SOLUTION OF FINITE ELEMENT PROBLEMS USING HYBRID PARALLELIZATION WITH MPI AND OPENMP

Miguel Vargas-Félix*, Salvador Botello-Rionda

Computer Science Department,
Centre for Mathematical Research (CIMAT)
Callejón Jalisco s/n, Mineral de Valenciana, Guanajuato, Gto. México 36240.
e-mail: miguelvargas, botello@cimat.mx, web: http://www.cimat.mx

SUMMARY

The finite element method is used to solve problems like solid deformation and heat diffusion in domains with complex geometries. This kind of geometries requires discretization with millions of elements, this is equivalent to solve systems of equations with millions of variables. The aim is to use computer clusters to solve this systems.

The solution method used is Schur substructuration. Using it is possible to divide a large system of equations into many small ones to solve them more efficiently. This method allows parallelization.

MPI (Message Passing Interface) is used to distribute the systems of equations to solve each one in a computer of a cluster. Each system of equations is solved using a solver implemented with OpenMP. The systems of equations are sparse.

PROBLEM DESCRIPTION

SOLID DEFORMATION

We want to calculate linear inner displacements of a solid resulting from forces or displacements imposed on its boundaries.

The displacement vector inside the domain is defined as

$$\mathbf{u}(x,y,z)=\begin{pmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{pmatrix},$$

the strain vector $\boldsymbol{\varepsilon}$ is

$$\boldsymbol{\varepsilon}=\begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_y \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix}=\begin{pmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix}=\mathbf{E}\,\mathbf{u}.$$

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

We define a differential operator $\mathbf{E}$.

Stress vector is defined as

$$\boldsymbol{\sigma}=(\sigma_x,\sigma_y,\sigma_z,\tau_{xy},\tau_{yz},\tau_{zx})^{\mathrm{T}},$$

where σ_x, σ_y and σ_z are normal stresses; τ_x, τ_x and τ_x are tangential stresses.

Stress an strain are related by

$$\boldsymbol{\sigma}=\mathbf{D}\boldsymbol{\varepsilon};\tag{1}$$

$\boldsymbol{D}$ is called the constitutive matrix, it depends on Young moduli and Poisson coefficients characteristic of media.

Solution is found using the finite element method with the Galerkin weighted residuals. This means that we solve the integral problem in each element using a weak formulation. The integral expression of equilibrium in elasticity problems can be obtained using the principle of virtual work [Zien05 pp65-71].,

$$\int_V \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\boldsymbol{\sigma}\,dV=\int_V \delta\boldsymbol{u}^{\mathrm{T}}\boldsymbol{b}\,dV+\oint_A \delta\boldsymbol{u}^{\mathrm{T}}\boldsymbol{t}\,dA+\sum_i \delta\boldsymbol{u}_i^{\mathrm{T}}\boldsymbol{q}_i,\tag{2}$$

here $\boldsymbol{b}$, $\boldsymbol{t}$ and $\boldsymbol{q}$ are the vectors of mass, boundary and punctual forces respectively. The weight functions for weak formulation are chosen to be the interpolation functions of the element, these are N_i, $i=1,\ldots,M$. M is the number of nodes of the element, $\boldsymbol{u}_i$ is the coordinate of the ith node, we have that

$$\boldsymbol{u}=\sum_{i=1}^{M} N_i\boldsymbol{u}_i.\tag{3}$$

Using (3), we can rewrite (1) as:

$$\boldsymbol{\varepsilon}=\sum_{i=1}^{M}\mathbf{E}N_i\boldsymbol{u}_i,$$

or in a more compact form

$$\boldsymbol{\varepsilon}=\begin{pmatrix}\mathbf{E}N_1 & \mathbf{E}N_2 & \cdots & \mathbf{E}N_M\end{pmatrix}\begin{pmatrix}\boldsymbol{u}_1\\ \boldsymbol{u}_2\\ \vdots\\ \boldsymbol{u}_M\end{pmatrix}=\boldsymbol{B}\boldsymbol{u}.$$

Now we can express (6) as $\boldsymbol{\sigma}=\boldsymbol{D}\boldsymbol{B}\boldsymbol{u}$, and then (2) by

$$\underbrace{\int_{V^{\mathrm{e}}}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{B}\,dV^{\mathrm{e}}}_{\boldsymbol{K}^{\mathrm{e}}}\boldsymbol{u}=\underbrace{\int_{V^{\mathrm{e}}}\boldsymbol{b}\,dV^{\mathrm{e}}}_{\boldsymbol{f}_b^{\mathrm{e}}}+\underbrace{\oint_{A^{\mathrm{e}}}\boldsymbol{t}\,dA^{\mathrm{e}}}_{\boldsymbol{f}_t^{\mathrm{e}}}+\boldsymbol{q}^{\mathrm{e}}.\tag{4}$$

By integrating (4) we obtain a system of equations for each element,

$$\boldsymbol{K}^{\mathrm{e}}\boldsymbol{u}^{\mathrm{e}}=\boldsymbol{f}_b^{\mathrm{e}}+\boldsymbol{f}_t^{\mathrm{e}}+\boldsymbol{q}^{\mathrm{e}}.$$

All systems of equations are assembled in a global system of equations,

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

$\boldsymbol{K}$ is called the stiffness matrix, if enough boundary conditions are applied, it will be symmetric positive definite (SPD). By construction it is sparse with storage requirements of order $O(n)$, where n is the total number of nodes in the domain. By solving this system we will obtain the displacements of all nodes in the domain.

HEAT DIFFUSION

The other problem that we want to solve is the stationary case of the heat diffusion, it is modeled using the Poisson equation,

$$\frac{\partial}{\partial x}\left(k\frac{\partial\varphi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial\varphi}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\frac{\partial\varphi}{\partial z}\right)=S(x,y,z), \tag{5}$$

where $\varphi(x,y,z)$ is the unknown temperature distributed on the domain. Lets define the flux vector

$$\boldsymbol{q}=k\begin{pmatrix}\frac{\partial\varphi}{\partial x}\\ \frac{\partial\varphi}{\partial y}\\ \frac{\partial\varphi}{\partial z}\end{pmatrix}.$$

Boundary conditions could be Dirichlet

$$\varphi(x,y)=\bar{\varphi}(x,y) \text{ en } \Gamma_{\varphi},$$

or Neumann

$$\boldsymbol{q}(x,y)=\bar{\boldsymbol{q}}(x,y) \text{ en } \Gamma_{\varphi}.$$

In complex domains is complicated to obtain a solution $\varphi(x,y)$ that satisfies (5). We will look for an approximate solution Φ that satisfies

$$f(\varphi(x,y))=0,$$

in the sense of a weighted integral, like

$$\int_{\Omega} W f(\varphi(x,y))\,dx\,dy=0,$$

where $W=W(x,y)$ is a weighting function.
Reformulating the problem as a weighted integral

$$\int_{\Omega} W\left[\frac{\partial}{\partial x}\left(k\frac{\partial\varphi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial\varphi}{\partial y}\right)-S\right]d\Omega+\underbrace{W[\varphi-\bar{\varphi}]+W[q-\bar{q}]}_{\text{Boundary conditions}}=0.$$

