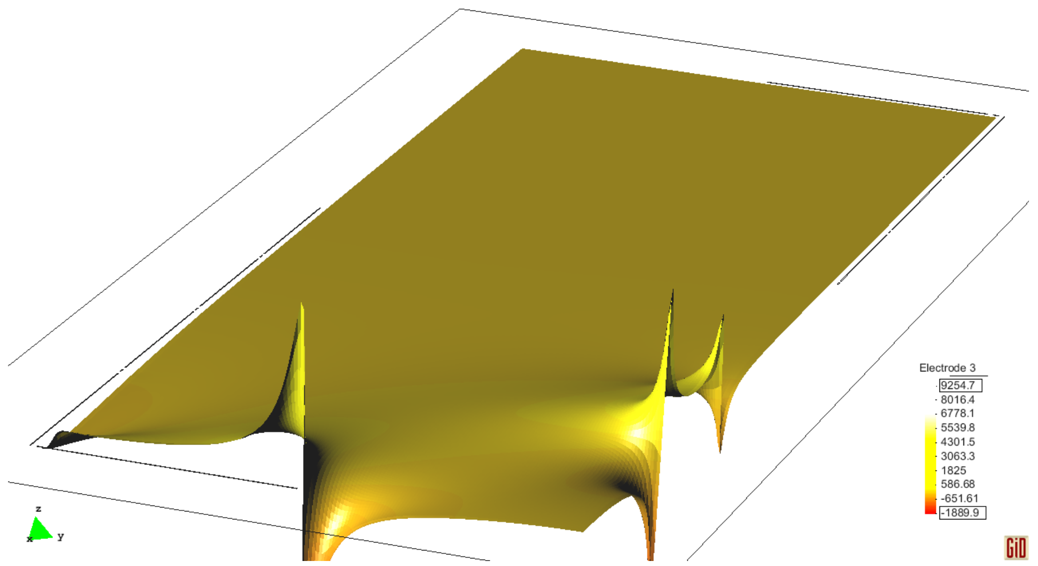


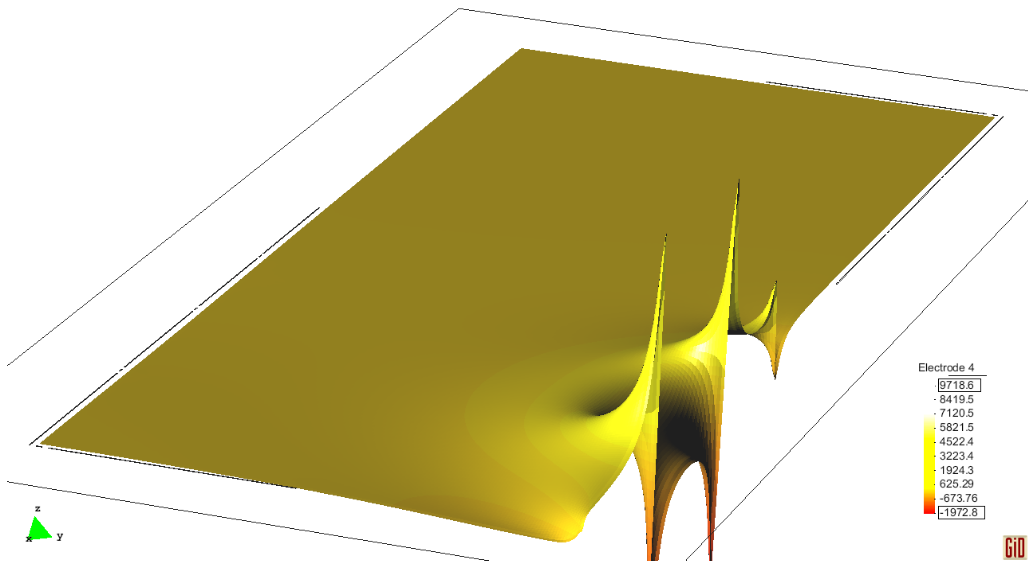
The sensitivity area has 458*174 = 79,692 pixels.

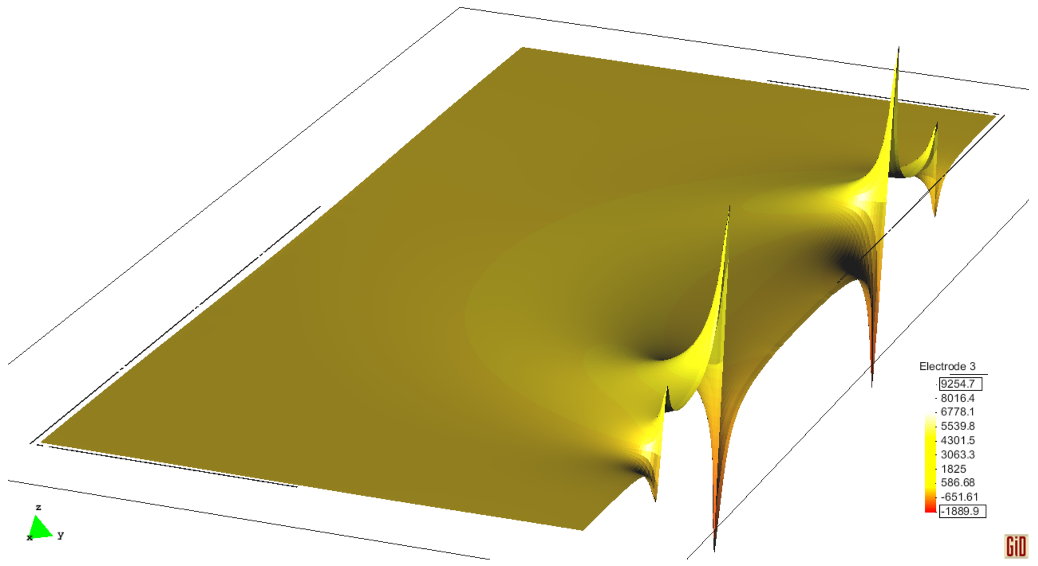
Each pixel has to be tested with all electrodes.

