Comparison of different solution strategies for structure deformation using hybrid OpenMP-MPI methods

Miguel Vargas, Salvador Botello

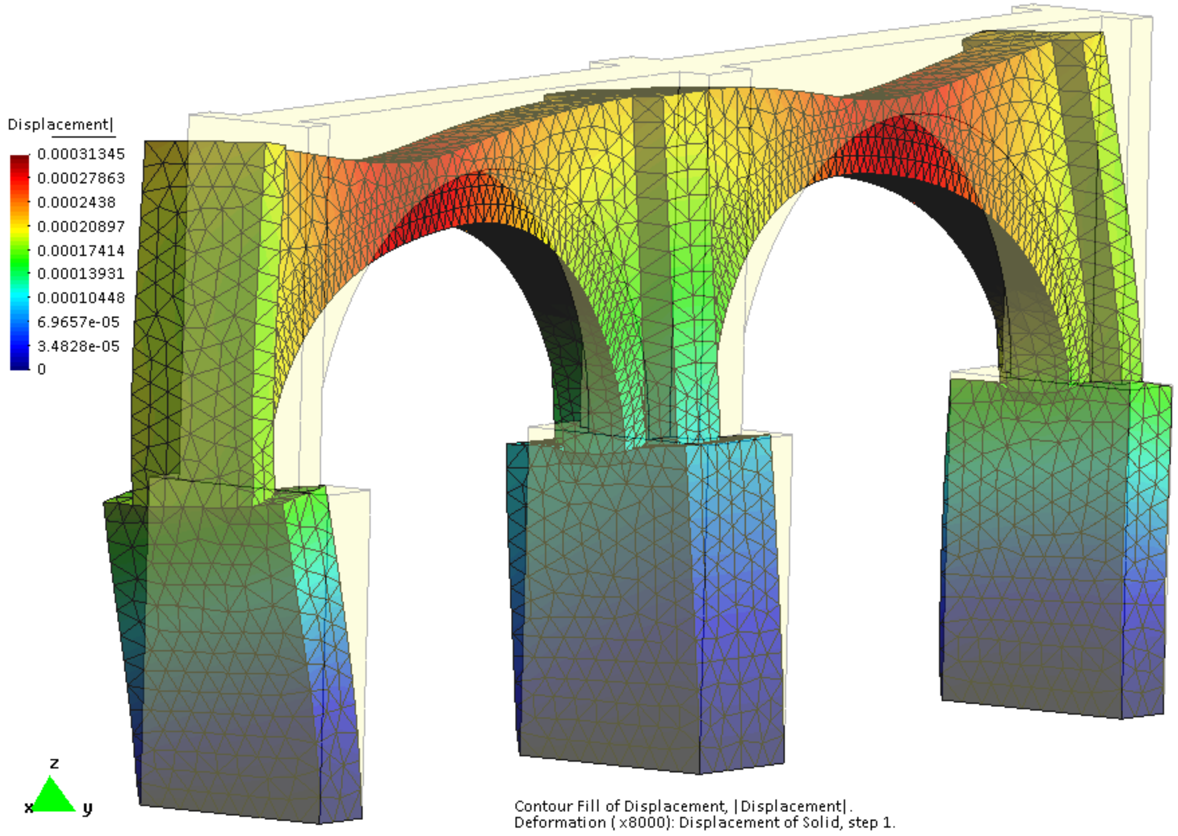
CIMAT

Description of the problem

This is a high performance/large scale application case studie of the finite element method for solid mechanics. Our goal is to calculate linear deformation, stain an stress of solids dicretized with large meshes (millions of elements) using parallel computing.

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

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



Where $\boldsymbol{u}$ is the displacement vector, $\boldsymbol{\varepsilon}$ the stain, $\boldsymbol{\sigma}$ the stress. $\boldsymbol{D}$ is called the constitutive matrix.

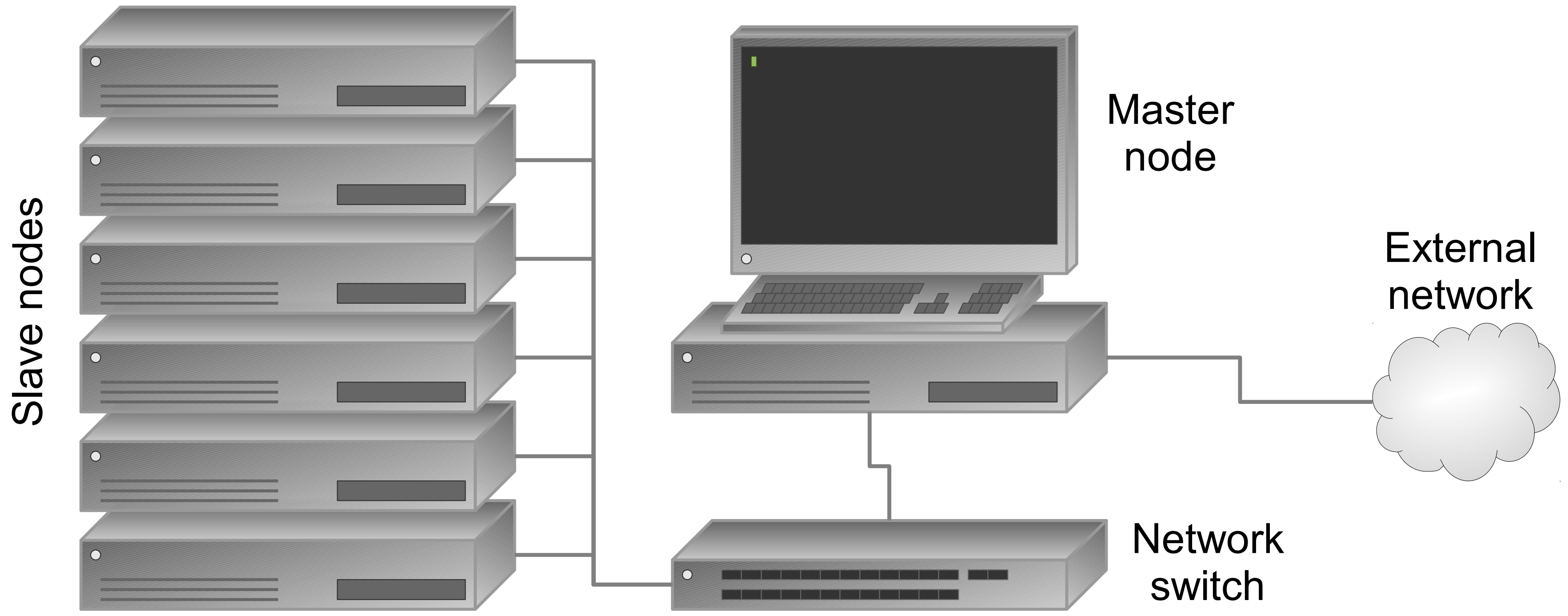
The solution is found using the finite element method with the Galerkin weighted residuals.

By discretizing the domain, and applying boundary conditions (imposed displacements or applied forces), we assemble a linear system of equations

$$\boldsymbol{K}\,\boldsymbol{u} = \boldsymbol{f}\,,$$

where $\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{u}$ is the displacement vector and $\boldsymbol{f}$ is the force vector. If the problem is well-defined $\boldsymbol{K}$ is symmetric positive definite (SPD). This matrix is also ***sparse***.

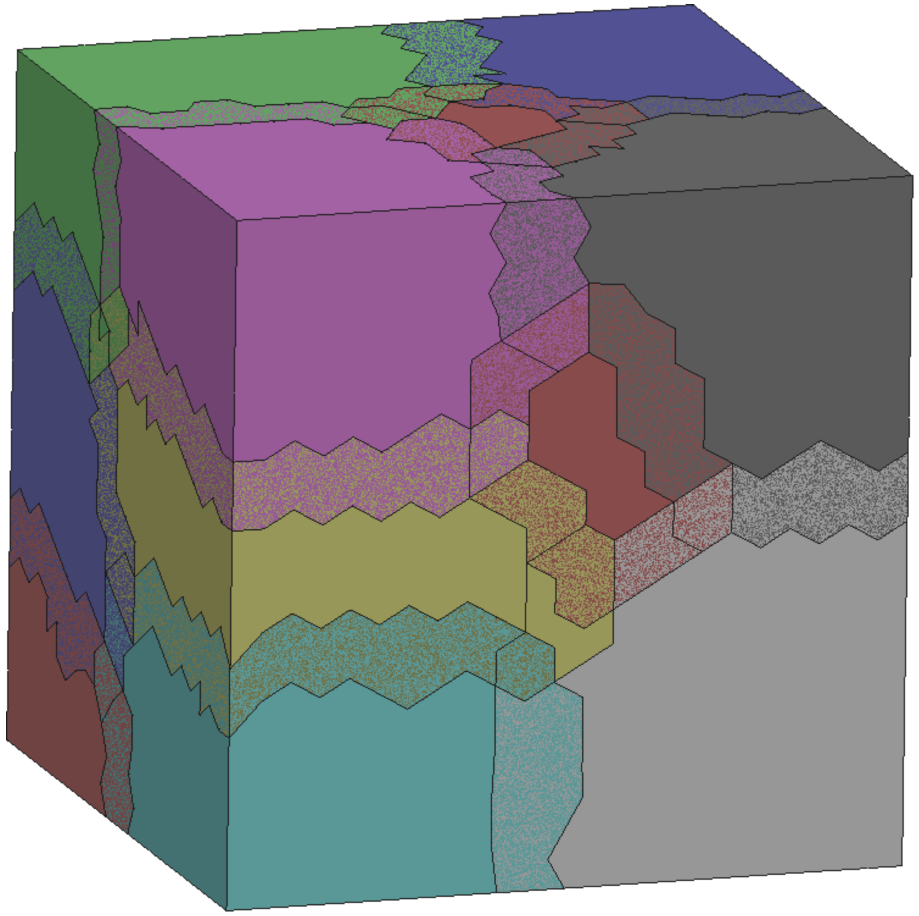
Our goal now is to solve big SPD systems of equations using parallel computing implemented in a Beowulf cluster. A group of multicore computers connected in a network.

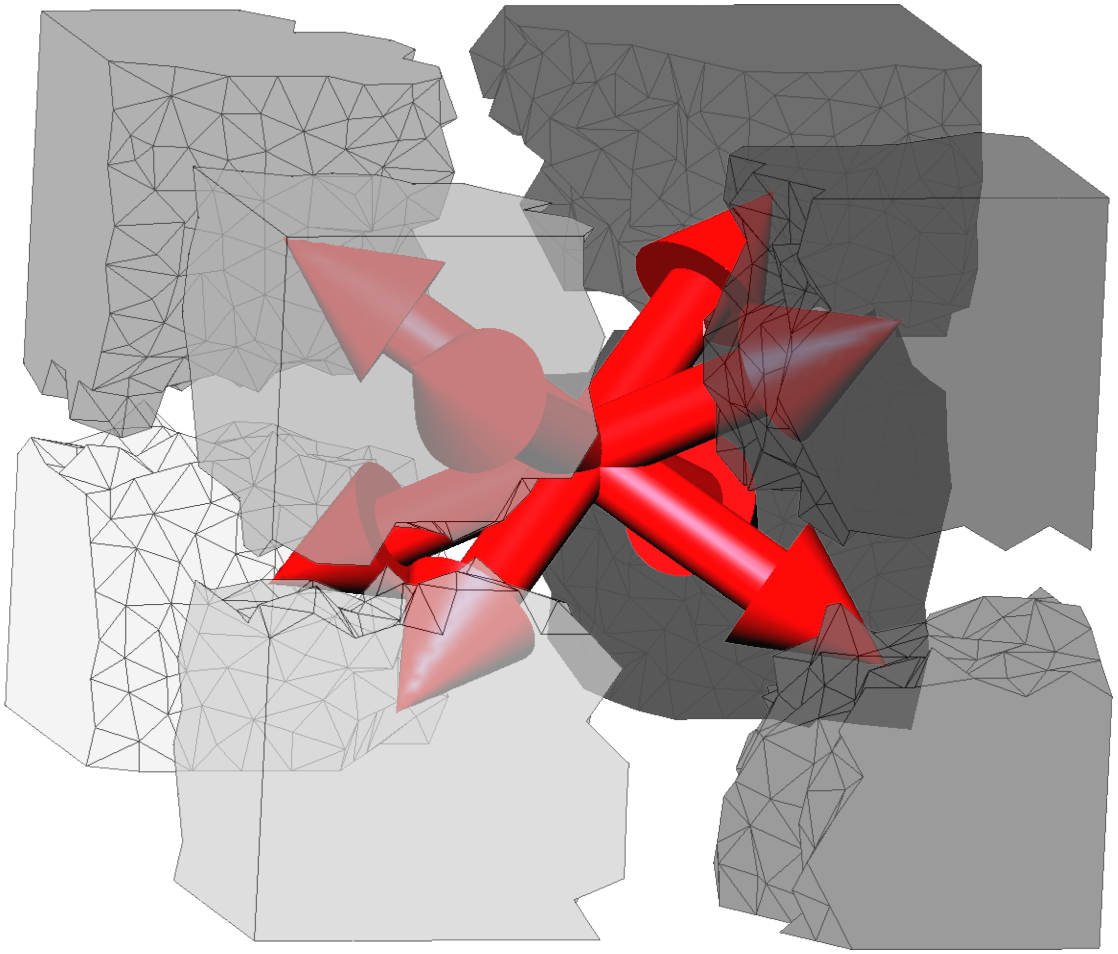


We will combine shared and distributed memory techniques.