Integrating by parts

$$Wk\frac{\partial\varphi}{\partial x}\bigg|_{\omega}-\int_{\Omega}\frac{\partial W}{\partial x}k\frac{\partial\varphi}{\partial x}d\Omega+Wk\frac{\partial\varphi}{\partial y}\bigg|_{\omega}-\int_{\Omega}\frac{\partial W}{\partial y}k\frac{\partial\varphi}{\partial y}d\Omega-\int_{\Omega}WS\,d\Omega+W[\varphi-\bar{\varphi}]+W[q-\bar{q}]=0,$$

using the definition of flux vector

$$Wq_x\big|_{\omega}-\int_{\Omega}\frac{\partial W}{\partial x}k\frac{\partial\varphi}{\partial x}d\Omega+Wq_y\big|_{\omega}-\int_{\Omega}\frac{\partial W}{\partial y}k\frac{\partial\varphi}{\partial y}d\Omega-\int_{\Omega}WS\,d\Omega+W[\varphi-\bar{\varphi}]+W[q-\bar{q}]=0.$$

There are several ways to select weight functions $W(x,y)$ when element equations are build. We used the Galerkin method, in this one the shape functions are used as weight functions

$$\sum_{i=1}^{3}\left[N_iq_x\big|_{\omega}-\int_{\Omega}\frac{\partial N_i}{\partial x}k\frac{\partial\varphi}{\partial x}d\Omega+N_iq_y\big|_{\omega}-\int_{\Omega}\frac{\partial N_i}{\partial y}k\frac{\partial\varphi}{\partial y}d\Omega-\int_{\Omega}N_iS\,d\Omega+N_i(\varphi-\bar{\varphi})+N_i(q-\bar{q})\right]=0.$$

SCHUR COMPLEMENT METHOD

This is a domain decomposition method with no overlapping [Krui04], the basic idea is to split a large system of equations into smaller systems that can be solved independently in different computers in parallel.

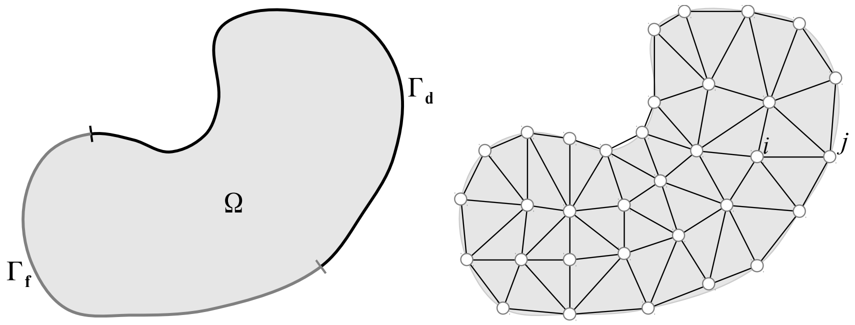


Figure 1. Finite element domain (left), domain discretization (center), partitioning (right).

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

$$\mathbf{K}\,\mathbf{d}=\mathbf{f}, \tag{6}$$

where $\mathbf{K}$ is a symmetric positive definite matrix of size $n\times n$.
If we divide the geometry into p partitions, the idea is to split the workload to let each partition to be handled by a computer in the cluster.

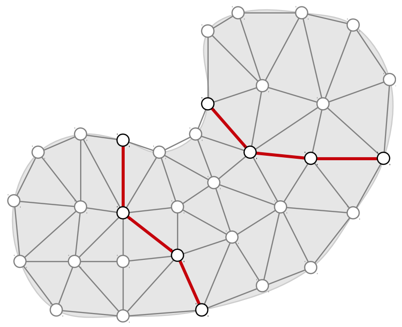


Figure 2. Partitioning example.

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

$$\begin{pmatrix} \mathbf{K}_1^{\mathrm{II}} & & \mathbf{0} & & & \mathbf{K}_1^{\mathrm{IB}} \\ & \mathbf{K}_2^{\mathrm{II}} & & & & \mathbf{K}_2^{\mathrm{IB}} \\ \mathbf{0} & & \mathbf{K}_3^{\mathrm{II}} & & & \mathbf{K}_3^{\mathrm{IB}} \\ \vdots & & & \ddots & & \vdots \\ & & & & \mathbf{K}_p^{\mathrm{II}} & \mathbf{K}_p^{\mathrm{IB}} \\ \mathbf{K}_1^{\mathrm{BI}} & \mathbf{K}_2^{\mathrm{BI}} & \mathbf{K}_3^{\mathrm{BI}} & \cdots & \mathbf{K}_p^{\mathrm{BI}} & \mathbf{K}^{\mathrm{BB}} \end{pmatrix} \begin{pmatrix} \mathbf{d}_1^{\mathrm{I}} \\ \mathbf{d}_2^{\mathrm{I}} \\ \mathbf{d}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{d}_p^{\mathrm{I}} \\ \mathbf{d}^{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1^{\mathrm{I}} \\ \mathbf{f}_2^{\mathrm{I}} \\ \mathbf{f}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{f}_p^{\mathrm{I}} \\ \mathbf{f}^{\mathrm{B}} \end{pmatrix}. \tag{7}$$

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.

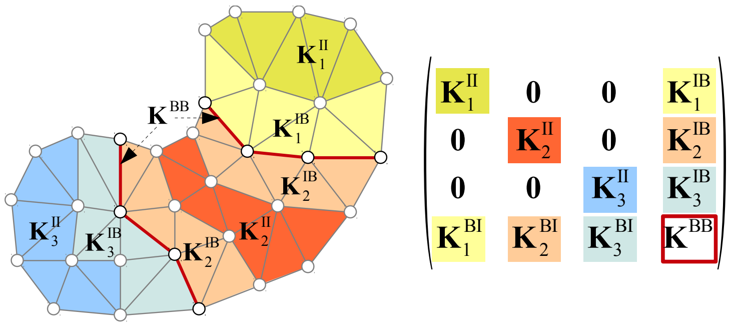

$\mathbf{K}^{\mathrm{BB}}$
$\mathbf{K}_1^{\mathrm{II}}$
$\mathbf{K}_1^{\mathrm{IB}}$
$\mathbf{K}_2^{\mathrm{IB}}$
$\mathbf{K}_2^{\mathrm{II}}$
$\mathbf{K}_3^{\mathrm{II}}$
$\mathbf{K}_3^{\mathrm{IB}}$
$\mathbf{K}_2^{\mathrm{IB}}$
$\begin{pmatrix} \mathbf{K}_1^{\mathrm{II}} & \mathbf{0} & \mathbf{0} &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

Figure 3. Substructuring example with three partitions.

