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

Miguel Vargas-Félix, Salvador Botello-Rionda
Computer Science Departament,
Centre for Mathematical Research (CIMAT).
Callejón Jalisco s/n, Mineral de Valenciana, Guanajuato, Gto. México 36240.
e-mail: miguelvargas@cimat.mx, botello@cimat.mx
WWW: http://www.cimat.mx

Table of Contents

Introduction........2
Modules........3
  FEMT library........3
  FEMSolver........5
  FEMSolver.MPI........5
  EqnSolver........6
  MatSolver........6
Sparse matrices........7
  Matrix storage........7
Cholesky factorization for sparse matrices........8
  Reordering rows and columns........8
  Symbolic Cholesky factorization........10
  Filling entries in parallel........11
  LDL’ factorization........12
LU factorization for sparse matrices........12
Solvers for triangular sparse matrices........13
Parallel preconditioned conjugate gradient........14
  Jacobi preconditioner........15
  Incomplete Cholesky factorization preconditioner........15
  Factorized sparse approximate inverse preconditioner........16
Parallel biconjugated gradient........17
  Preconditioning with incomplete LU factorization........18
  Approximate inverse preconditioner........19
Schur sustructuring method........20
  Partitioning........20
  Schur complement method........21
Parallelization........22
  Parallelization using multi-core computers........22

Computer clusters and MPI....24
Numerical experiments....24
Solutions with OpenMP....24
Solutions with domain decomposition using MPI....27
Large systems of equations....30
Library license....32
References....32

Introduction

We present a new open source library and tools for solving large sparse linear systems of equations. This software is specialy set to solve systems of equations resulting from finite element, finite volume and finite differences discretizations.

In this work we will describe the solvers included in the library, there are three kind of solvers: direct, iterative and domain decomposition. Direct and iterative solvers are designed to run in parallel in multi-core computers using OpenMP. The domain decomposition solver has been designed to run in clusters of computers using a combination of MPI (Message Passing Interface) and OpenMP.

Direct solvers implemented are sparse versions of Cholesky and LU factorizations. We use reordering and symbolic factorization to reduce and determine fill-in.

Iterative solvers are conjugate gradient and biconjugate gradient for sparse matrices. To improve convergence, we implemented several preconditioners suitable for sparse systems: Jacobi, incomplete Cholesky and LU factorizations, and sparse approximate inverses.

The domain decomposition method included is Schur substructuring method. This method allows to split a large system of equations into many smaller systems that can be solved in different processors or computers of a cluster, reducing solution times and making possible to solve systems that can not fit in a single computer.

The FEMSolver is multi-platform library (Windows, GNU/Linux, Mac OS), released under the GNU Library General Public License. It has been programmed in modern standard C++, and has modules to access it from other languajes like Fortran, Python, C, etc.

We will show some numerical results of finite element modelation of solid deformation and heat difussion, with systems of equations that have from a few million, to more than one hundred million degrees of freedom.

Modules

FEMT library

This is a library designed to help scientists and engineers to solve large systems of equations from problems of finite element, finite volume and finite difference problems. It was programmed in modern C++, with focus on performance and efficient resource usage. It has a simple and intuitive set of classes to handle meshes, systems of equations and solvers. Below is a list of its capabilities.

Containers

The following containers are available, these have been implemented to be efficient in time access and memory consumption, most of the code use templates to allow these containers to handle any data type, like float, double, integer, bool, etc.

- Sparse matrices (compress row and compress column storage), sparse vectors.
- Dense matrices and vectors.
- Upper and lower triangular matrices.
- Sets.
- Lists.

File operations

FEMT library includes routines to read/write files with several common formats:

- CSV (comma separated values).
- Matrix Market (http://math.nist.gov/MatrixMarket).
- MatLab (format v4).