“Divide et impera” with MPI

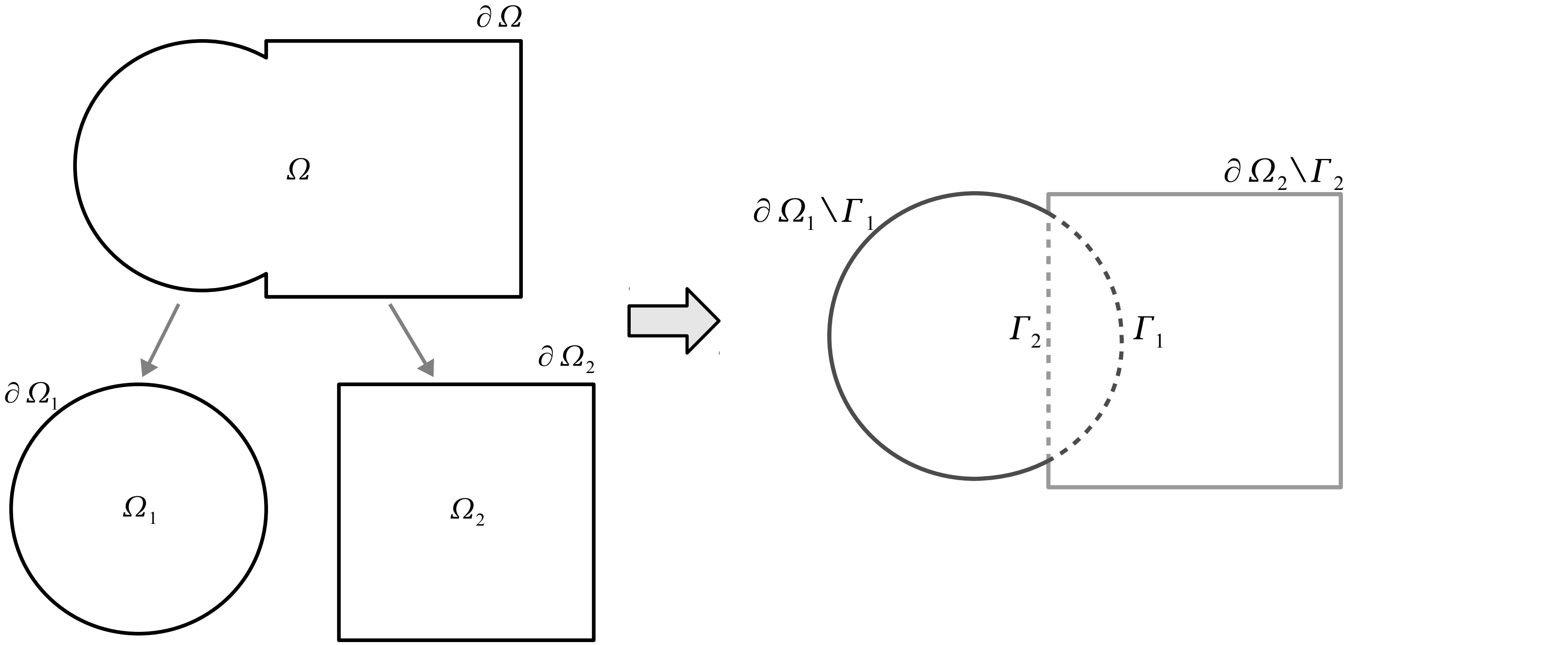
To distribute workload in big problems we partitionate the domain into several subdomains.





Domain decomposition (Schwarz alternating method)

This is an iterative method to split a big problem in small problems.



We have a domain Ω with boundary $\partial\Omega$.

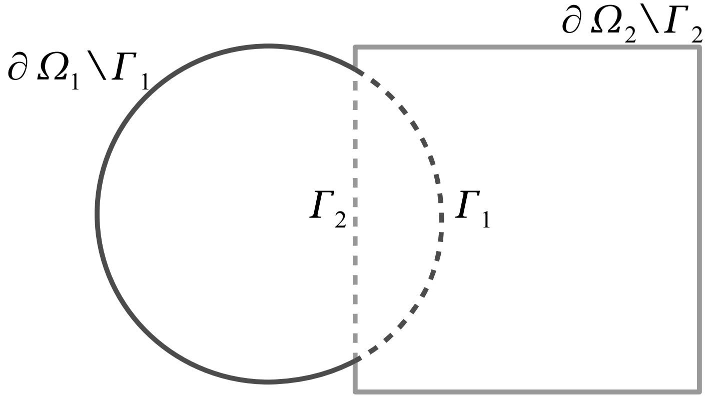
$\boldsymbol{L}$ is a differential operator $\boldsymbol{L}\boldsymbol{x}=\boldsymbol{y}$ in Ω.

Dirichlet conditions $\boldsymbol{x}=\boldsymbol{b}$ on $\partial\Omega$.

The domain is divided in two partitions Ω_1 and Ω_2 with boundaries $\partial\Omega_1$ and $\partial\Omega_2$.

Partitions Ω_1 y Ω_2 are overlapped, $\Omega=\Omega_1\cup\Omega_2$.

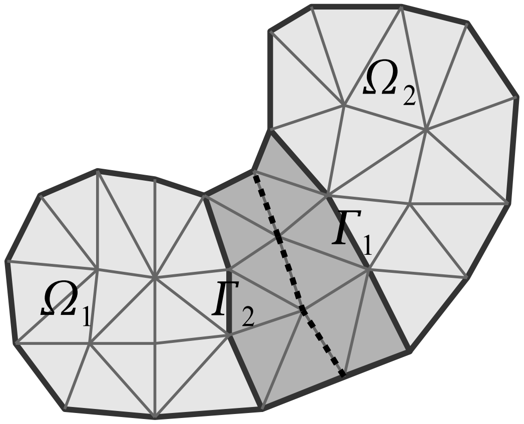
Γ_1 and Γ_2 are artificial boundaries, they belong to $\partial\Omega_1$ and $\partial\Omega_2$ and exist inside Ω.

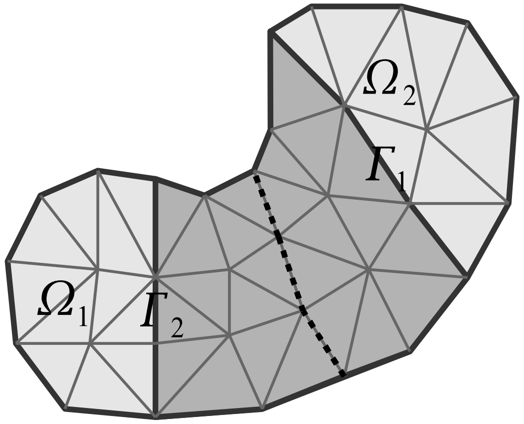


$\partial\Omega_1\setminus\Gamma_1$
$\partial\Omega_2\setminus\Gamma_2$
Γ_2
Γ_1

```
$\boldsymbol{x}_1^0, \boldsymbol{x}_2^0$ initial approximations
while $\|\boldsymbol{x}_1^i-\boldsymbol{x}_1^{i-1}\|>\varepsilon$ or $\|\boldsymbol{x}_2^i-\boldsymbol{x}_2^{i-1}\|>\varepsilon$
    Solve   $\boldsymbol{L}\boldsymbol{x}_1^i=\boldsymbol{y}$   in $\Omega_1$                              Solve   $\boldsymbol{L}\boldsymbol{x}_2^i=\boldsymbol{y}$   in $\Omega_2$
      with  $\boldsymbol{x}_1^i=\boldsymbol{b}$   in $\partial\boldsymbol{\Omega}_1\setminus\boldsymbol{\Gamma}_1$          with  $\boldsymbol{x}_2^i=\boldsymbol{b}$   in $\partial\boldsymbol{\Omega}_2\setminus\boldsymbol{\Gamma}_2$
    Update  $\boldsymbol{x}_1^i\leftarrow\boldsymbol{x}_2^{i-1}|_{\Gamma_1}$   in $\boldsymbol{\Gamma}_1$                  Update  $\boldsymbol{x}_2^i\leftarrow\boldsymbol{x}_1^{i-1}|_{\Gamma_2}$   in $\boldsymbol{\Gamma}_2$
$i\leftarrow i+1$
```

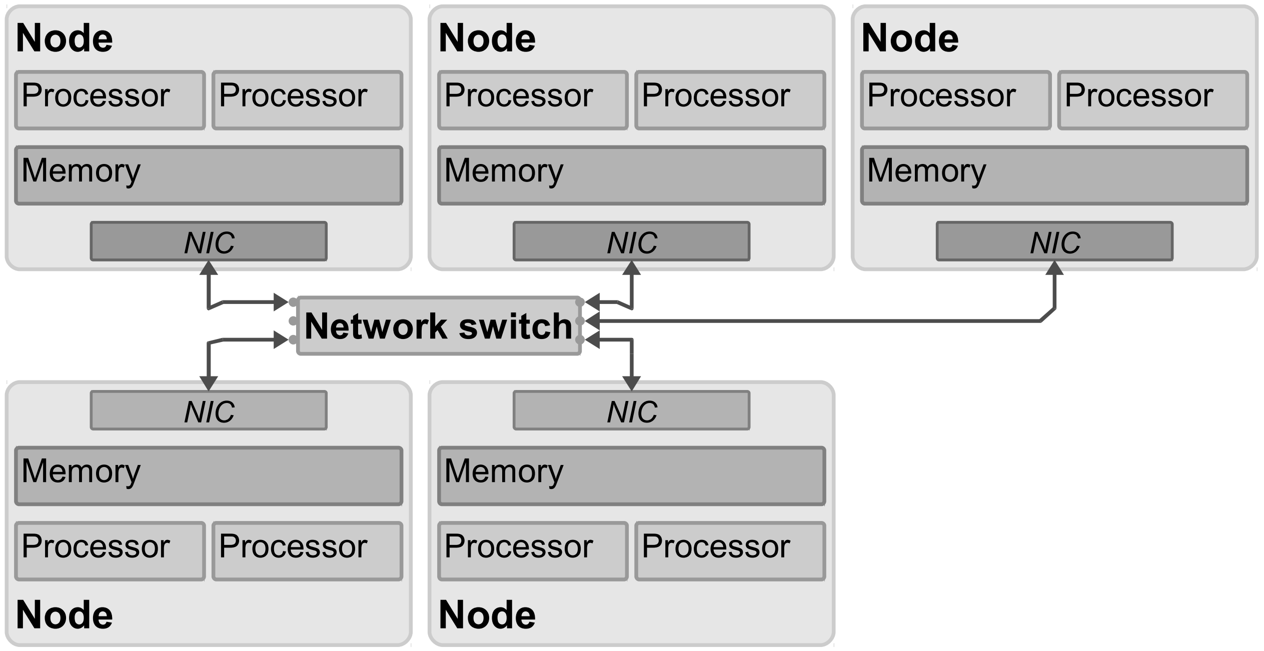
Adding overlapping improves convergence:



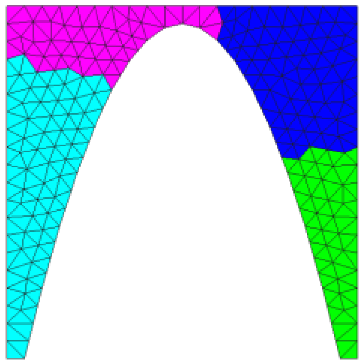


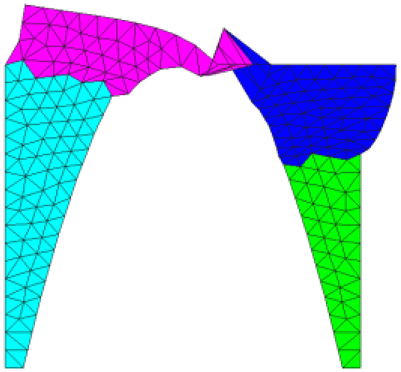
Implementation using MPI

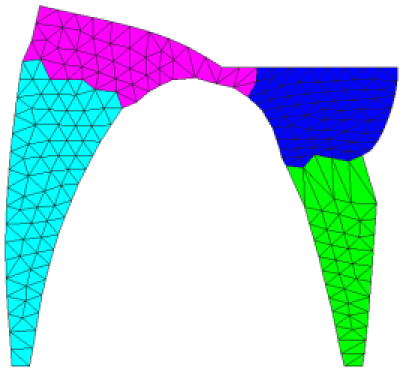
MPI (Message Passing Interface is a set of routines and programs to make easy to implement and administrate applications that ejecute several processes at the same time. It has routines to transmit data with great efficiency. It could be used in Beowulf clusters.

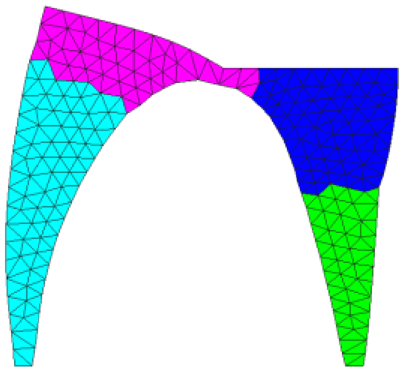


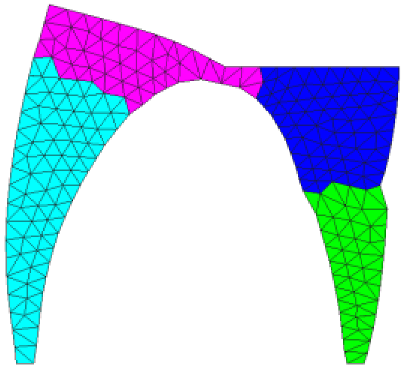
- The idea is to assing a MPI process to handle and solve each partition.
- OpenMP is used in each MPI process to solve the system of equations of the partition.
- Values on artificial boundaries are interchanged using MPI message routines.
- Schwarz iterations continue until global convergence is achived.
- For partitioning we used METIS library.