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

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

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

$$\mathbf{d}_i^{\mathrm{I}} = \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \left(\mathbf{f}_i^{\mathrm{I}} - \mathbf{K}_i^{\mathrm{IB}} \mathbf{d}^{\mathrm{B}}\right). \tag{8}$$

After applying Gaussian elimination by blocks on (7), the reduced system of equations becomes

$$\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{K}_i^{\mathrm{IB}}\right) \mathbf{d}^{\mathrm{B}} = \mathbf{f}^{\mathrm{B}} - \sum_{i=1}^{p} \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{f}_i^{\mathrm{I}}. \tag{9}$$

Once the vector $\mathbf{d}^{\mathrm{B}}$ is computed using (9), we can calculate the internal unknowns $\mathbf{d}_i^{\mathrm{I}}$ with (8).

It is not necessary to calculate the inverse in (9). Let's define $\bar{\mathbf{K}}_i^{\mathrm{BB}} = \mathbf{K}_i^{\mathrm{BI}} \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \mathbf{K}_i^{\mathrm{IB}}$, to calculate it [Sori00], we proceed column by column using an extra vector $\mathbf{t}$, and solving for $c=1\ldots n$

$$\mathbf{K}_i^{\mathrm{II}} \mathbf{t} = \left[\mathbf{K}_i^{\mathrm{IB}}\right]_c, \tag{10}$$

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

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

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

$$\mathbf{K}_i^{\mathrm{II}} \mathbf{t} = \mathbf{f}_i^{\mathrm{I}}, \tag{11}$$

and then

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

Each $\bar{\mathbf{K}}_i^{\mathrm{BB}}$ and $\bar{\mathbf{f}}_i^{\mathrm{B}}$ holds the contribution of each partition to (9), this can be written as

$$\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \bar{\mathbf{K}}_i^{\mathrm{BB}}\right) \mathbf{d}^{\mathrm{B}} = \mathbf{f}^{\mathrm{B}} - \sum_{i=1}^{p} \bar{\mathbf{f}}_i^{\mathrm{B}}, \tag{12}$$

once (12) is solved, we can calculate the inner results of each partition using (8).

Since $\mathbf{K}_i^{\mathrm{II}}$ is sparse and has to be solved many times in (10), a efficient way to proceed is to use a Cholesky factorization of $\mathbf{K}_i^{\mathrm{II}}$. To reduce memory usage and increase speed a sparse Cholesky factorization has to be implemented, this method is explained below.

In case of (12), $\mathbf{K}^{\mathrm{BB}}$ is sparse, but $\bar{\mathbf{K}}_i^{\mathrm{BB}}$ are not. To solve this system of equations an sparse version of conjugate gradient was implemented, the matrix $\left(\mathbf{K}^{\mathrm{BB}} - \sum_{i=1}^{p} \bar{\mathbf{K}}_i^{\mathrm{BB}}\right)$ is not assembled, but maintained distributed. In the conjugate gradient method is only important to know how to multiply the matrix by the descent direction, in our implementation each $\bar{\mathbf{K}}_i^{\mathrm{BB}}$ is maintained in their respective computer and the multiplication is done in a distributed way an the resulted vector is formed with contributions from all partitions. To improve the convergence of the conjugate gradient a Jacobi preconditioned is used. This algorithm is described below.

One benefit of this method is that the condition number of the system is reduced when solving (12), this decreases the number of iterations needed to converge.

SPARSE MATRICES

Considering all elements we assemble a system of equations (with certain Dirichlet or Neumann boundary conditions) to solve a linear system of equations

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

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.

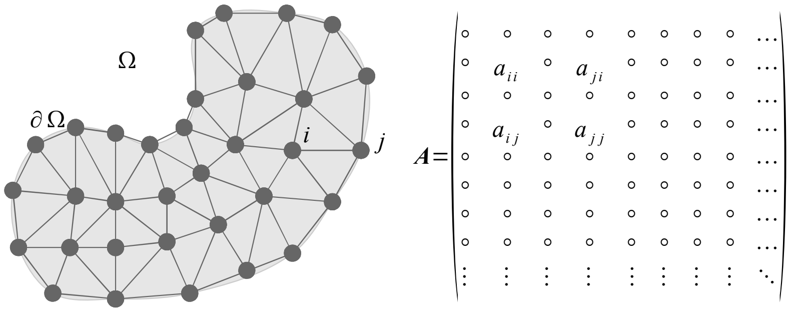


Figure 4. Discretized domain (mesh) and its matrix representation.

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

For example, $\boldsymbol{A}\in\mathbb{R}^{556\times556}$ has 309,136 entries, with $\eta(\boldsymbol{A})=1810$, this means that only the 0.58% of the entries are non zero.

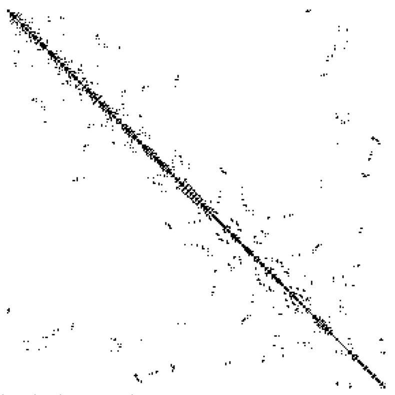


Figure 5. Black dots indicates a non zero entry in the matrix

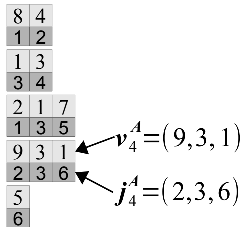
In finite element problems all matrices have symmetric structure, and depending on the problem symmetric values or not.

MATRIX STORAGE

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

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

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



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

If we do not order entries of each row, then to search an entry with certain column index will have a cost of $O(|\boldsymbol{v}_i^A|)$ in the worst case. To improve it we will keep $\boldsymbol{v}_i^A$ and $\boldsymbol{j}_i^A$ ordered by the indexes $\boldsymbol{j}_i^A$. Then we could perform a binary algorithm to have an search cost of $O(\log_2|\boldsymbol{v}_i^A|)$.

The main advantage of using Compressed Row Storage is when data in each row is stored continuously and accessed in a sequential way, this is important because we will have and efficient processor cache usage [Drep07].

CHOLESKY FACTORIZATION FOR SPARSE MATRICES

The cost of using Cholesky factorization $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$ is expensive if we want to solve systems of equations with full matrices, but for sparse matrices we could reduce this cost significantly if we use reordering strategies and we store factor matrices using CRS identifying non zero entries using symbolic factorization. With these strategies we could maintain memory and time requirements near to $O(n)$. Also Cholesky factorization could be implemented in parallel.