Finite element routines

A group of classes to handle finite element data:

- Meshes with several kind of elements (triangular, quadrilateral, tetrahedron, hexahedron).
- Nodes for 2D and 3D geometries.
- Shape functions for, triangles (3 and 6 nodes), quadrilaterals (4, 8 and 9 nodes), tetrahedra (4 and 10 nodes), hexahedra (8, 20 and 27 nodes).
- Jacobian and determinant calculation for all shape functions.
- Quadrature rules, gauss quadrature (1, 2, 3, 4, 5, 6 and 7 points), triangle (1, 3, 4 and 7 points), tetrahedron (1, 4, 5 and 11 points).
- Facet integration.
- Mesh partitioning using METIS library (http://glaros.dtc.umn.edu/gkhome/views/metis).
- Graph reordering for reducing fill-in in factorizations.

Parser

Parser for mathematical expresions. It can evaluate in run time strings.

- Suports single and double precision.

- Unlimited number of variables.
- Supports the following functions: abs, acos, asin, atan, atan2, ceil, cos, cosh, tan, exp, floor, log, log10, mod, pow, sin, sinh, sqrt, tan, tanh.

Solvers for sparse matrices with OpenMP

The following list enumerates all solvers implemented for sparse matrices, all have been desigened to run in parallel in multi-core computers using the OpenMP schema:

- Cholesky.
- Cholesky (LDL').
- LU (Doolittle version).
- Upper triangular matrices.
- Lower triangular matrices.
- Conjugate gradient.
- Conjugate gradient + Jacobi preconditioner.
- Conjugate gradient + incomplete Cholesky factorization preconditioner.
- Conjugate gradient + incomplete Cholesky LDL' factorization preconditioner.
- Conjugate gradient + factorized sparse approximate inverse preconditioner.
- Biconjugate gradient.
- Biconjugate gradient + Jacobi preconditioner.
- Biconjugate gradient + incomplete LU factorization preconditioner.
- Conjugate gradient + minimal residual sparse approximate inverse preconditioner.

Solvers for sparse matrices with MPI

These solvers have been designed to run in clusters of computers and solve systems of equations that can not fit in a single computer.

- Distributed conjugate gradient with Jacobi preconditioner. This version of the conjugate gradient stores the matrix in several computers that work simultaneously to obtain the solution.
- Schur substructuring method. This is a domain decomposition method that starts partitioning the domain, splitting a large system of equations into many small systems, each one is sent to a computer of the cluster to be evaluated independently, a global solution is obained from the comtribution of all systems.

Solvers for dense matrices with OpenMP

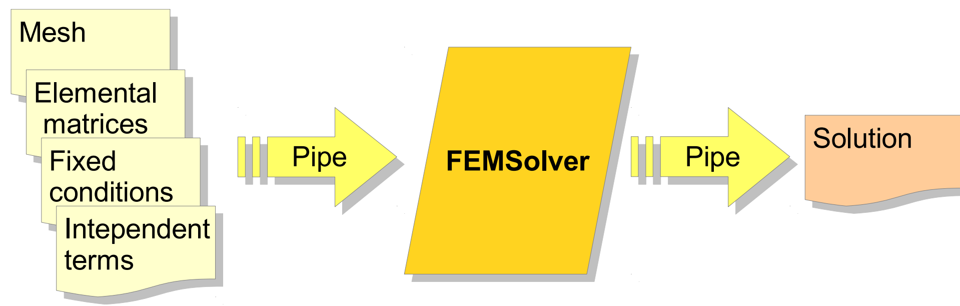
There are also some solvers implemented for dense matrices that run in parallel on multi-core computers.

- Cholesky.
- LU (Doolittle version).
- Upper triangular matrices.
- Lower triangular matrices.
- Conjugate gradient.

- Conjugate gradient + Jacobi preconditioner.

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.



If you know how to write files, you can use FEMSolver. The user only needs to write on a pipe data describing the mesh, all the elemental matrices, fixed conditions, and the vector of intependent terms. FEMSolver reads these data and runs one of the solvers from the FEMT library and retuns the solution using other pipe, the user only has to read it like reading a normal file.