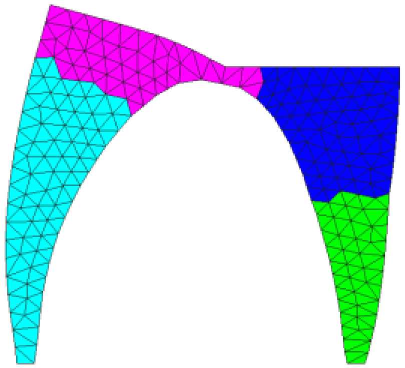

0000

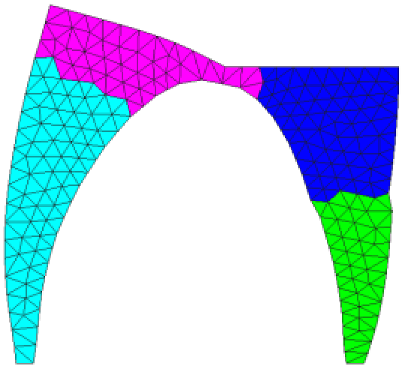

0001

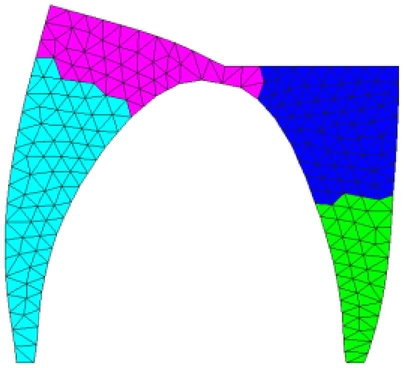

0002

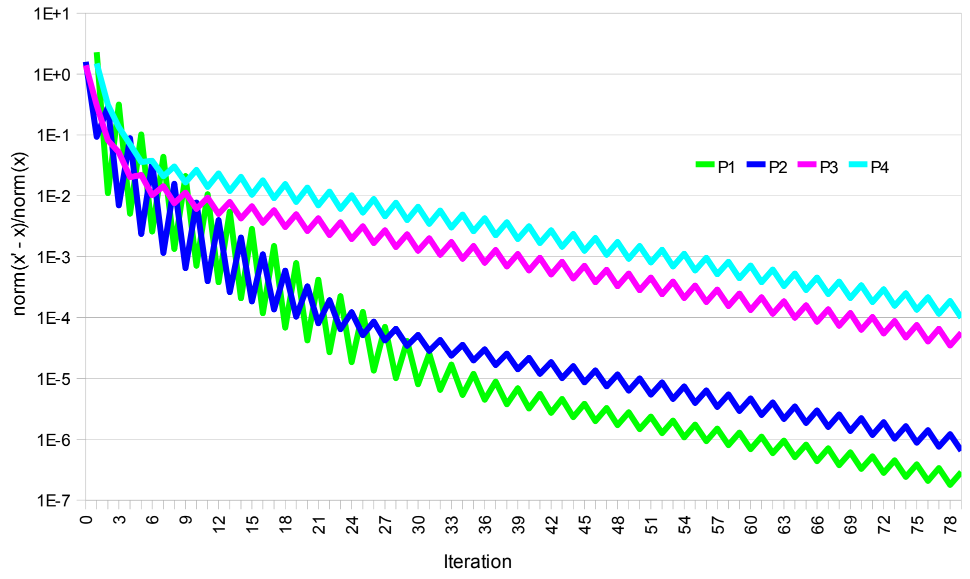

0003


0004


0005


0006

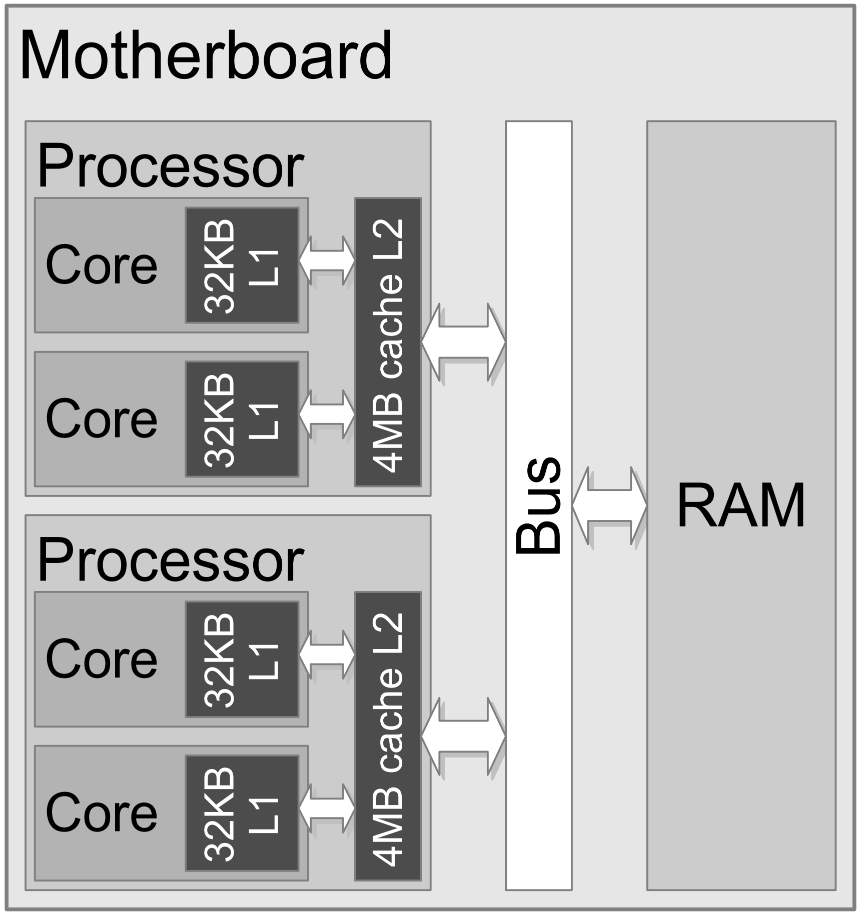

0007



$$d^i = \frac{\left\| \boldsymbol{u}^i_{t-1} - \boldsymbol{u}^i_t \right\|}{\left\| \boldsymbol{u}^i_t \right\|}$$

Solving SPD sparse systems of equations using shared memory

A simple but powerful way to program a multicore computer with shared memory is OpenMP.



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 importat issue to achieve high performance is to use the cache efficiently.

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

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

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

Parallel preconditioned conjugate gradient for sparse matrices

Preconditioning $\boldsymbol{M}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}=0)$ is used to improve CG convergence.

Preconditioners must be sparse.

```
$r_0 \leftarrow b - A x_0$, initial residual
$p_0 \leftarrow M r_0$, initial descent direction
$k \leftarrow 0$
while $\|r_k\| > \varepsilon$
    $\alpha_k \leftarrow -\frac{r_k^T M r_k}{p_k^T A p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
    $\beta_k \leftarrow \frac{r_{k+1}^T M r_{k+1}}{r_k^T M r_k}$
    $p_{k+1} \leftarrow M r_{k+1} + \beta_k p_k$
    $k \leftarrow k+1$
```

We tested three different preconditioners:

- Jacobi $\boldsymbol{M}^{-1}=(\mathrm{diag}(\boldsymbol{A}))^{-1}$.
- Incomplete Cholesky factorization $\boldsymbol{M}^{-1}=\boldsymbol{G}_l\boldsymbol{G}_l^T$, $\boldsymbol{G}_l\approx\boldsymbol{L}$.
- Factorized sparse approximate inverese $\boldsymbol{M}=\boldsymbol{H}_l^T\boldsymbol{H}_l$, $\boldsymbol{H}_l\approx\boldsymbol{L}^{-1}$.

Matrix-vector and dot products are parallelized using OpenMP.

Compress row storage method is used to store matrices.

Only non-zero entries and their indexes are stored.

Entries in each row are contiguos and sorted to improve cache access.

Each processor core works with a group of rows to parallelize the operations.

It looks like:

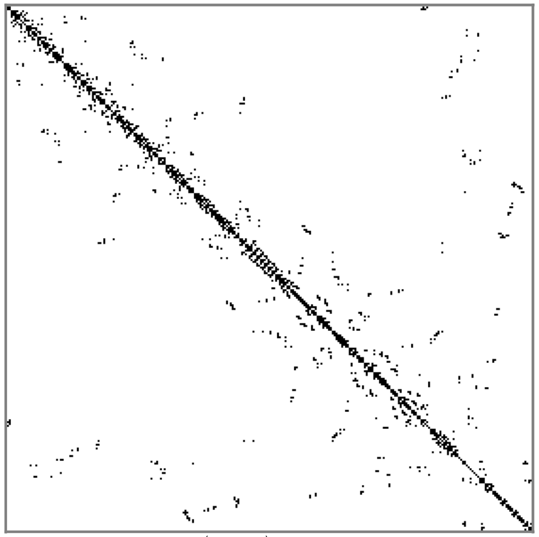
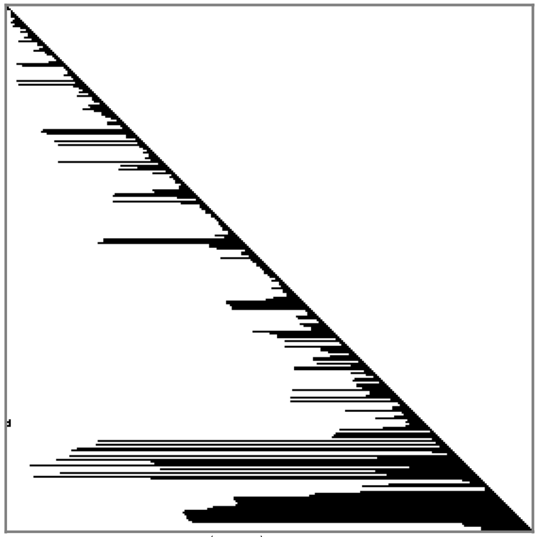
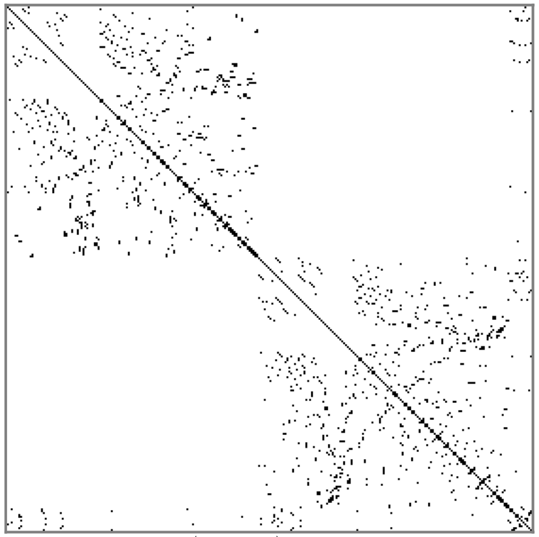
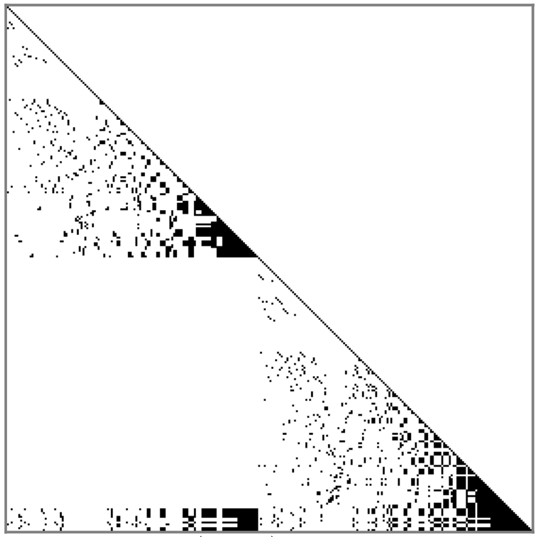
```
Vector<T> g(n); // Gradient
Vector<T> p(n); // Descent direcction
Vector<T> w(n); // w = A*p

double gg = 0.0;
#pragma omp parallel for default(shared) reduction(+:gg) schedule(guided)
for (int i = 1; i <= n; ++i)
{
    int* __restrict A_index_i = A.index[i];
    double* __restrict A_entry_i = A.entry[i];

    double sum = 0.0;
    int k_max = A.count[i];
    for (register int k = 1; k <= k_max; ++k)
    {
        sum += A_entry_i[k]*x.entry[A_index_i[k]];
    }
    g.entry[i] = sum - b.entry[i]; // g = AX - b;
    p.entry[i] = -g.entry[i]; // p = -g
    gg += g.entry[i]*g.entry[i]; // gg = g'*g
}

int step = 0;
while (step < max_steps)
{
    if (Norm(gg) <= tolerance) // Test termination condition
    {
        break;
    }

    double pw = 0.0;
    #pragma omp parallel for default(shared) reduction(+:pw) schedule(guided)
    for (int i = 1; i <= n; ++i)
    {
        int* __restrict A_index_i = A.index[i];
        double* __restrict A_entry_i = A.entry[i];

        double sum = 0.0;
        int k_max = A.count[i];
        for (register int k = 1; k <= k_max; ++k)
        {
            sum += A_entry_i[k]*p.entry[A_index_i[k]];
        }
        w.entry[i] = sum; // w = AP
        pw += p.entry[i]*w.entry[i]; // pw = p'*w
    }

    double alpha = gg/pw; // alpha = (g'*g)/(p'*w)

    double gngn = 0.0;
    #pragma omp parallel for default(shared) reduction(+:gngn)
    for (int i = 1; i <= n; ++i)
    {
        x.entry[i] += alpha*p.entry[i]; // Xn = x + alpha*p
        g.entry[i] += alpha*w.entry[i]; // Gn = g + alpha*w
        gngn += g.entry[i]*g.entry[i]; // gngn = Gn'*Gn
    }

    double beta = gngn/gg; // beta = (Gn'*Gn)/(g'*g)

    #pragma omp parallel for default(shared)
    for (int i = 1; i <= n; ++i)
    {
        p.entry[i] = beta*p.entry[i] - g.entry[i]; // Pn = -g + beta*p
    }

    gg = gngn;
    ++step;
}
```

Parallel Cholesky factorización for sparse matrices

$\boldsymbol{K}=\boldsymbol{L}\,\boldsymbol{L}^{\mathrm{T}}$, it is expensive to store and calculate $\boldsymbol{L}$ entries

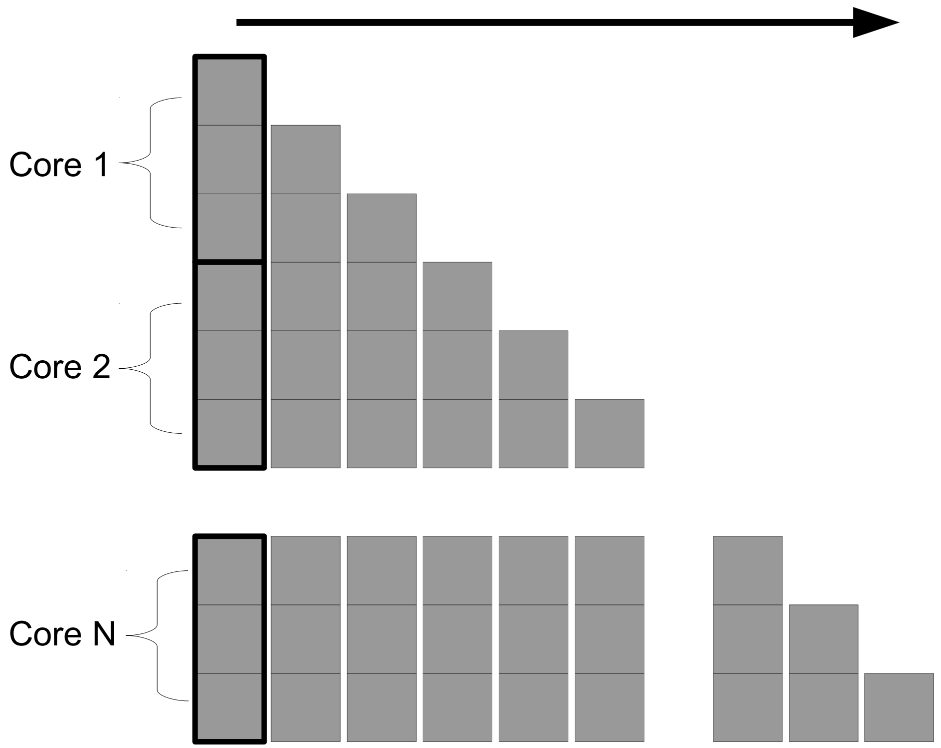
$$L_{ij}=\frac{1}{L_{jj}}\left(K_{ij}-\sum_{k=1}^{j-1} L_{ik}\,L_{jk}\right), \text{ for } i>j$$

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

Stiffnes matrix	Cholesky factorization
 $\mathrm{nnz}(\boldsymbol{K})=1810$	 $\mathrm{nnz}(\boldsymbol{L})=8729$
 $\mathrm{nnz}(\boldsymbol{K}')=1810$	 $\mathrm{nnz}(\boldsymbol{L}')=3215$

We used several strategies to make Cholesky factorization efficient:

- Matrix ordering to reduce factorization ***fill-in***. Minimum degree algorithm or nested disection algorithm (METIS library).
- Symbolic Cholesky factorization to determine non-zero entries before calculation.
- Factorization matrix is stored using compress row storage to improve forward-substitution.
- $\boldsymbol{L}^{\mathrm{T}}$ is stored to improve speed of back-substitution.
- The fill of each column of $\boldsymbol{L}$ is calculated in parallel using OpenMP.