Formulae to calculate $\boldsymbol{L}$ entries are

$$L_{ij}=\frac{1}{L_{jj}}\left(A_{ij}-\sum_{k=1}^{j-1}L_{ik}L_{jk}\right),\text{ for } i>j; \tag{13}$$

$$L_{jj}=\sqrt{A_{jj}-\sum_{k=1}^{j-1}L_{jk}^2}. \tag{14}$$

REORDERING ROWS AND COLUMNS

By reordering the rows and columns of a SPD matrix $\boldsymbol{A}$ we could reduce the fill-in (the number of non-zero entries) of $\boldsymbol{L}$. The next images show the non zero entries of $\boldsymbol{A}\in\mathbb{R}^{556\times 556}$ and the non zero entries of its Cholesky factorization $\boldsymbol{L}$.

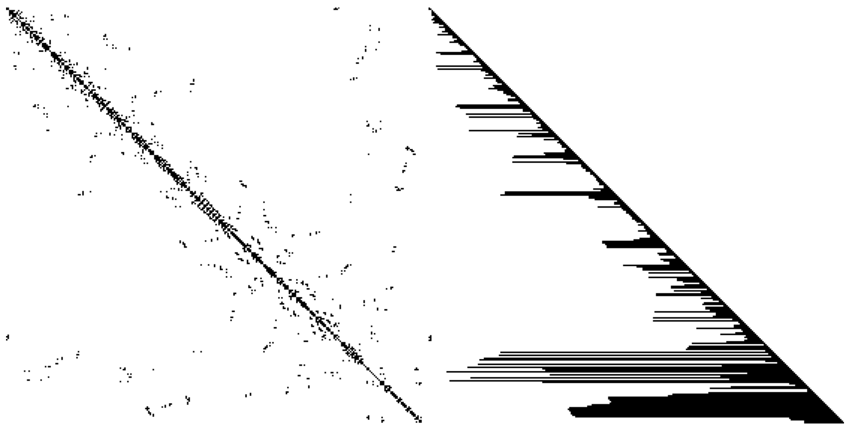


Figure 6. Left: non-zero entries of $\boldsymbol{A}$. Right: non-zero entries of $\mathbf{L}$ (Cholesky factorization of $\boldsymbol{A}$)

The number of non zero entries of $\boldsymbol{A}$ is $\eta(\boldsymbol{A})=1810$, and for $\boldsymbol{L}$ is $\eta(\boldsymbol{L})=8729$. The next images show $\boldsymbol{A}$ with an efficient reordering by rows and columns.

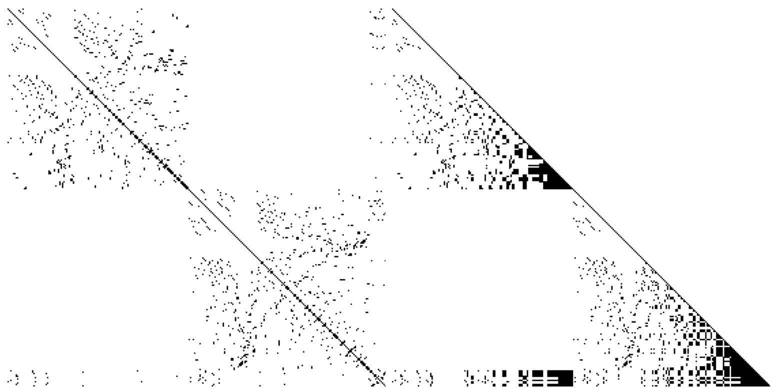


Figure 7. Left: non-zero entries of reordered $\boldsymbol{A}$. Right: non-zero entries of $\boldsymbol{L}$.

By reordering we have a new factorization with $\eta(\boldsymbol{L})=3215$, reducing the fill-in to 0.368 of the size of the not reordered version. Both factorizations allow us to solve the same system of equations.

The most common reordering heuristic to reduce fill-in is the minimum degree algorithm, the basic version is presented in [Geor81 p116]:

```
Let be a matrix $\boldsymbol{A}$ and its corresponding graph $G_0$
$i \leftarrow 1$
repeat
    Let node $x_i$ in $G_{i-1}(X_{i-1}, E_{i-1})$ have minimum degree
    Form a new elimination graph $G_i(X_i, E_i)$ as follow:
        Eliminate $x_i$ and its edges from $G_{i-1}$
        Add edges make $\mathrm{adj}(x_1)$ adjacent pairs in $G_i$
    $i \leftarrow i+1$
while $i < |X|$
```

More advanced versions of this algorithm can be consulted in [Geor89].

There are more complex algorithms that perform better in terms of time and memory requirements, the nested dissection algorithm developed by Karypis and Kumar [Kary99] included in METIS library gives very good results.

Symbolic Cholesky factorization

This algorithm identifies non zero entries of $\boldsymbol{L}$, a deep explanation could be found in [Gall90 p86-88].

For an sparse matrix $\boldsymbol{A}$, we define

$$\boldsymbol{a}_j \overset{\text{def}}{=} \{k > j \mid A_{kj} \neq 0\},\ j = 1 \dots n,$$

as the set of non zero entries of column j of the strictly lower triangular part of $\boldsymbol{A}$.

In similar way, for matrix $\boldsymbol{L}$ we define the set

$$\boldsymbol{l}_j \overset{\text{def}}{=} \{k > j \mid L_{kj} \neq 0\},\ j = 1 \dots n.$$

We also use sets define sets $\boldsymbol{r}_j$ that will contain columns of $\boldsymbol{L}$ which structure will affect the column j of $\boldsymbol{L}$. The algorithm is:

```
$\boldsymbol{r}_j \leftarrow \emptyset,\ j \leftarrow 1 \dots n$
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
```

For the next example matrix column 2, $\boldsymbol{a}_2$ and $\boldsymbol{l}_2$ will be:

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

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

Figure 8. Example matrix, showing how $\boldsymbol{a}_2$ and $\boldsymbol{l}_2$ are formed.

This algorithm is very efficient, complexity in time and memory usage has an order of $O(\eta(\boldsymbol{L}))$. Symbolic factorization could be seen as a sequence of elimination graphs [Geor81 pp92-100].