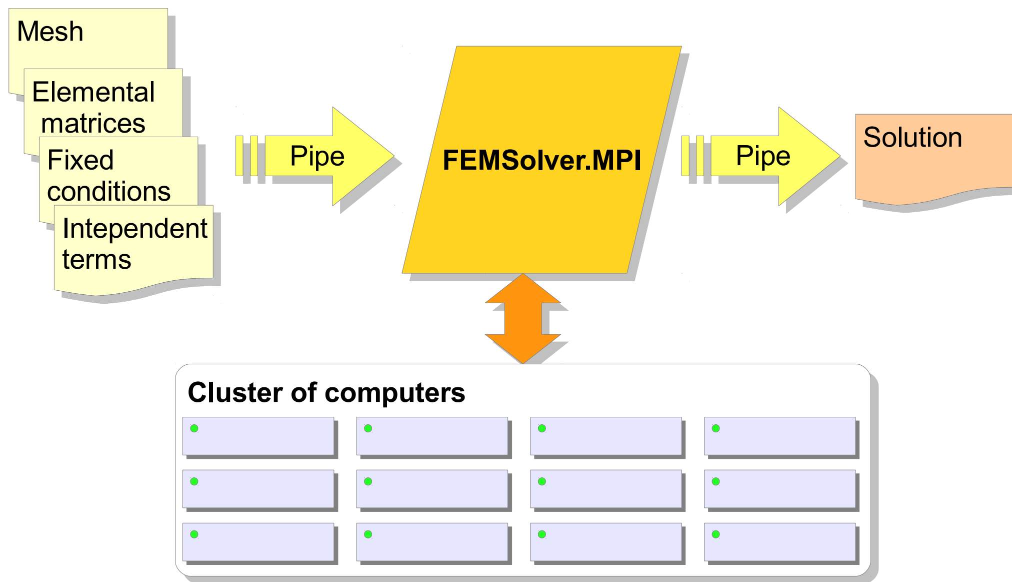
This flexible schema allows an used using any programming languaje (C/C++, Fortran, Python, C#, Java, etc.) to solve large systems of equations resulting from finite element discretizations.

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

FEMSolver helps to focus on the solution of the problem and not in how to implement it.

FEMSolver.MPI

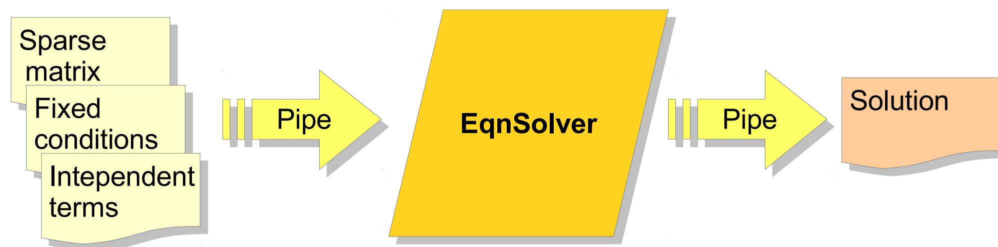
FEMSolver.MPI is a similar program to FEMSolver, but instead of solving the system of equations using a single computer, it can use a cluster of computers to distribute the workload and solve even larger systems of equations.



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

EqnSolver

This program was designed to solve systems of equations from finite volume and finite differences problems. It works in a similar way than FEMSolver, but instead of mesh and elemental matrices, it takes as input a sparse matrix.



It can use any of the solvers of the FEMT library.

MatSolver

Another simple way to access the FEMT libary solvers is through systems of equations written in the MatLab file format, MatSolver reads this file, calls any of the solvers available and stores the result in a file with MatLab format.

Sparse matrices

In problems modeled with finite element or finite volume methods is common to have to solve 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.

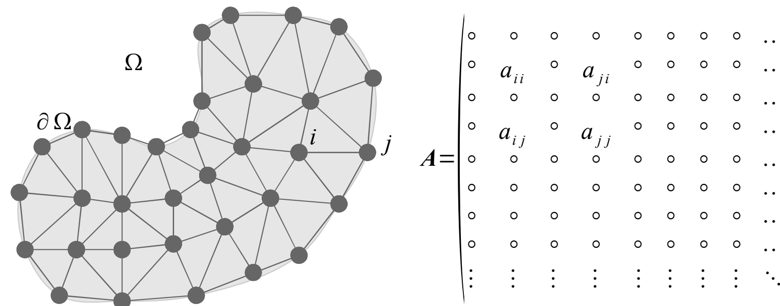


Figure 1. Discretized domain (mesh) and its matrix representation.