This algorithm is also used to generate the incomplete Cholesky preconditioner.

It looks like:

```
for (int j = 1; j <= L.columns; ++j)
{
    double* __restrict L_entry_j = L.entry[j];
    double* __restrict Lt_entry_j = Lt.entry[j];

    double L_jj = A(j, j);
    int L_count_j = L.count[j];
    for (register int q = 1; q < L_count_j; ++q)
    {
        L_jj -= L_entry_j[q]*L_entry_j[q];
    }
    L_jj = sqrt(L_jj);
    L_entry_j[L_count_j] = L_jj; // L(j)(j) = sqrt(A(j)(j)-sum(k=1, j-1, L(j)(k)*L(j)(k)))
    Lt_entry_j[1] = L_jj;

    int Lt_count_j = Lt.count[j];
    #pragma omp parallel for default(shared) schedule(guided)
    for (int q = 2; q <= Lt_count_j; ++q)
    {
        int i = Lt.index[j][q];

        double* __restrict L_entry_i = L.entry[i];

        double L_ij = A(i, j);

        const register int* __restrict L_index_j = L.index[j];
        const register int* __restrict L_index_i = L.index[i];

        register int qi = 1;
        register int qj = 1;
        register int ki = L_index_i[qi];
        register int kj = L_index_j[qj];
        for (bool next = true; next; )
        {
            while (ki < kj)
            {
                ++qi;
                ki = L_index_i[qi];
            }
            while (ki > kj)
            {
                ++qj;
                kj = L_index_j[qj];
            }
            while (ki == kj)
            {
                if (ki == j)
                {
                    next = false;
                    break;
                }
                L_ij -= L_entry_i[qi]*L_entry_j[qj];
                ++qi;
                ++qj;
                ki = L_index_i[qi];
                kj = L_index_j[qj];
            }
        }
        L_ij /= L_jj;
        L_entry_i[qi] = L_ij;  // L(i)(j) = (A(i)(j)-sum(k = 1, j-1, L(i)(k)*L(j)(k)))/L(j)(j)
        Lt_entry_j[q] = L_ij;
    }
}
```

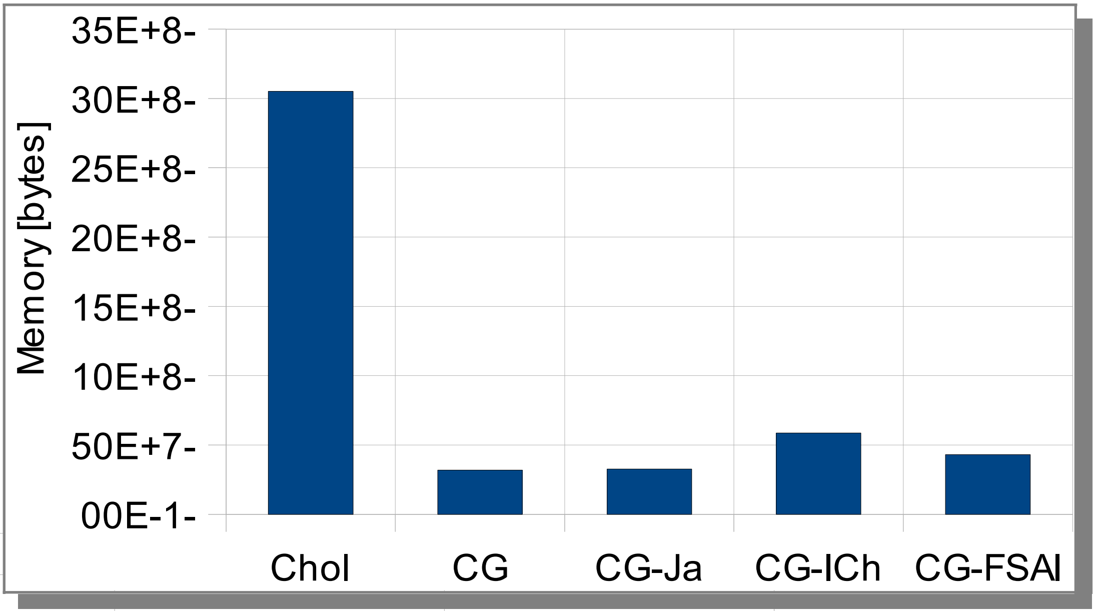
2D Solid (OpenMP only)

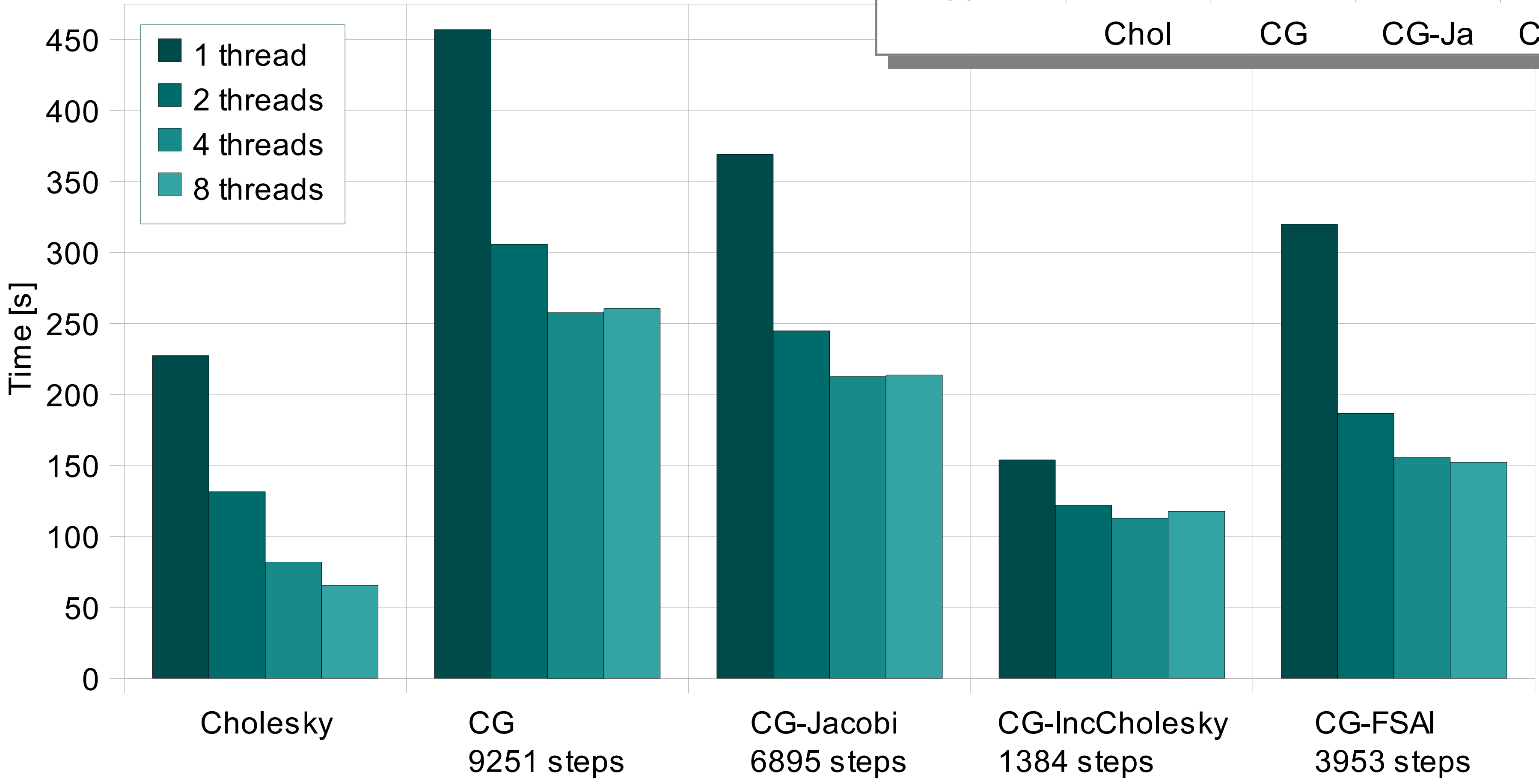
Next results are from a 2D solid problem with 1,005,362 equations.

$\text{nnz}(\boldsymbol{K}) = 18'062{,}500$

Tolerance used in CG algorithms is 1x10^{-5}

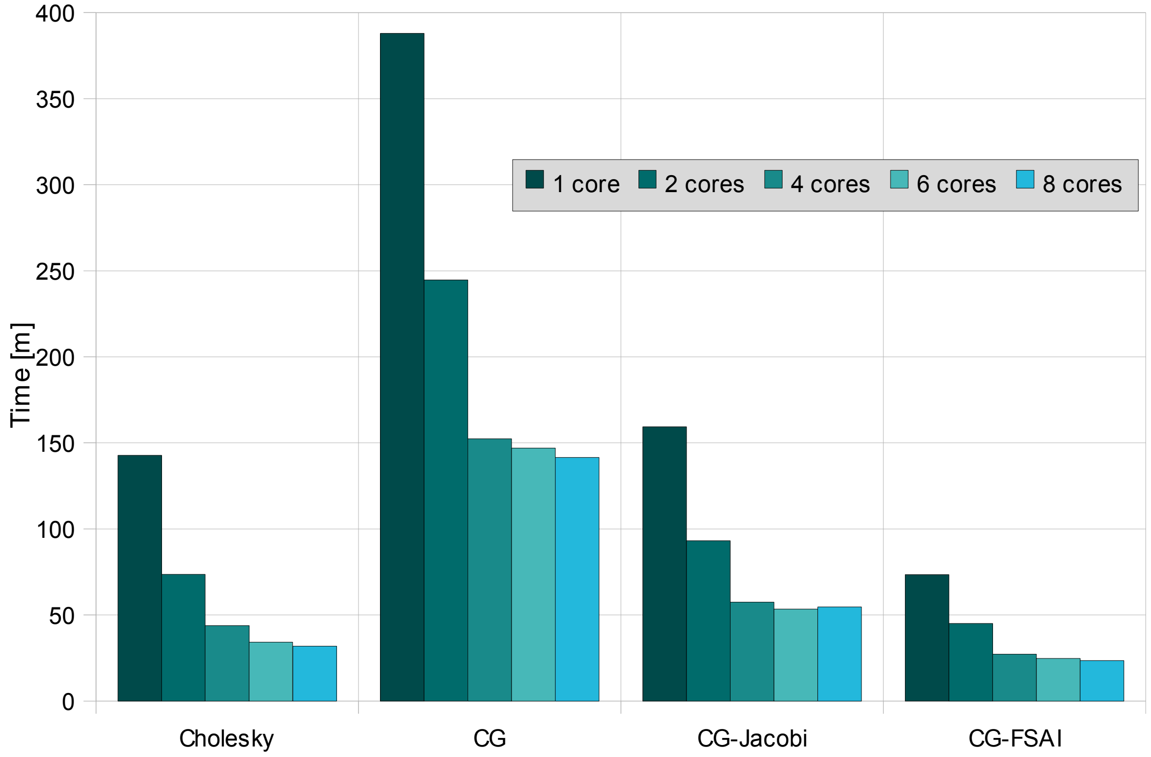
$\text{nnz}(\boldsymbol{L}) = 111'873{,}237$

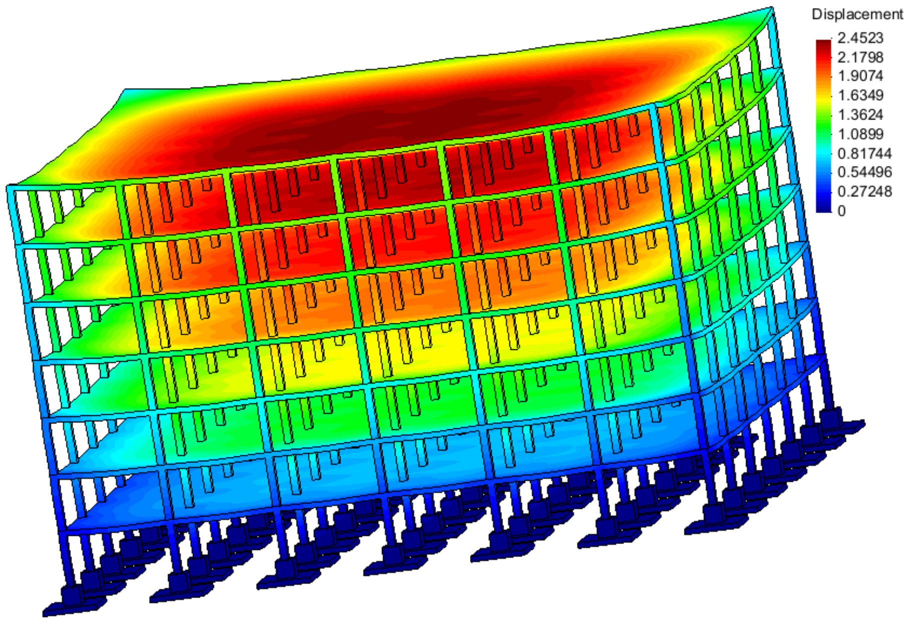




3D solid (OpenMP only)