FILLING ENTRIES IN PARALLEL

Once non zero entries are determined we can rewrite (13) and (14) as

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

$$L_{jj} = \sqrt{A_{jj} - \sum_{\substack{k \in \boldsymbol{j}_j^L \\ k < j}} L_{jk}^2}.$$

The resulting algorithm to fill non zero entries is [Varg10]:

```
for $j \leftarrow 1 \ldots n$
    $L_{jj} \leftarrow A_{jj}$
    for $q \leftarrow 1 \ldots |\boldsymbol{v}_j^{\boldsymbol{L}}|$
        $L_{jj} \leftarrow L_{jj} - (\boldsymbol{v}_j^{\boldsymbol{L}})_q (\boldsymbol{v}_j^{\boldsymbol{L}})_q$
    $L_{jj} \leftarrow \sqrt{L_{jj}}$
    $L_{jj}^{\mathrm{T}} \leftarrow L_{jj}$
    parallel for $q \leftarrow 1 \ldots \left|\boldsymbol{j}_j^{\boldsymbol{L}^{\mathrm{T}}}\right|$
        $i \leftarrow \left(\boldsymbol{j}_j^{\boldsymbol{L}^{\mathrm{T}}}\right)_q$
        $L_{ij} \leftarrow A_{ij}$
        $r \leftarrow 1;\ \rho \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_r$
        $s \leftarrow 1;\ \sigma \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_s$
        repeat
            while $\rho < \sigma$
                $r \leftarrow r+1;\ \rho \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_r$
            while $\rho > \sigma$
                $s \leftarrow s+1;\ \sigma \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_s$
            while $\rho = \sigma$
                if $\rho = j$
                    exit repeat
                $L_{ij} \leftarrow L_{ij} - \left(\boldsymbol{v}_i^{\boldsymbol{L}}\right)_r \left(\boldsymbol{v}_j^{\boldsymbol{L}}\right)_s$
                $r \leftarrow r+1;\ \rho \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_r$
                $s \leftarrow s+1;\ \sigma \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_s$
        $L_{ij} \leftarrow \frac{L_{ij}}{L_{jj}}$
        $L_{ji}^{\mathrm{T}} \leftarrow L_{ij}$
```

This algorithm could be parallelized if we fill column by column. Entries of each column can be calculated in parallel with OpenMP, because there are no dependence among them [Heat91 pp442-445]. Calculus of each column is divided among cores.

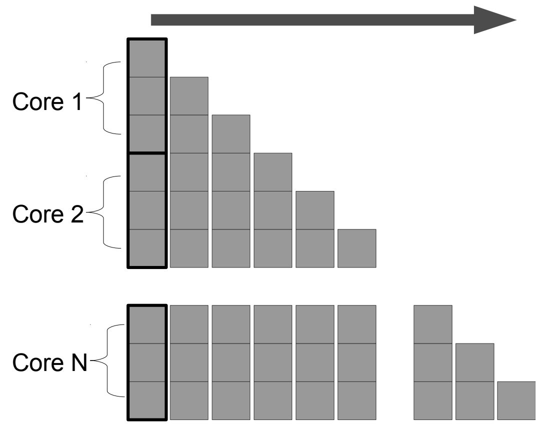


Figure 9. Calculation order to parallelize the Cholesky factorization.

Cholesky solver is particularly efficient because the stiffness matrix is factorized once. The domain is partitioned in many small sub-domains to have small and fast Cholesky factorizations. The parallelization was made using the OpenMP schema.

PARALLEL PRECONDITIONED CONJUGATE GRADIENT

Conjugate gradient (CG) is a natural choice to solve systems of equations with SPD matrices, we will discuss some strategies to improve convergence rate and make it suitable to solve large sparse systems using parallelization.

The condition number κ of a non singular matrix $\boldsymbol{A} \in \mathbb{R}^{m \times m}$, given a norm $\|\cdot\|$ is defined as

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|.$$

For the norm $\|\cdot\|_2$,

$$\kappa_2(\boldsymbol{A})=\|\boldsymbol{A}\|_2\cdot\|\boldsymbol{A}^{-1}\|_2=\frac{\sigma_{max}(\boldsymbol{A})}{\sigma_{min}(\boldsymbol{A})},$$

where σ is a singular value of $\boldsymbol{A}$.

For a SPD matrix,

$$\kappa(\boldsymbol{A})=\frac{\lambda_{max}(\boldsymbol{A})}{\lambda_{min}(\boldsymbol{A})},$$

where λ is an eigenvalue of $\boldsymbol{A}$.

A system of equations $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ is bad conditioned if a small change in the values of $\boldsymbol{A}$ or $\boldsymbol{b}$ results in a large change in $\boldsymbol{x}$. In well conditioned systems a small change of $\boldsymbol{A}$ or $\boldsymbol{b}$ produces an small change in $\boldsymbol{x}$. Matrices with a condition number near to 1 are well conditioned.

A preconditioner for a matrix $\boldsymbol{A}$ is another matrix $\boldsymbol{M}$ such that $\boldsymbol{M}\,\boldsymbol{A}$ has a lower condition number

$$\kappa(\boldsymbol{M}\,\boldsymbol{A})<\kappa(\boldsymbol{A}).$$

In iterative stationary methods (like Gauss-Seidel) and more general methods of Krylov subspace (like conjugate gradient) a preconditioner reduces the condition number and also the amount of steps necessary for the algorithm to converge.

Instead of solving

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

with preconditioning we solve

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

The preconditioned conjugate gradient algorithm is:

```
$x_0$, initial approximation
$r_0 \leftarrow b - A x_0$, initial gradient
$q_0 \leftarrow M r_0$
$p_0 \leftarrow q_0$, initial descent direction
$k \leftarrow 0$
while $\|r_k\| > \varepsilon$
    $\alpha_k \leftarrow -\frac{r_k^T q_k}{p_k^T A p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
    $q_{k+1} \leftarrow M r_{k+1}$
    $\beta_k \leftarrow \frac{r_{k+1}^T q_{k+1}}{r_k^T q_k}$
    $p_{k+1} \leftarrow q_{k+1} + \beta_k p_k$
    $k \leftarrow k+1$
```

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

We used the Jacobi preconditioner, it is suitable for sparse systems with SPD matrices. The diagonal part of $\boldsymbol{M}^{-1}$ is stored as a vector,

$$\boldsymbol{M}^{-1}=\left(\operatorname{diag}\left(\boldsymbol{A}\right)\right)^{-1}.$$

Parallelization of this algorithm is straightforward, because the calculus of each entry of $\boldsymbol{q}_k$ is independent.

Parallelization of the preconditioned CG is done using OpenMP, operations parallelized are matrix-vector, dot products and vector sums. To synchronize threads has a computational cost, it is possible to modify to CG to reduce this costs maintaining numerical stability [DAze93].

COMPUTER CLUSTERS AND MPI

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.