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

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

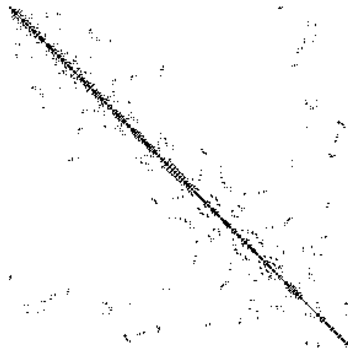


Figure 2. Black dots indicates a non zero entry in the matrix

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

representation

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

Row	Values	Indexes
1	8 4	1 2
2	1 3	3 4
3	2 1 7	1 3 5
4	9 3 1	2 3 6
5	5	6

$v_4^A=(9,3,1)$

$j_4^A=(2,3,6)$

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

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

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 $A=LL^{\mathrm{T}}$ 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 L entries are

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

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

Reordering rows and columns

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

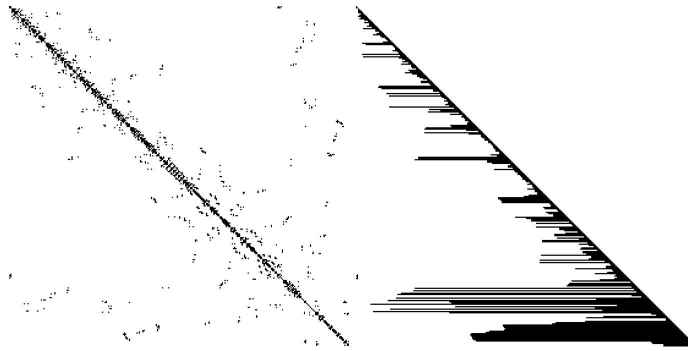


Figure 3. Left: non-zero entries of $\boldsymbol{A}$. Right: non-zero entries of $\boldsymbol{L}$ (Cholesky factorization of $\boldsymbol{A}$)

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.

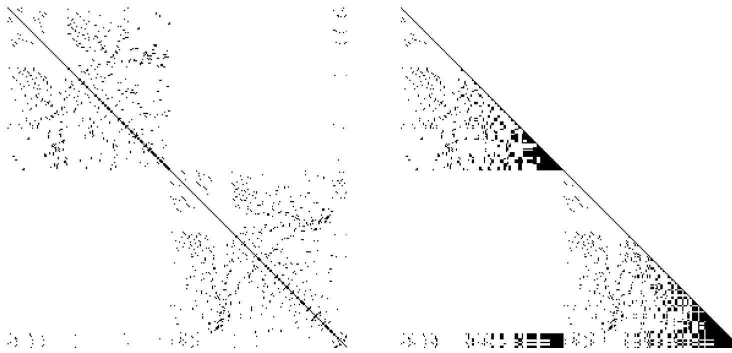


Figure 4. Left: non-zero entries of reordered $\boldsymbol{A}$. Right: non-zero entries of $\boldsymbol{L}$.

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 $\operatorname{adj}(x_1)$ 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:

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

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

Figure 5. 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 (1) and (2) 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]:

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

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.

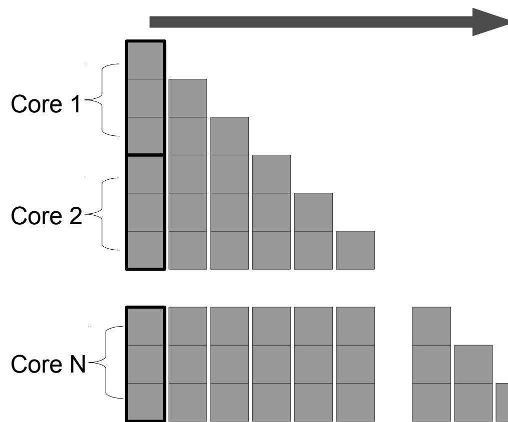


Figure 6. Calculation order to parallelize the Cholesky factorization.