Elements:	264,250
Nodes:	326,228
Variables:	978,684
nnz(A):	69,255,522
nnz(L):	787,567,656

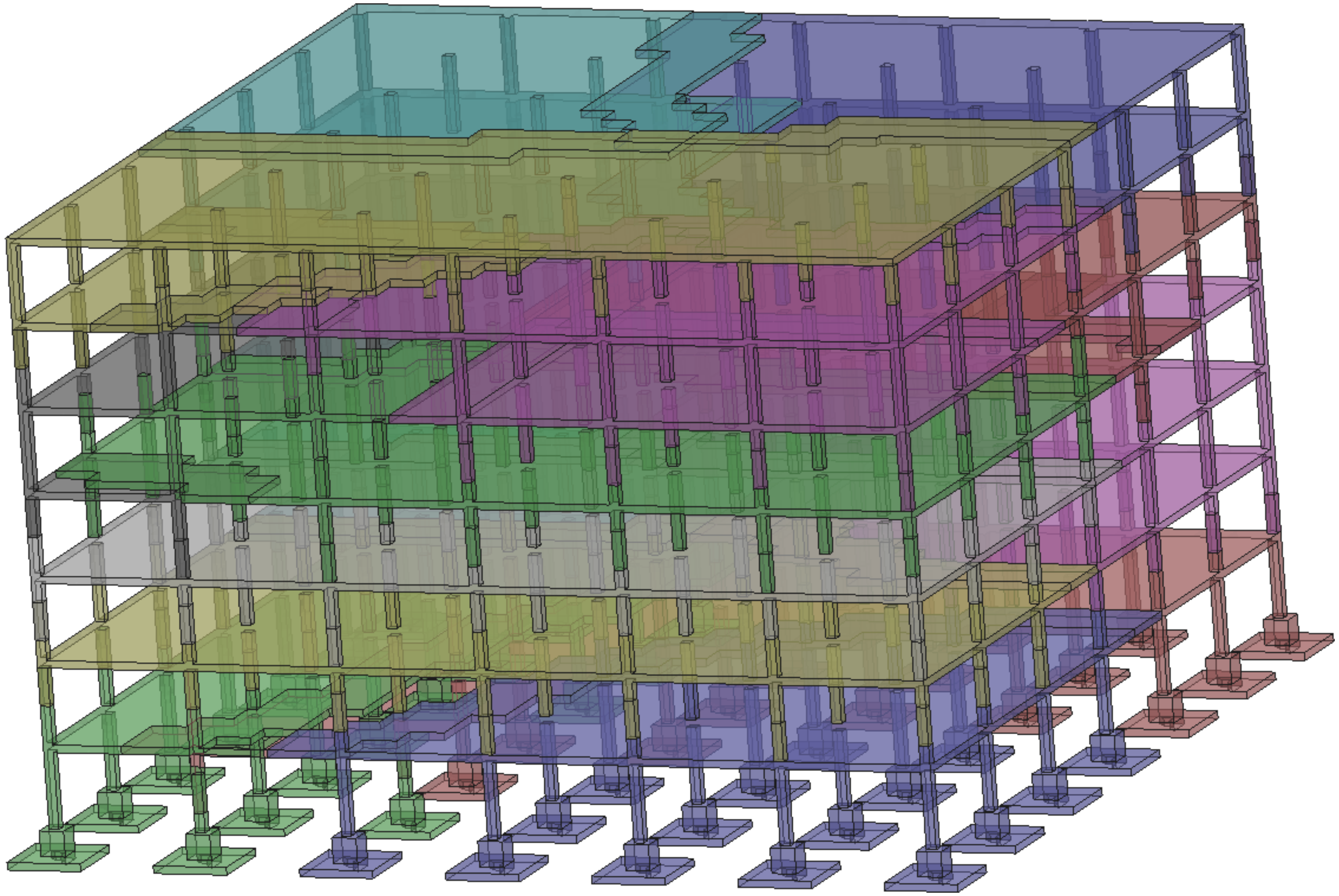




Solver	1 core [m]	2 cores [m]	4 cores [m]	6 cores [m]	8 cores [m]	Memory
Cholesky	142	73	43	34	31	19,864’132,056
CG	387	244	152	146	141	922’437,575
CG-Jacobi	159	93	57	53	54	923’360,936
CG-FSAI	73	45	27	24	23	1,440’239,572

Now with MPI+OpenMP

Simulation of a builiding that deformates by self-weight. Basement has fixed displacement. The domain was discretized in 264,250 elements, 326,228 nodes, 978,684 equations, nnz($\boldsymbol{K}$) = 69’255,522.

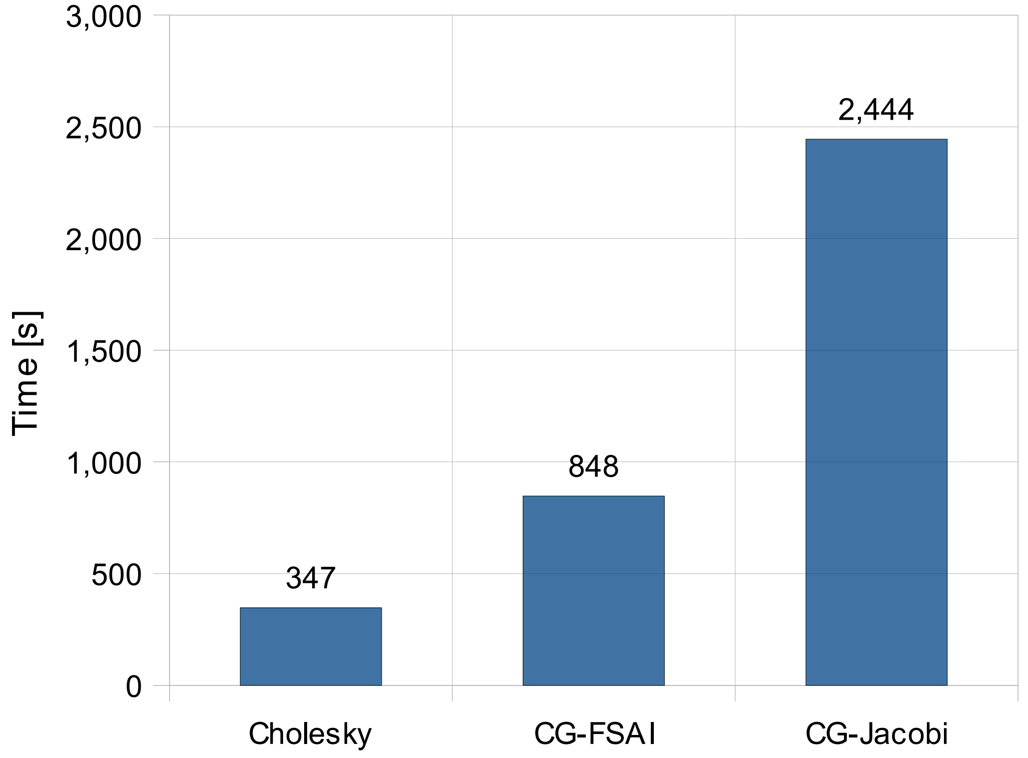


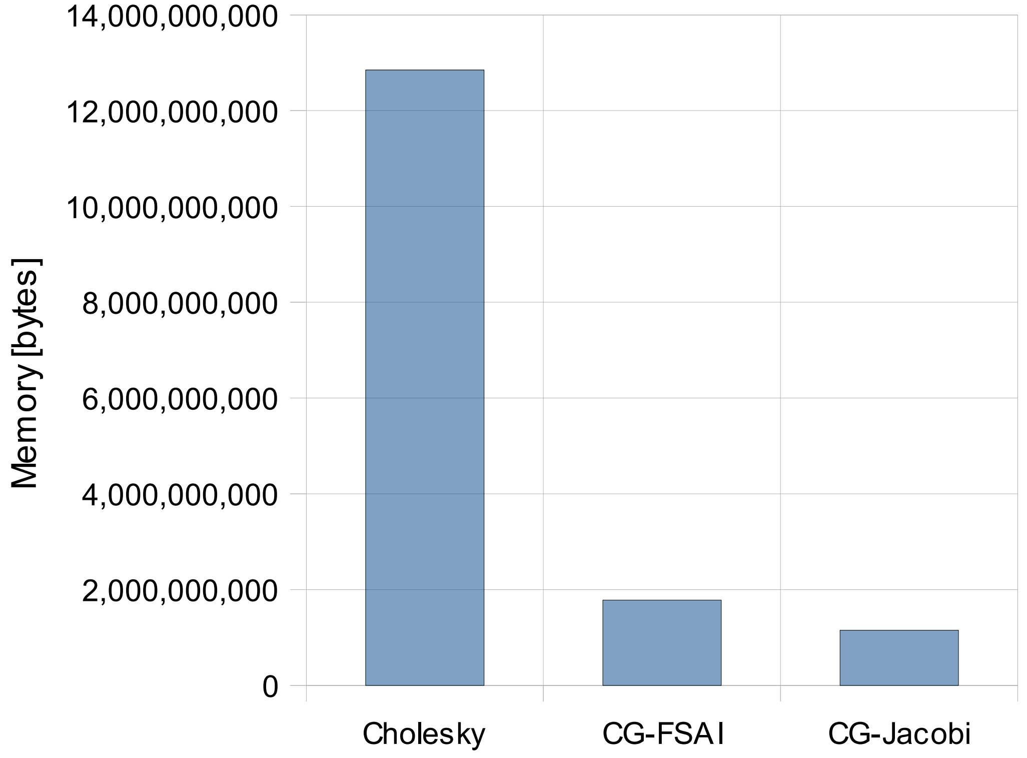
We will solve it using a combination of distributed and shared memory schemas (MPI+OpenMP).

Results, domain decomposition (14 partitions, 4 threads per solver)

Running in 14 computers each one with 4 cores.

Solver	Total time [s]	Total memory	Memory per slave
Cholesky	347	12,853,865,804	917,054,441
CG-FSAI	848	1,779,394,516	126,020,778
CG-Jacobi	2,444	1,149,968,796	81,061,798

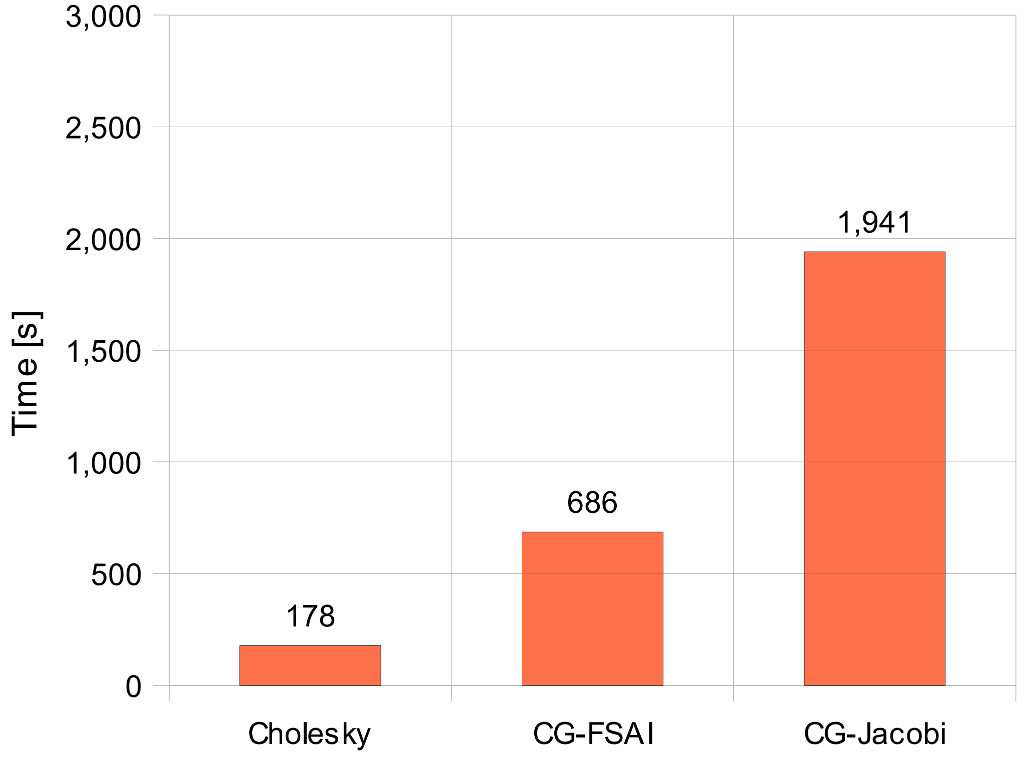


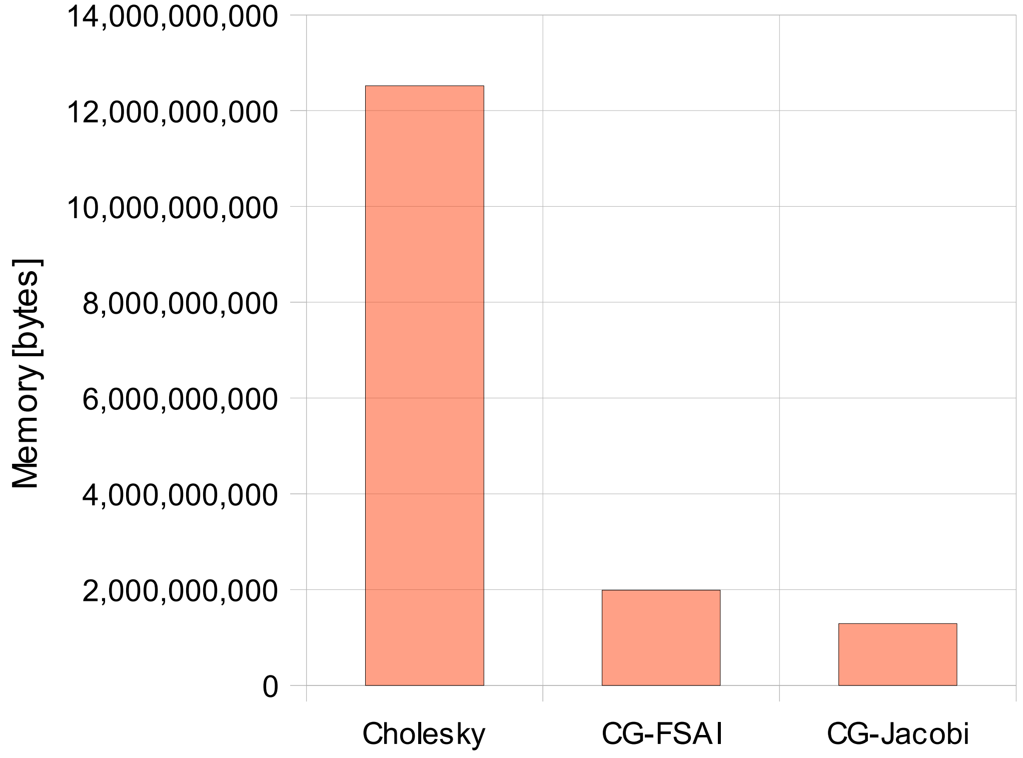


Results, domain decomposition (28 partitions, 2 threads per solver)

Running in 14 computers each one with 4 cores.