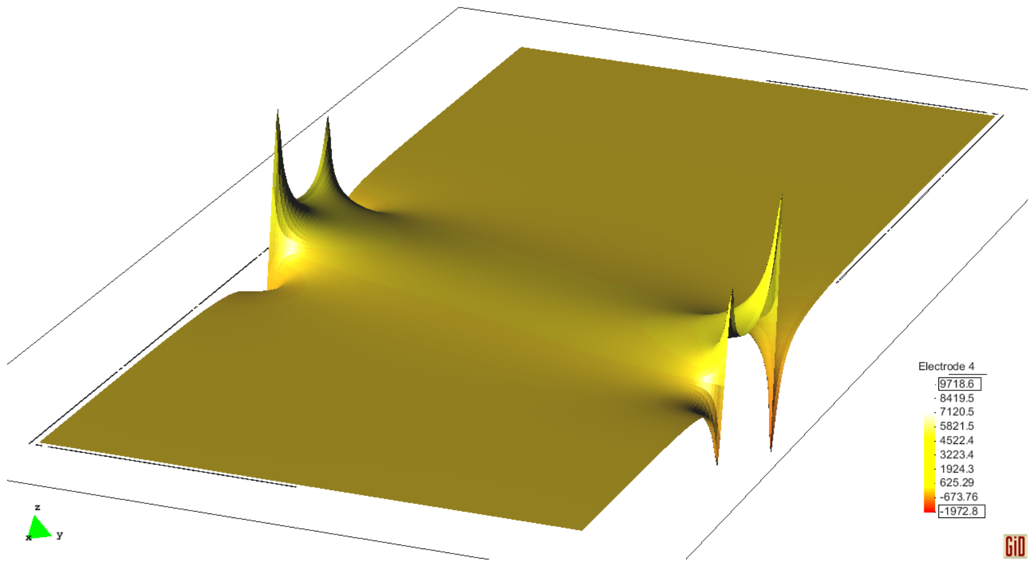
Therefore, the finite element problem has to be solved 79,692*16 = 1’275,072 times.

All this solutions are used to create sensitivity maps.









Planned future work

Better parallelization

- Support for libNUMA and Windows NUMA.
- Data containers with address alignment to support parallelization using SSE instructions.

More solvers

- Generalized minimal residual method (GMRES) (beta)
- Biconjugate gradient stabilized method (BiCGSTAB)
- Schur substructuration for non-sysmmetric matrices (beta).

More preconditioners

- Factorized sparse approximate inverse for non-symmetric matrices (beta)

Support for more compilers

- LLVM clang++ (New standard for Mac OS X and BSD systems) (beta)

Thank you!
Questions?

FEMT:

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

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

References

[Chow98] E. Chow, Y. Saad. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. SIAM Journal on Scientific Computing. Vol. 19-3, pp. 995-1023. 1998.

[Chow01] E. Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. International Journal of High Performance Computing, Vol 15. pp 56-74, 2001.

[DAze93] E. F. D'Azevedo, V. L. Eijkhout, C. H. Romine. Conjugate Gradient Algorithms with Reduced Synchronization Overhead on Distributed Memory Multiprocessors. Lapack Working Note 56. 1993.

[Drep07] U. Drepper. What Every Programmer Should Know About Memory. Red Hat, Inc. 2007.

[Farh91] C. Farhat and F. X. Roux, A method of finite element tearing and interconnecting and its parallel solution algorithm, Internat. J. Numer. Meths. Engrg. 32, 1205-1227 (1991)

[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.

[Geor89] A. George, J. W. H. Liu. The evolution of the minimum degree ordering algorithm. SIAM Review Vol 31-1, pp 1-19, 1989.

[Golu96] G. H. Golub, C. F. Van Loan. Matrix Computations. Third edition. The Johns Hopkins University Press, 1996.

[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.

[Smit96] B. F. Smith, P. E. Bjorstad, W. D. Gropp. Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, 1996.

[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

[Ster95] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, C. V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. Proceedings of the 24th International Conference on Parallel Processing, 1995.

[Tose05] A. Toselli, O. Widlund. Domain Decomposition Methods - Algorithms and Theory. Springer, 2005.

[Varg10] J. M. Vargas-Felix, S. Botello-Rionda. “Parallel Direct Solvers for Finite Element Problems”. Comunicaciones del CIMAT, I-10-08 (CC), 2010.
http://www.cimat.mx/reportes/enlinea/I-10-08.pdf

[Wulf95] W. A. Wulf , S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News, 23(1):20-24, March 1995.

[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.

[Zien05] O.C. Zienkiewicz, R.L. Taylor, J.Z. Zhu, The Finite Element Method: Its Basis and Fundamentals. Sixth edition, 2005.