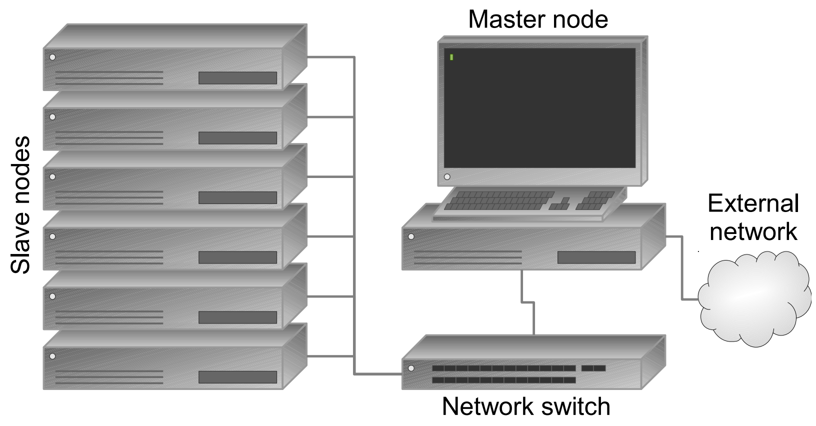


Figure 10. Diagram of a Beowulf cluster of computers.

In our software implementation each partition is assigned to one process. To parallelize the program and move data among nodes we used the Message Passing Interface (MPI) schema [MPIF08], it contains set of tools that makes easy to start several instances of a program (processes) and run them in parallel. Also, MPI has several libraries with a rich set of routines to send and receive data messages among processes in an efficient way. MPI can be configured to execute one or several processes per node.

For partitioning the mesh we used the METIS library [Kary99].

PARALLELIZATION USING MULTI-CORE COMPUTERS

Using domain decomposition with MPI we could have a partition assigned to each node of a cluster, we can solve all partitions concurrently. If each node is a multi-core computer we can also parallelize the solution of the system of equations of each partition. To implement this parallelization we use the OpenMP model.

This parallelization model consists in compiler directives inserted in the source code to parallelize sections of code. All cores have access to the same memory, this model is known as shared memory schema.

In modern computers with shared memory architecture the processor is a lot faster than the memory [Wulf95].

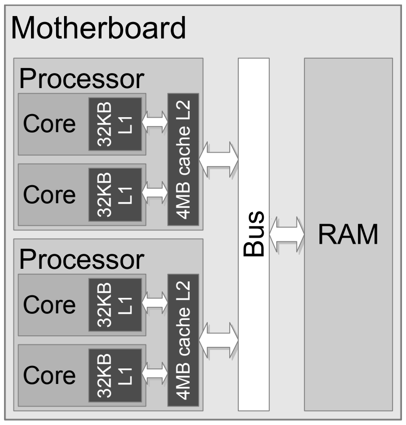


Figure 11. Schematic of a multi-processor and multi-core computer.

To overcome this, a high speed memory called cache exists between the processor and RAM. This cache reads blocks of data from RAM meanwhile the processor is busy, using an heuristic to predict what the program will require to read next. Modern processor have several caches that are organized by levels (L1, L2, etc), L1 cache is next to the core. It is important to considerate the cache when programming high performance applications, the next table indicates the number of clock cycles needed to access each kind of memory by a Pentium M processor:

Access to	CPU cycles
CPU registers	<=1
L1 cache	3
L2 cache	14
RAM	240

A big bottleneck in multi-core systems with shared memory is that only one core can access the RAM at the same time.

Another bottleneck is the cache consistency. If two or more cores are accessing the same RAM data then different copies of this data could exists in each core's cache, if a core modifies its cache copy then the system will need to update all caches and RAM, to keep consistency is complex and expensive [Drep07]. Also, it is necessary to consider that cache circuits are designed to be more efficient when reading continuous memory data in an ascendent sequence [Drep07 p15].

To avoid lose of performance due to wait for RAM access and synchronization times due to cache inconsistency several strategies can be use:

- Work with 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 these strategies.

NUMERICAL EXPERIMENTS

We are going to present just a couple examples, these were executed in a cluster with 15 nodes, each one with two dual core Intel Xeon E5502 (1.87GHz) processors, a total of 60 cores. A node is used as a master process to load the geometry and the problem parameters, partition an split the systems of equations. The other 14 nodes are used to solve the system of equations of each partition. Times are in seconds. Tolerance used is $1\text{x}10^{-10}$.

SOLID DEFORMATION

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.

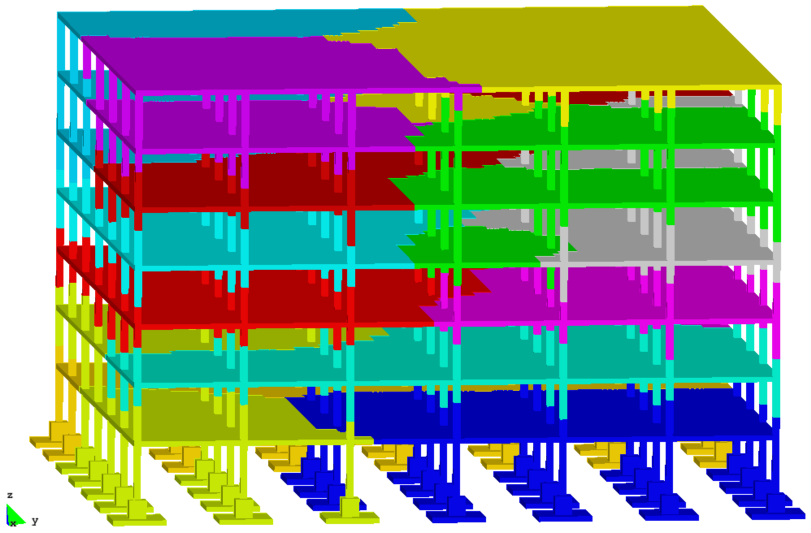
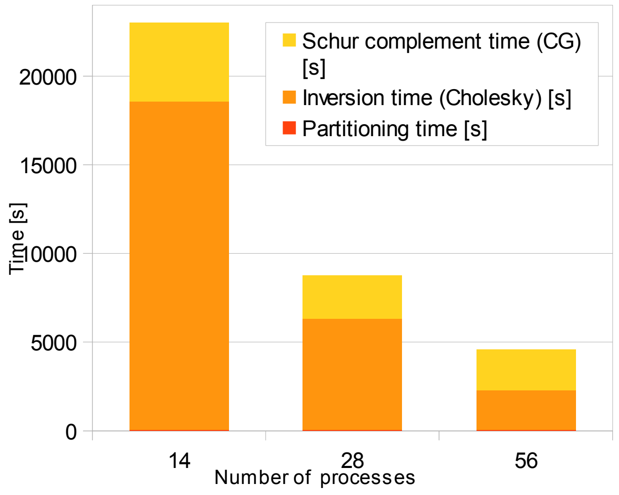
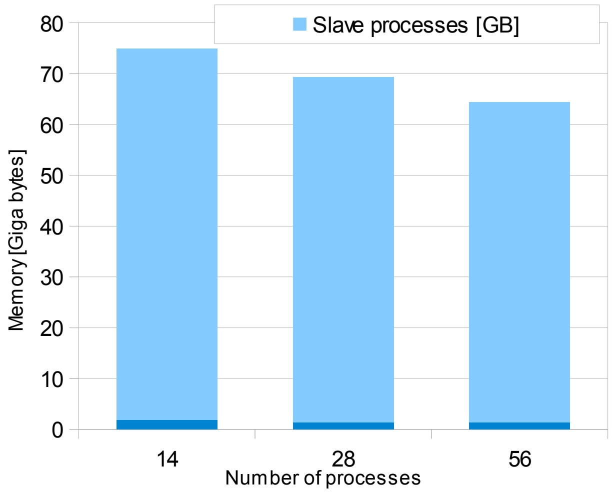


Figure 12. Substructuration of the domain.