Solver	Total time [s]	Total memory	Memory per slave
Cholesky	178	12,520,198,517	893,173,100
CG-FSAI	686	1,985,459,829	140,691,765
CG-Jacobi	1,940	1,290,499,837	91,051,766

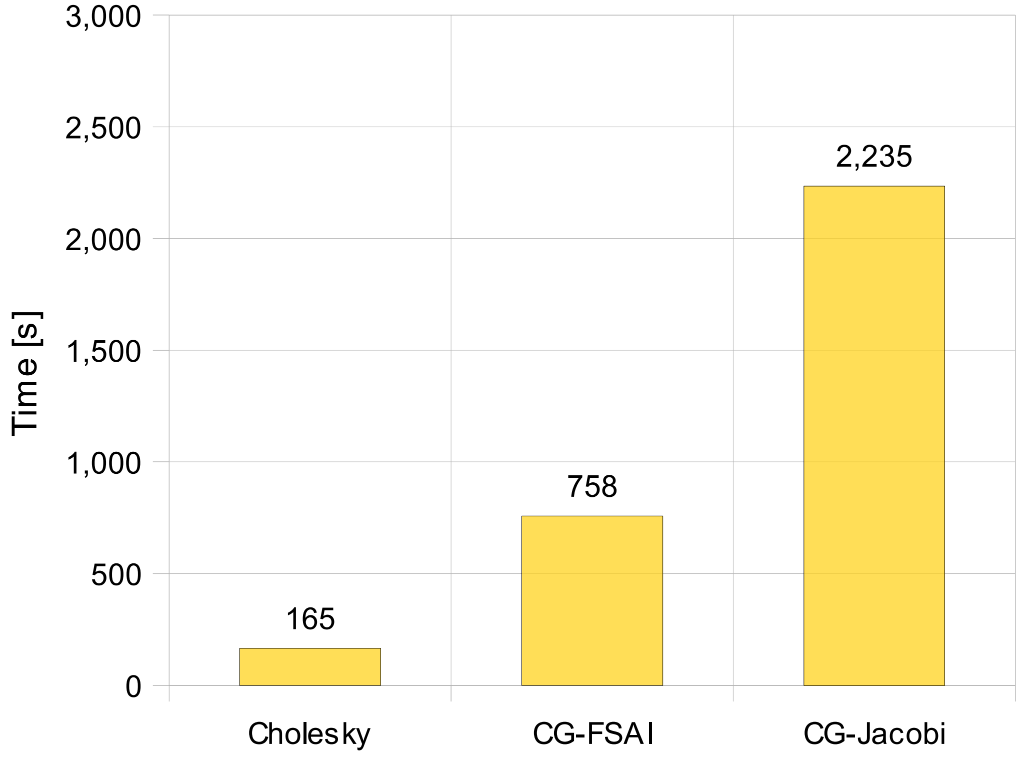


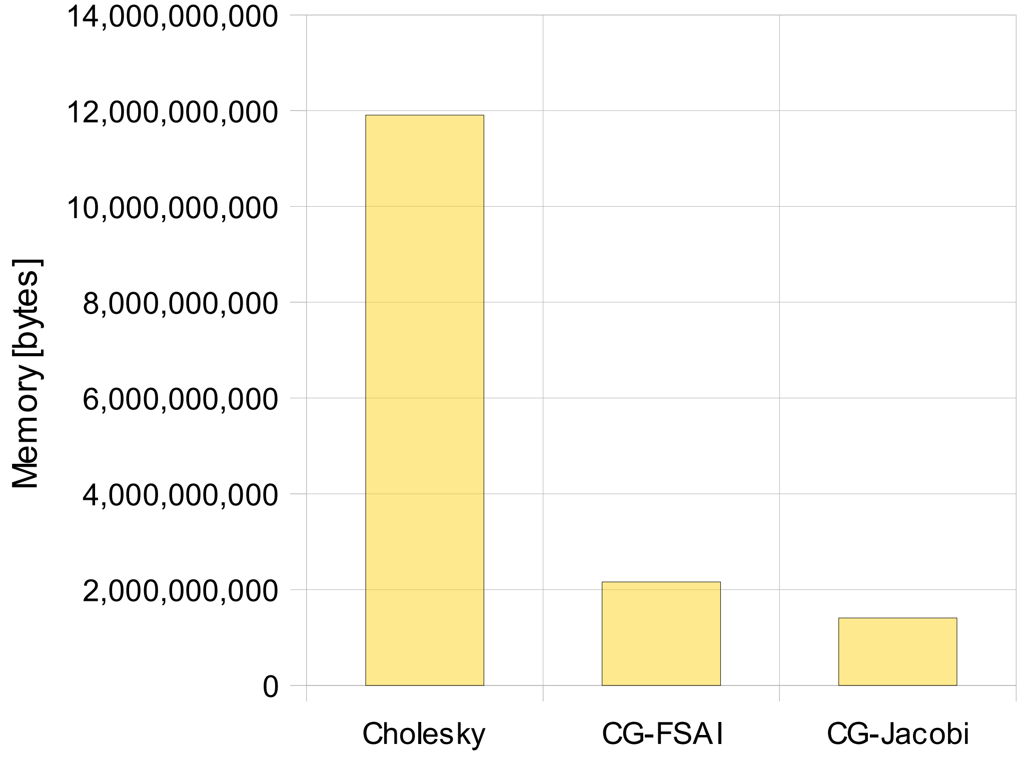


Results, domain decomposition (56 partitions, 1 thread per solver)

Running in 14 computers each one with 4 cores.

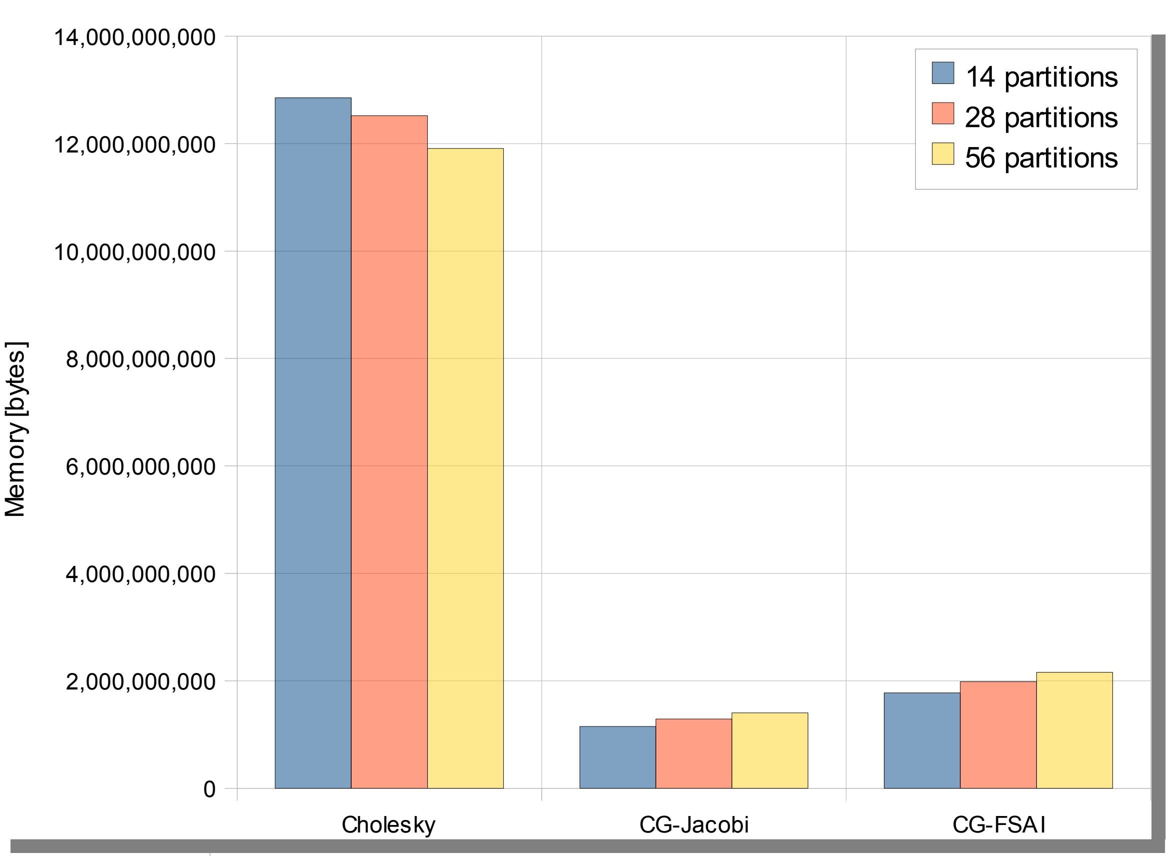
Solver	Total time [s]	Total memory	Memory per slave
Cholesky	165	11,906,979,912	849,323,496
CG-FSAI	758	2,156,224,760	152,840,985
CG-Jacobi	2,235	1,405,361,320	99,207,882