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 paralellization was made using the OpenMP schema.

LDL' factorization

A similar schema can be used for this factorization, formulae to calculate $\boldsymbol{L}$ entries are

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

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

Using sparse matrices, we can use the following

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

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

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, like the ones resulting of the finite element and finite volume methods. The minimum degree algorithm gives also a good ordering for factorization. In this case $\boldsymbol{L}$ and $\boldsymbol{U}^{\mathrm{T}}$ will have the same structure.

Formulae to calculate $\boldsymbol{L}$ and $\boldsymbol{U}$ (using Doolittle's algorithm) are

$$U_{ij}=A_{ij}-\sum_{k=1}^{j-1} L_{ik} U_{kj} \text{ for } i>j,$$

$$L_{ji}=\frac{1}{U_{ii}}\left(A_{ji}-\sum_{k=1}^{i-1} L_{jk} U_{ki}\right) \text{ for } i>j,$$

$$U_{ii}=A_{ii}-\sum_{k=1}^{i-1} L_{ik} U_{ki},\ L_{ii}=1.$$

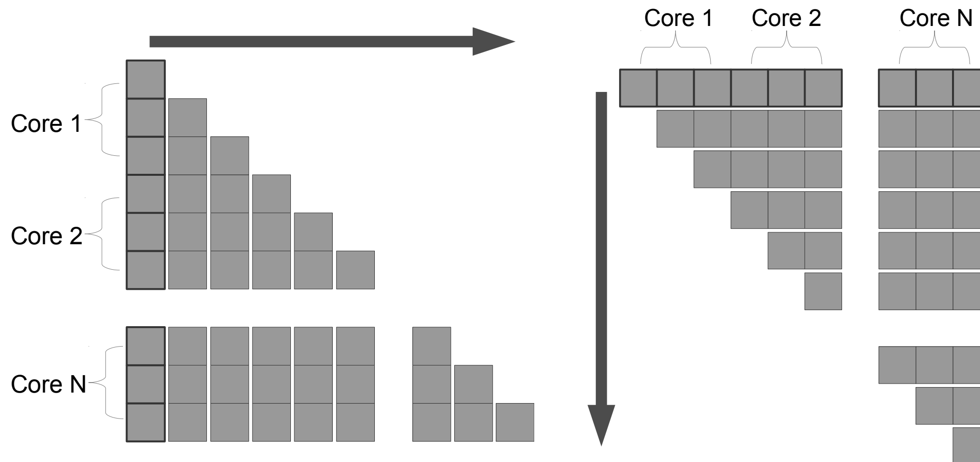
By storing these matrices using sparse compressed row, we can rewrite them as

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

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

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

To parallelize the algorithm, the fill of $\boldsymbol{U}$ must be done row by row, each row filled in parallel, $\boldsymbol{L}$ must be filled column by column, each one in parallel. The sequence to fill $\boldsymbol{L}$ y $\boldsymbol{U}$ in parallel is shown in the following figures.



Similarity to the Cholesky algorithm, to improve performance we will store $\boldsymbol{L}$, $\boldsymbol{U}$ and $\boldsymbol{U}^{\mathrm{T}}$ matrices using CRS. It is shown in the next algorithm [Varg10]:

For $j \leftarrow 1 \ldots n$
    $U_{jj} \leftarrow A_{jj}$
    For $q \leftarrow 1 \ldots \left(\left| \boldsymbol{V}_j(\boldsymbol{L}) \right| - 1 \right)$
        $U_{jj} \leftarrow U_{jj} - \boldsymbol{V}_j^q(\boldsymbol{L}) \boldsymbol{V}_j^q(\boldsymbol{U})$
    $L_{jj} \leftarrow 1$
    $U_{jj}^{\mathrm{T}} \leftarrow U_{jj}$
    Parallel for $q \leftarrow 2 \ldots \left| \boldsymbol{J}_j(\boldsymbol{L}^{\mathrm{T}}) \right|$
        $i \leftarrow \boldsymbol{J}_j^q(\boldsymbol{L}^{\mathrm{T}})$
        $L_{ij} \leftarrow A_{ij}$
        $U_{ji}^{\mathrm{T}} \leftarrow A_{ji}$
        $r \leftarrow 1;\ \rho \leftarrow \boldsymbol{J}_i^r(\boldsymbol{L})$
        $s \leftarrow 1;\ \sigma \leftarrow \boldsymbol{J}_j^s(\boldsymbol{L})$
        Repeat
            While $\rho < \sigma$
                $r \leftarrow r+1;\ \rho \leftarrow \boldsymbol{J}_i^r(\boldsymbol{L})$
            While $\rho > \sigma$
                $s \leftarrow s+1;\ \sigma \leftarrow \boldsymbol{J}_j^s(\boldsymbol{L})$
            While $\rho = \sigma$
                If $\rho = j$
                    Exit repeat loop
                $L_{ij} \leftarrow L_{ij} - \boldsymbol{V}_i^r(\boldsymbol{L}) \boldsymbol{V}_j^s(\boldsymbol{U}^{\mathrm{T}})$
                $U_{ji}^{\mathrm{T}} \leftarrow U_{ji}^{\mathrm{T}} - \boldsymbol{V}_j^s(\boldsymbol{L}) \boldsymbol{V}_i^r(\boldsymbol{U}^{\mathrm{T}})$
                $r \leftarrow r+1;\ \rho \leftarrow \boldsymbol{J}_i^r(\boldsymbol{L})$
                $s \leftarrow s+1;\ \sigma \leftarrow \boldsymbol{J}_j^s(\boldsymbol{L})$
        $L_{ij} \leftarrow \dfrac{L_{ij}}{U_{jj}}$
        $L_{ji}^{\mathrm{T}} \leftarrow L_{ij}$
        $U_{ji} \leftarrow U_{ij}^{\mathrm{T}}$

Solvers for triangular sparse matrices

Using $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ in $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$, we have

$$\boldsymbol{L}\boldsymbol{U}\boldsymbol{x} = \boldsymbol{y},$$

let $\boldsymbol{z} = \boldsymbol{U}\boldsymbol{x}$, we have to solve two triangular systems

$$\boldsymbol{L}\boldsymbol{z} = \boldsymbol{y},\ \boldsymbol{U}\boldsymbol{x} = \boldsymbol{z},$$

for $\boldsymbol{L}\boldsymbol{z} = \boldsymbol{y}$ we solve for $\boldsymbol{z}$ making a forward substitution with

$$z_i = \frac{1}{L_{i,i}} \left(y_i - \sum_{\substack{k \in \boldsymbol{J}_i(\boldsymbol{L}) \\ k<i}} L_{i,k} z_k \right),$$

finally, $\boldsymbol{U}\,\boldsymbol{x}=\boldsymbol{z}$ is solved for $\boldsymbol{x}$ making a backward substitution

$$x_i = \frac{1}{U_{i,i}} \left(z_i - \sum_{\substack{k \in \boldsymbol{J}_i(\boldsymbol{U}) \\ k>i}}^{n} U_{i,k} x_k \right).$$

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:

$\boldsymbol{x}_0$, initial approximation
$\boldsymbol{r}_0 \leftarrow \boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}_0$, initial gradient
$\boldsymbol{q}_0 \leftarrow \boldsymbol{M}\boldsymbol{r}_0$
$\boldsymbol{p}_0 \leftarrow \boldsymbol{q}_0$, initial descent direction
$k \leftarrow 0$
while $\|\boldsymbol{r}_k\|>\varepsilon$
$\quad \alpha_k \leftarrow -\frac{\boldsymbol{r}_k^{\mathrm{T}}\boldsymbol{q}_k}{\boldsymbol{p}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_k}$
$\quad \boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k$
$\quad \boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k-\alpha_k\boldsymbol{A}\boldsymbol{p}_k$
$\quad \boldsymbol{q}_{k+1} \leftarrow \boldsymbol{M}\boldsymbol{r}_{k+1}$
$\quad \beta_k \leftarrow \frac{\boldsymbol{r}_{k+1}^{\mathrm{T}}\boldsymbol{q}_{k+1}}{\boldsymbol{r}_k^{\mathrm{T}}\boldsymbol{q}_k}$
$\quad \boldsymbol{p}_{k+1} \leftarrow \boldsymbol{q}_{k+1}+\beta_k\boldsymbol{p}_k$
$\quad k \leftarrow k+1$

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,

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

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].