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

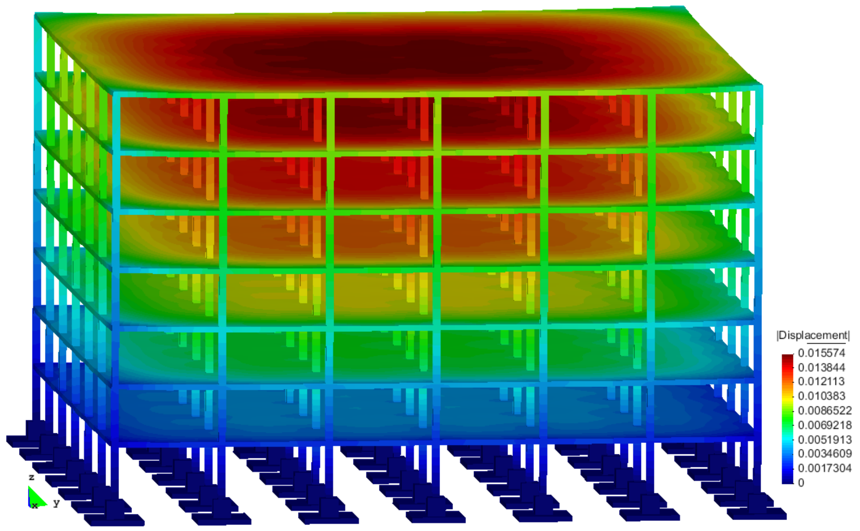


Figure 13. Resulting deformation.

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

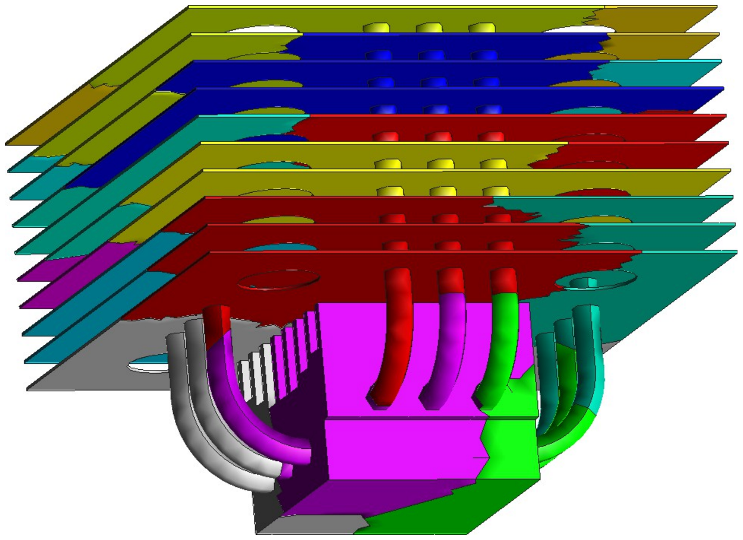
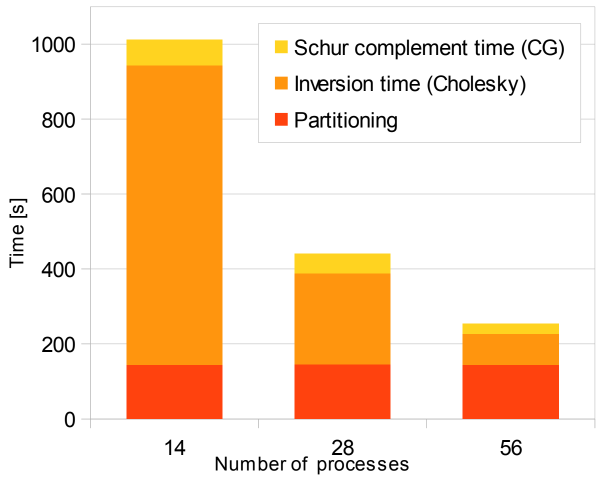
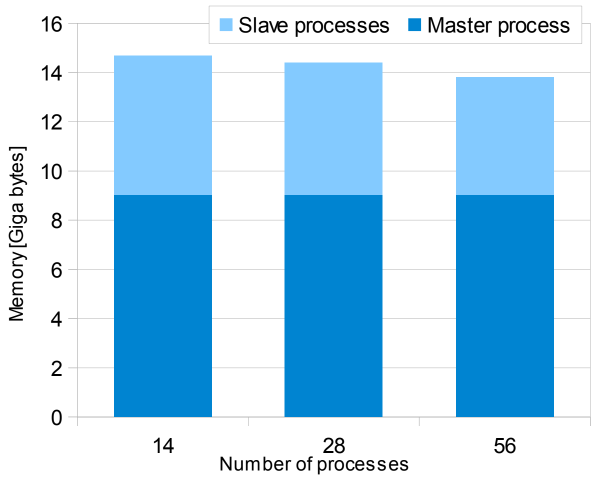


Figure 14. Substructuration of the domain.

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





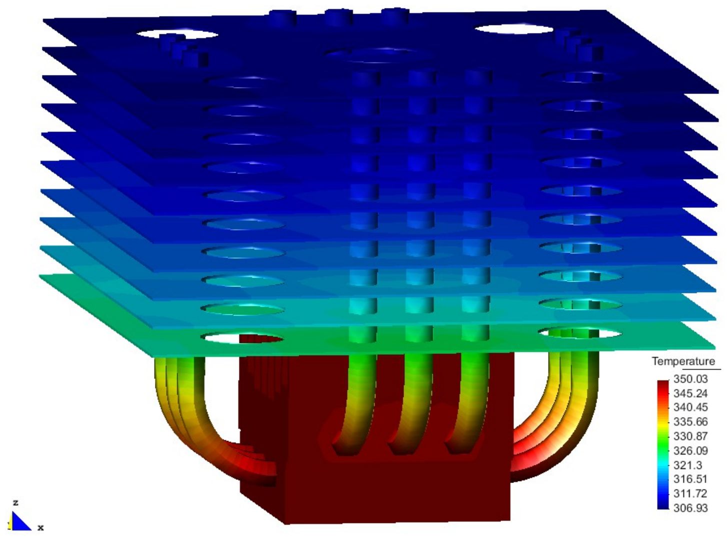


Figure 15. Resulting temperature distribution.