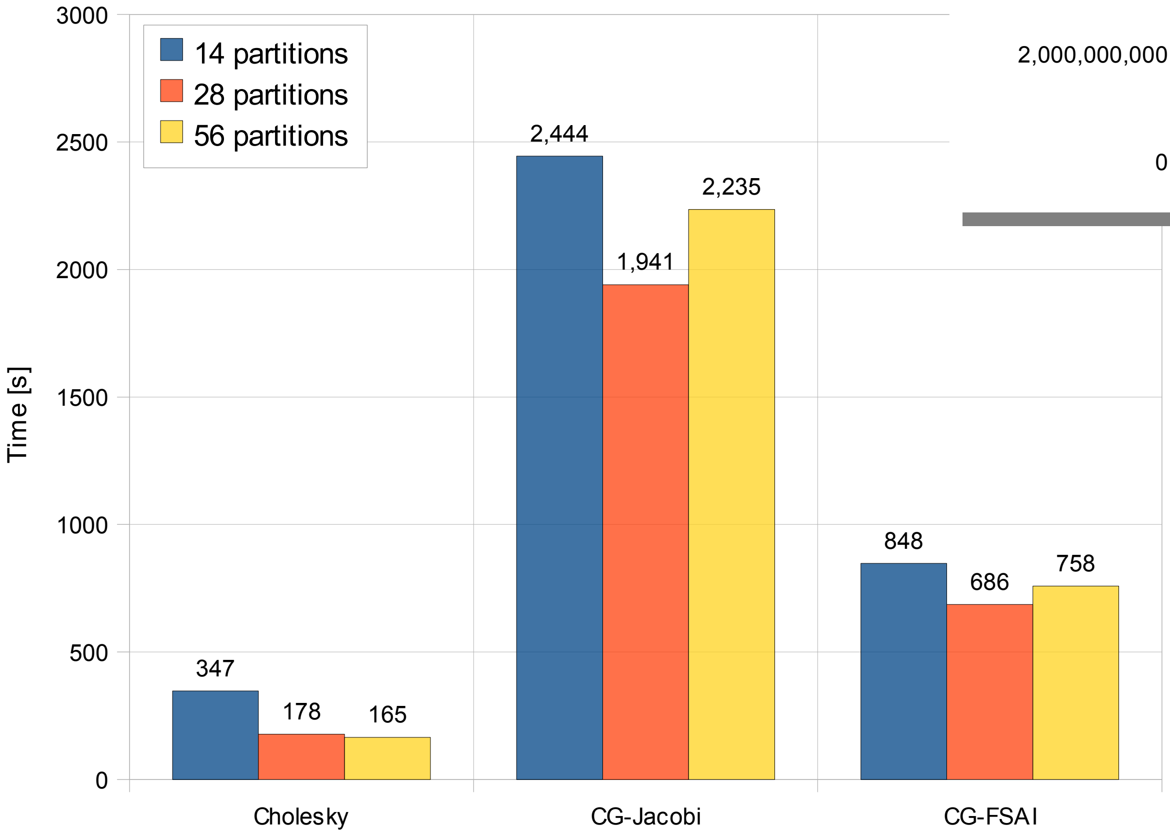




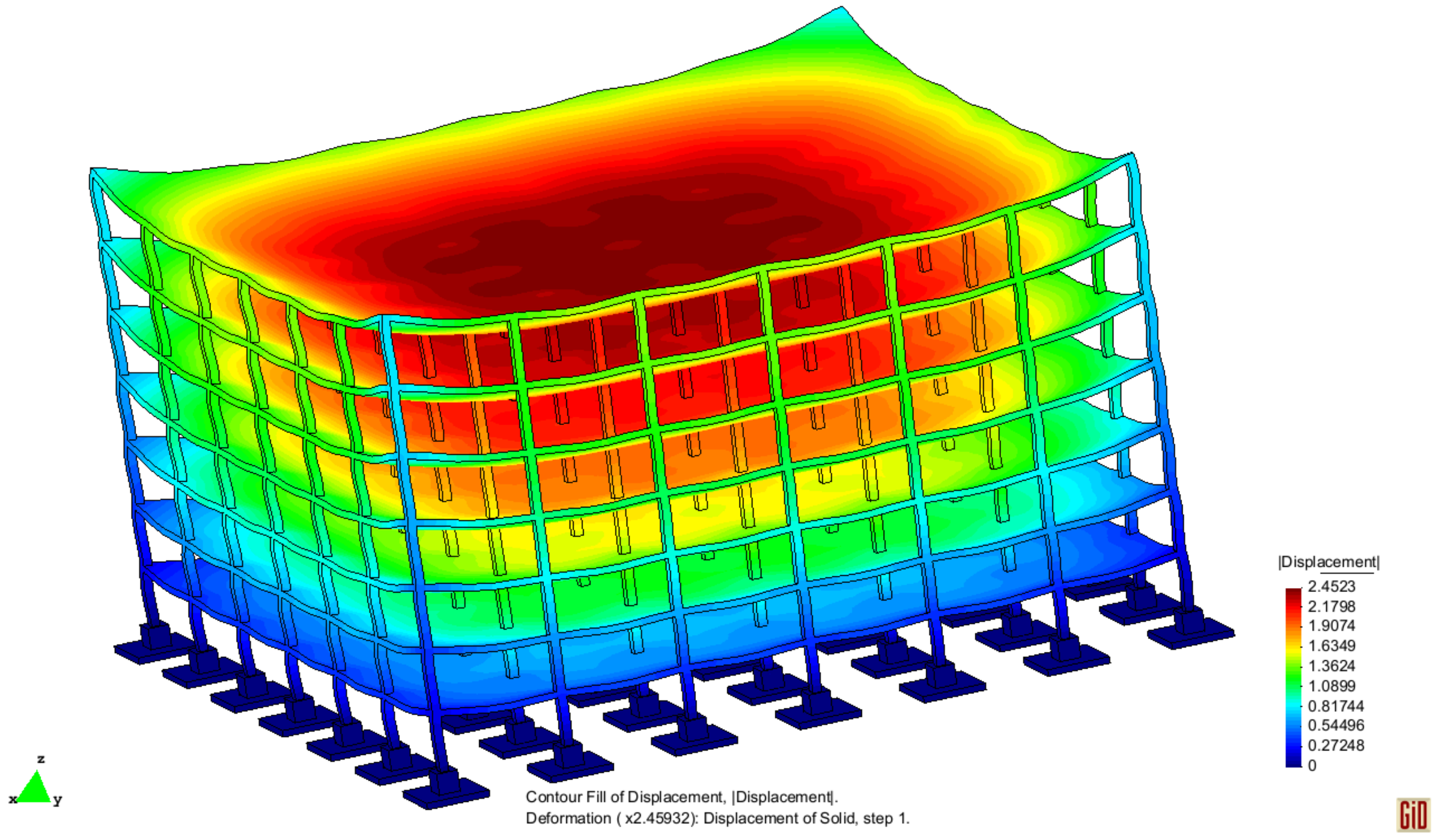
Comparison

264,250 elements,
326,228 nodes,
978,684 equations,
nnz($\boldsymbol{K}$) = 69’255,522.





A bigger example

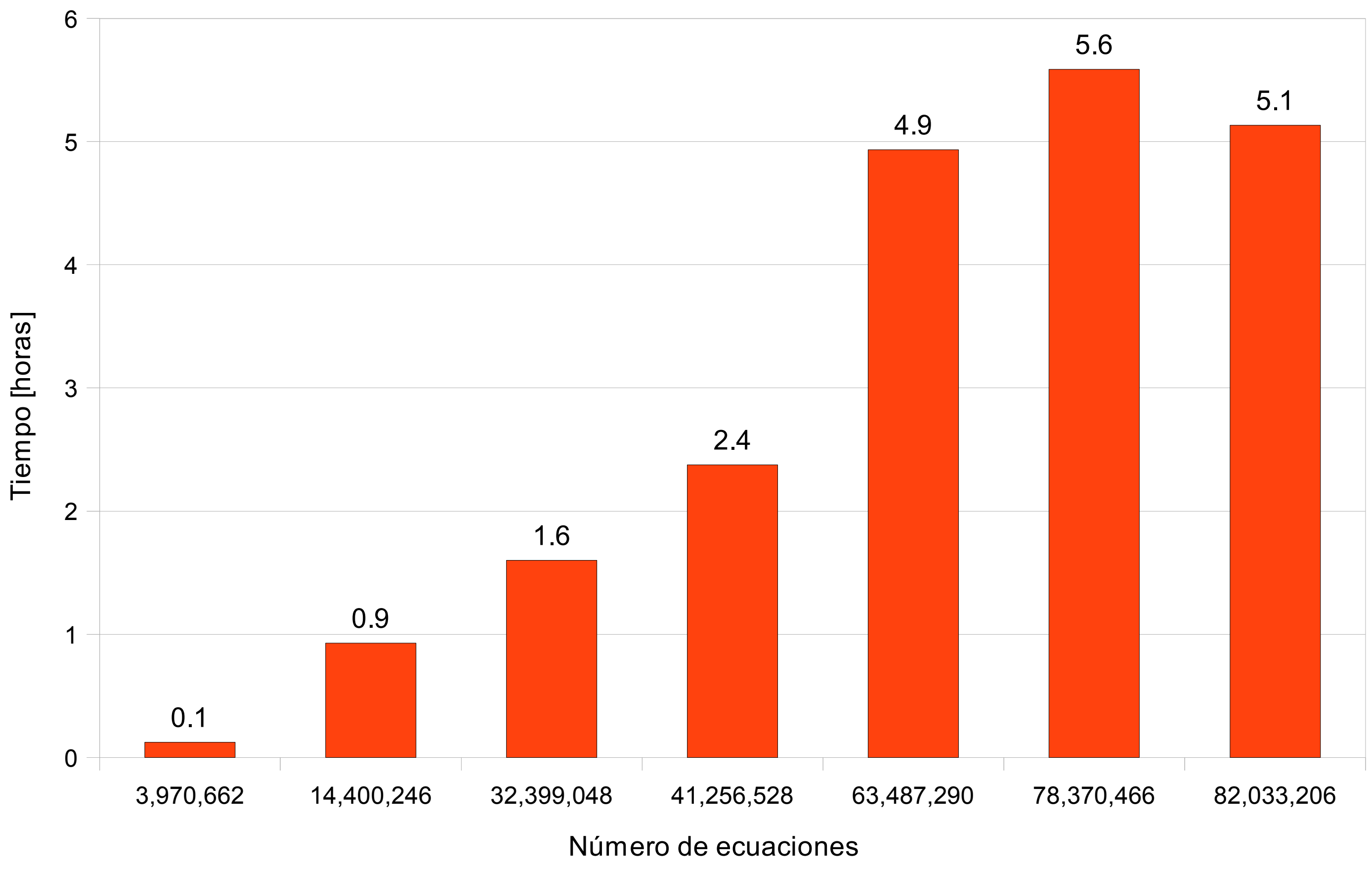

|Displacement|
2.4523
2.1798
1.9074
1.6349
1.3624
1.0899
0.81744
0.54496
0.27248
0
z
x
y
Contour Fill of Displacement, |Displacement|.
Deformation (x2.45932): Displacement of Solid, step 1.
GiD

Elements:	3’652,992
Nodes:	4’011,469
Equations:	12’034,407
nnz(K):	794’270,862
Partitions:	124
CPU cores:	128
Nodes:	31

Solver	Time [m]	Memory [GB]
Cholesky	130	260
CG	133,926	34
CG-Jacobi	42,510	34
CG-FSAI	11,239	42

Big systems of equations (2D solid)

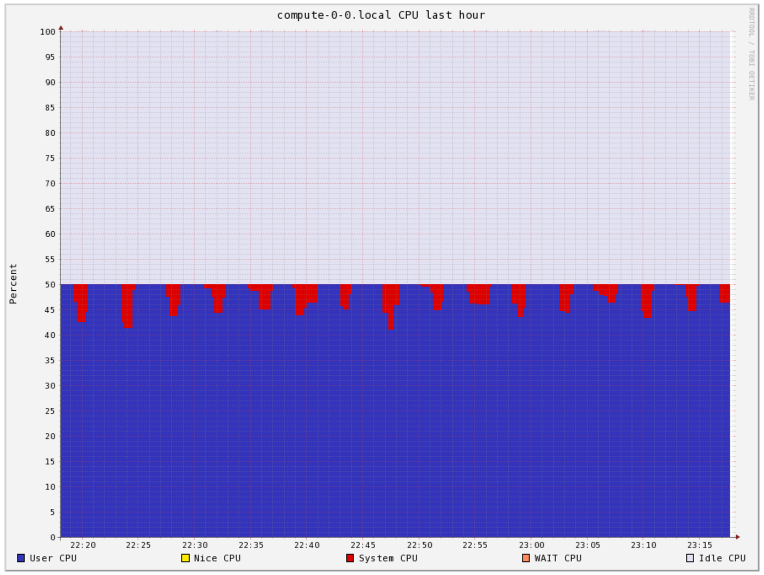
Equations	Time [h]	Memory [GB]	Cores	Overlap	Nodes
3,970,662	0.12	17	52	12	13
14,400,246	0.93	47	52	10	13
32,399,048	1.60	110	60	17	15
41,256,528	2.38	142	60	15	15
63,487,290	4.93	215	84	17	29
78,370,466	5.59	271	100	20	29
82,033,206	5.13	285	100	20	29

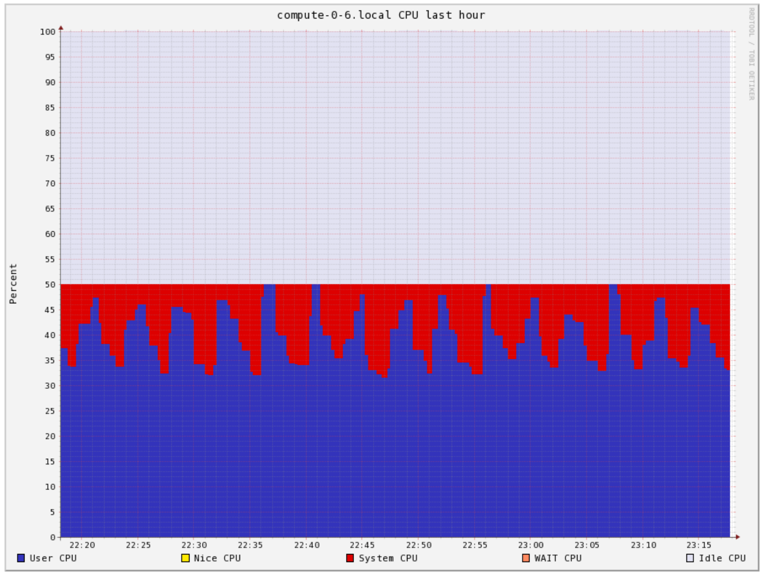


Lessons learned

We found that incomple Cholesky factorization is unstable for some matrices, it is posible to stabilize the solver making the preconditioner diagonal-domainant, but we have to use a heuristic.

The big issue with iterative solvers is load balancing:





It is complex to partition the domain in such way that local problems take the same time to be solved. This issue is less notizable when Cholesky solver is used.

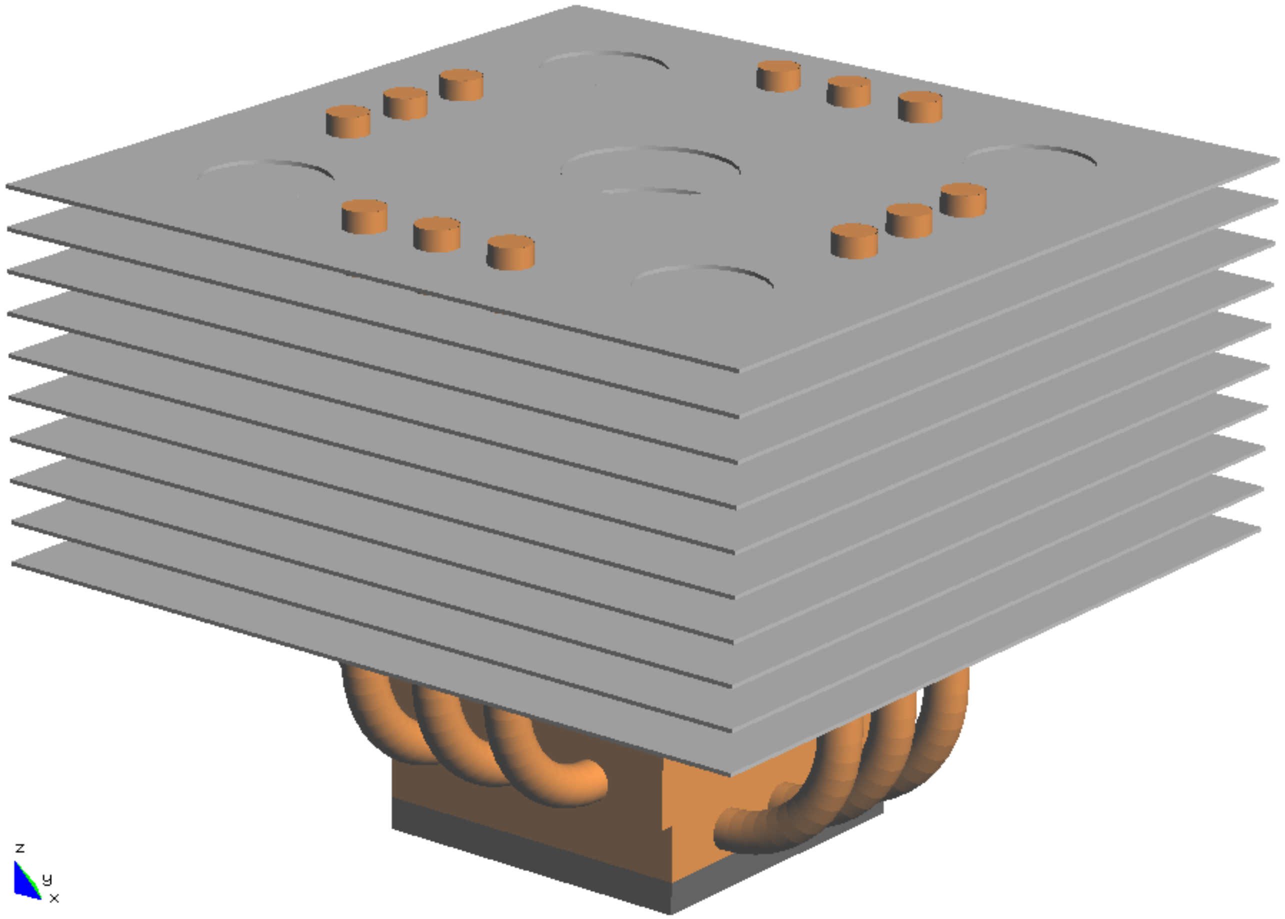
To split the problem using domain decomposition with Cholesky works well, the fastest configuration was using one thread per solver. The good news is that memory is getting cheaper.

If memory is a concern we can use CG with FSAI.

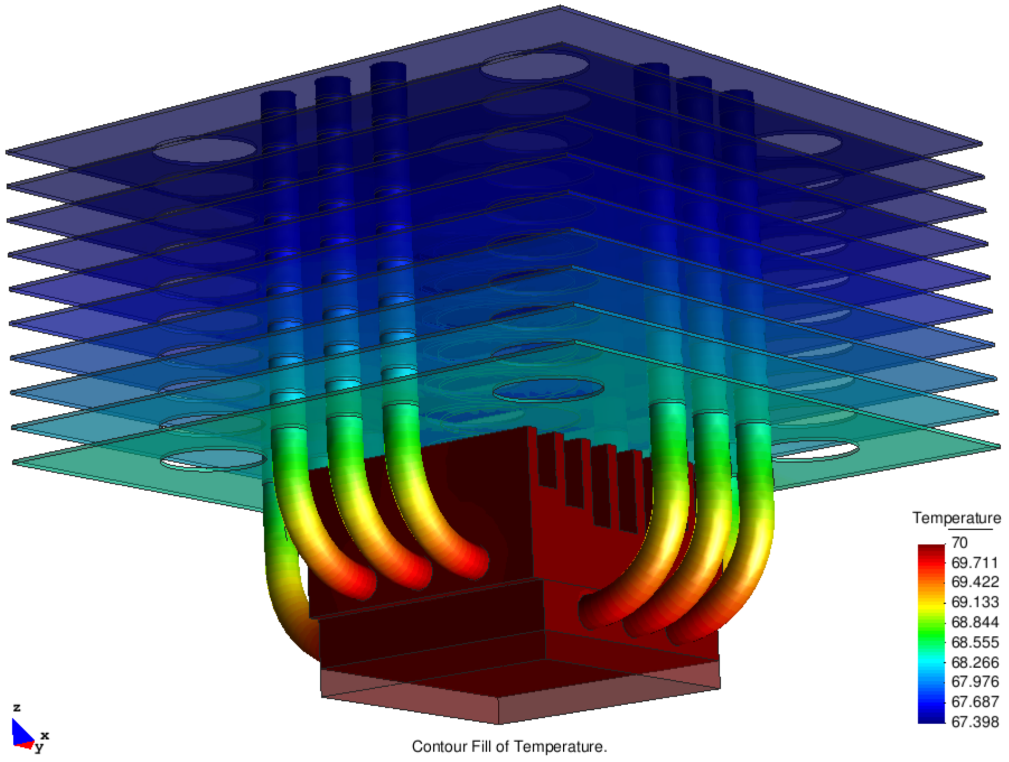
Next step is to work with gross meshes to have improve approximations.

Heat diffusion problems

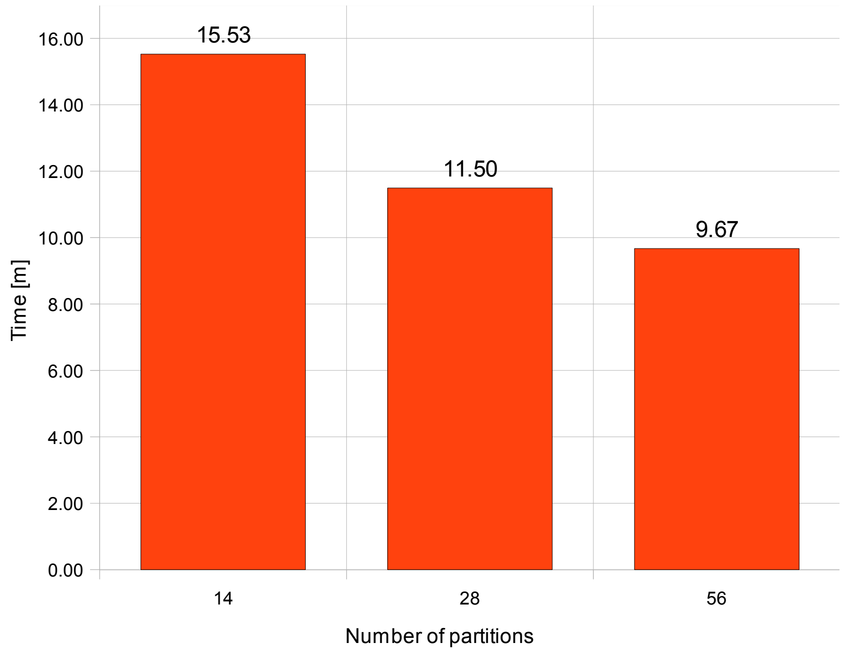
Processor heat sink.

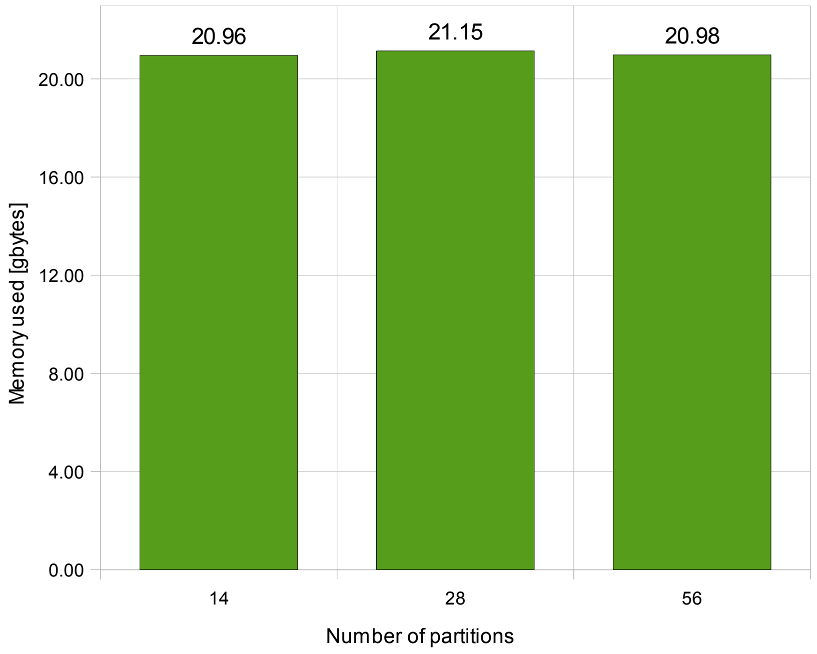


Degrees of freedom	1
Dimension	3
Element type	Tetrahedron
Nodes per element	10
Elements	1’409,407
Nodes	2’267,539
Equations	2’267,539
Solver type	Cholesky



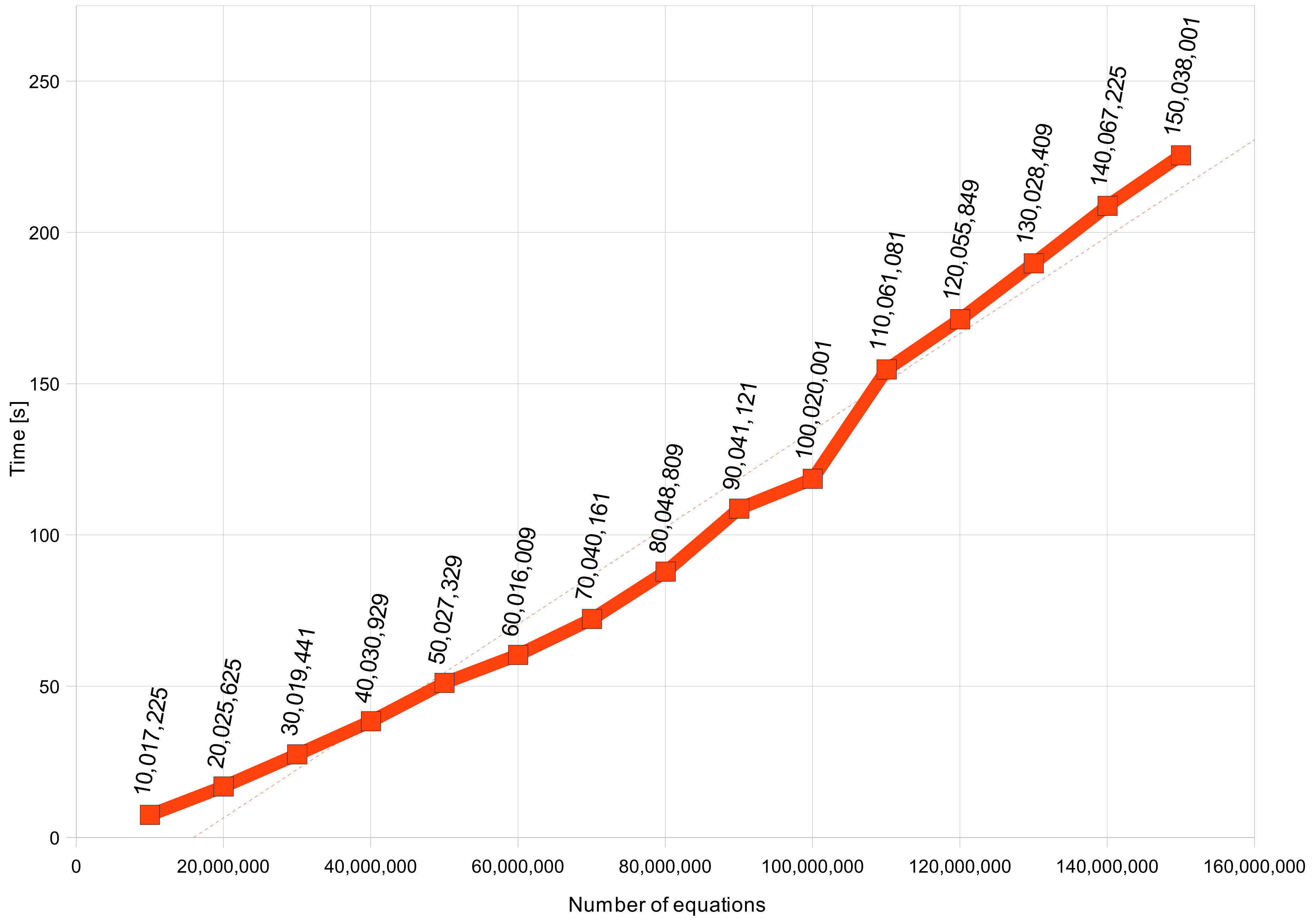
Partitions	Cores per partition	Time [m]	Memory [gbytes]
14	4	15.53	20.96
28	2	11.50	21.15
56	1	9.67	20.98



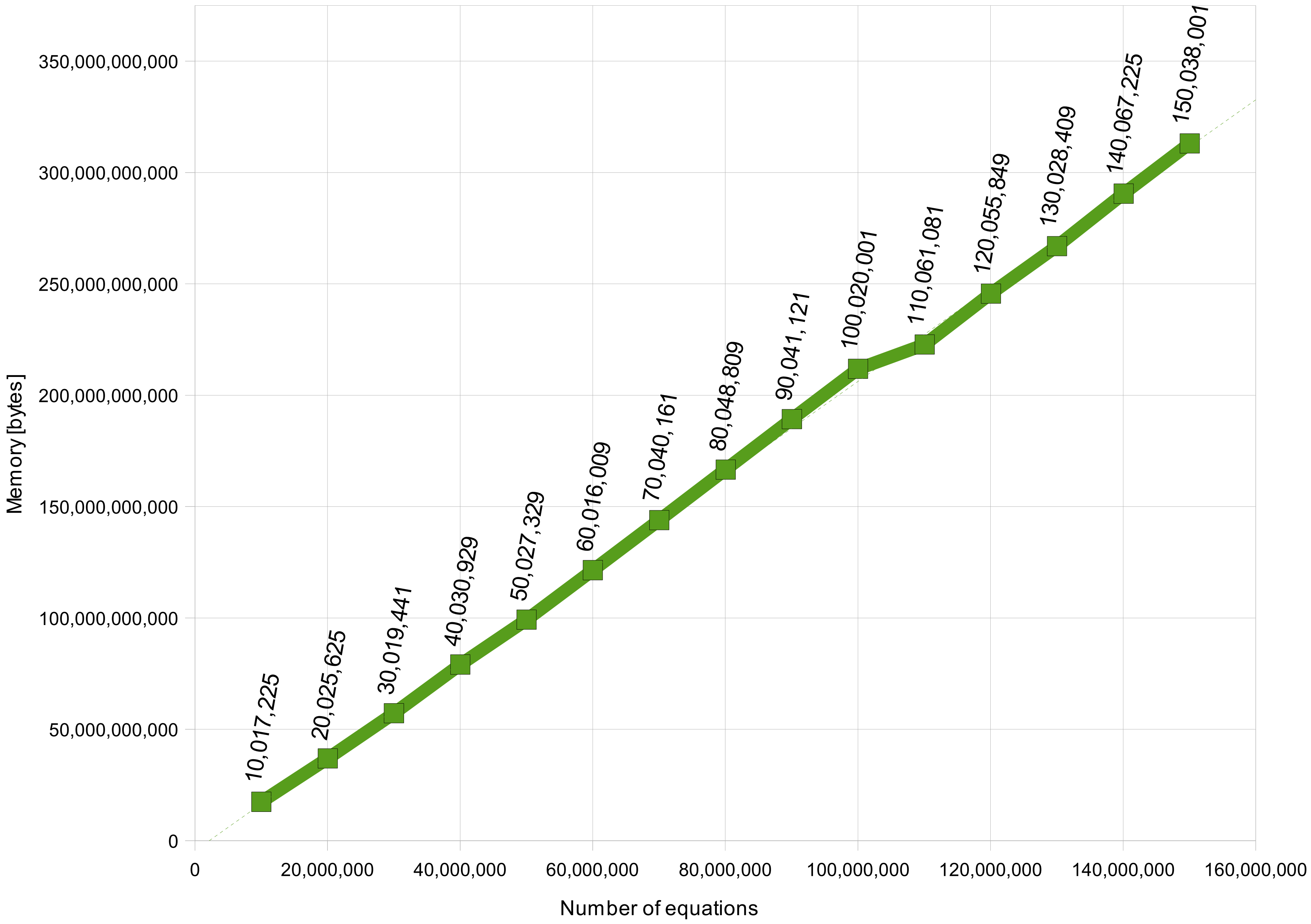


¿Questions?

Number of equations vs time (2D heat diffusion problem)



Number of equations vs memory (2D heat diffusion problem)


Memory [bytes]
Number of equations