LARGE SYSTEMS OF EQUATIONS

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

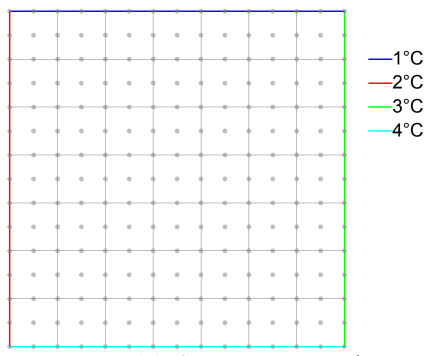
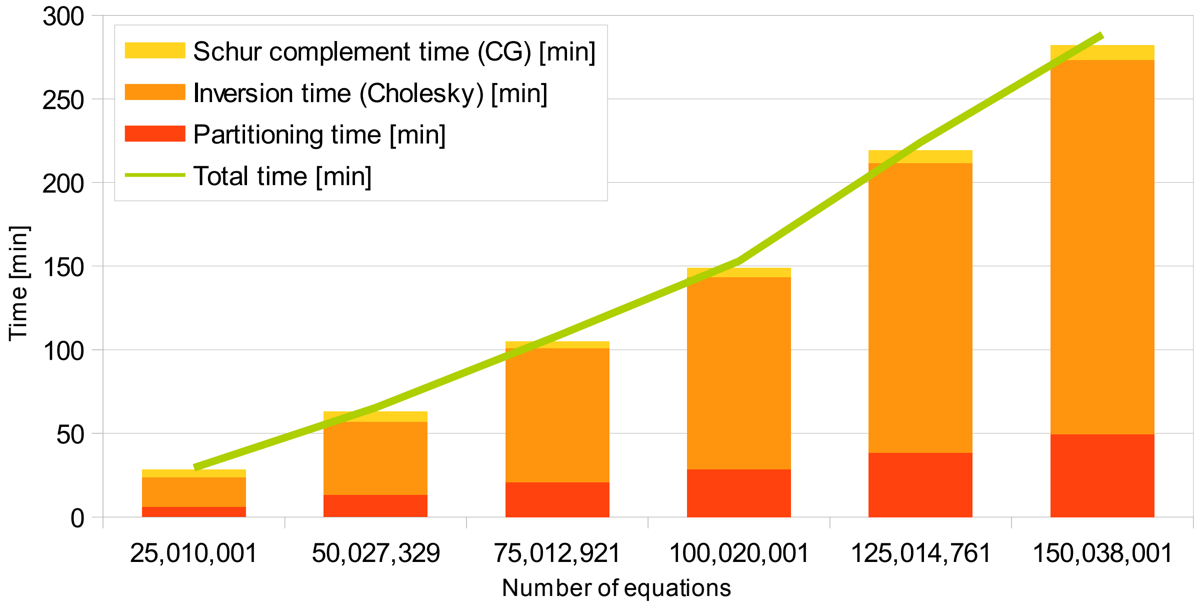


Figure 16. Geometry example.

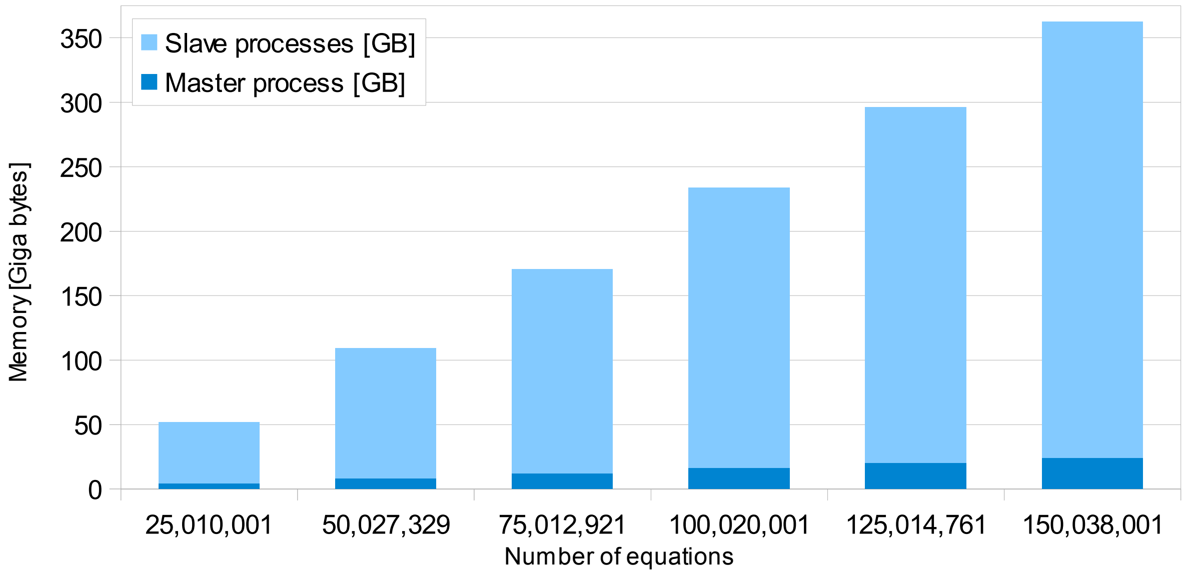
The domain was discretized using quadrilaterals with nine nodes, the discretization made was from 25 million nodes up to 150 million nodes. In all cases we divided the domain into 116 partitions.

In this case we used a larger cluster with mixed equipment 15 nodes with 4 Intel Xeon E5502 cores and 14 nodes with 4 AMD Opteron 2350 cores, a total of 116 cores. A node is used as a master process to load the geometry and the problem parameters, partition an split the systems of equations. Tolerance used was 1x10^{-10}.

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



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



CONCLUSIONS

We presented just a few case studies of the usage of the Schur sub structuring method for complex geometries with large number of degrees of freedom.

It is difficult to measure speed-up when working with complex geometries, the partitioning routines that we used [Kary99] have heuristics that try to divide equally the number of nodes, thus the shape of the partitions for each mesh could vary a lot. Nevertheless results have a linear tendency in reduction of solution times.

In this case we used the Jacobi preconditioner, but there are other preconditioners that lead to better convergence that could be interesting to test, like the family of methods called FETI (finite element tearing and interconnect) [Farh91].

REFERENCES

[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.

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

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

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

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

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

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

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

[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.

[Varg10] J. M. Vargas-Felix, S. Botello-Rionda. "Parallel Direct Solvers for Finite Element Problems". Comunicaciones del CIMAT, I-10-08 (CC), 2010.

[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.