Jacobi preconditioner

The diagonal part of $\boldsymbol{M}^{-1}$ is stored as a vector,

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

Parallelization of this algorithm is straightforward, because the calculus 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 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 $\boldsymbol{A}$).

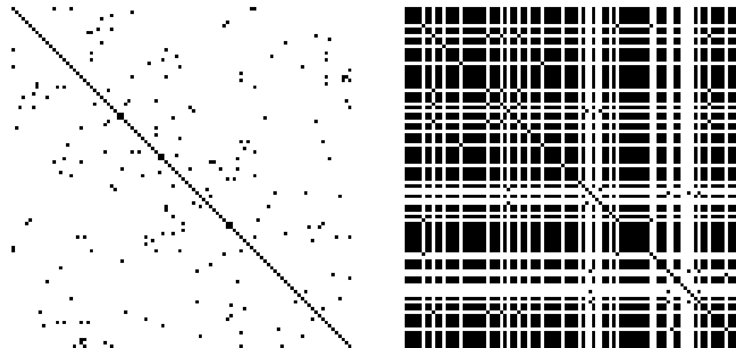
- For $0<l<m$ the structure of $\boldsymbol{G}_l$ is creating having a number of entries between $\boldsymbol{L}$ and the lower triangular form of $\boldsymbol{A}$, making easy to control the sparsity of the preconditioner.

Values of $\boldsymbol{G}_l$ are filled using (4) and (5). This preconditioner is not always stable [Golu96 p535].

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.



Ejemplo de las estructuras de una matriz rala y de su inversa

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

$$F(\boldsymbol{M})=\|\boldsymbol{I}-\boldsymbol{A}\boldsymbol{M}\|_{\mathrm{F}}^2. \tag{3}$$

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 (3) 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 (3), we minimize $\|\boldsymbol{I}-\boldsymbol{G}_l\boldsymbol{L}\|_{\mathrm{F}}^2$, it is noticeable that this can be done without knowing $\boldsymbol{L}$, solving the equations

$$(\boldsymbol{G}_l\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}})_{ij}=(\boldsymbol{L}^{\mathrm{T}})_{ij},\ (i,j)\in\boldsymbol{S}_{\boldsymbol{L}},$$

this is equivalent to

$$(\boldsymbol{G}_l \boldsymbol{A})_{ij} = (\boldsymbol{I})_{ij},\ (i,j) \in \boldsymbol{S}_L,$$

$\boldsymbol{S}_L$ contains the structure of $\boldsymbol{G}_l$.

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.

The algorithm to calculate the entries of $\boldsymbol{G}_l$ is:

```
Let S_l be the structure of G_l
for j ← 1…n
    for ∀(i, j)∈S_l
        solve (A G_l)_ij = δ_ij
```

Entries of $\boldsymbol{G}_l$ are calculated by rows. To solve $(\boldsymbol{A}\boldsymbol{G}_l)_{ij} = \delta_{ij}$ means that, if $m = \eta((\boldsymbol{G}_l)_j)$ is the number of non zero entries of the column j of $\boldsymbol{G}_l$, then we have to solve a small SPD system of size $m \times m$.

A simple way to define a structure $\boldsymbol{S}_l$ for $\boldsymbol{G}_l$ is to simply take the lower triangular part of $\boldsymbol{A}$.

Another way is to construct $\boldsymbol{S}_l$ from the structure take from

$$\tilde{\boldsymbol{A}},\ \tilde{\boldsymbol{A}}^2,\ ...,\ \tilde{\boldsymbol{A}}^l,$$

where $\tilde{\boldsymbol{A}}$ is a truncated version of $\boldsymbol{A}$,

$$\tilde{A}_{ij} = \begin{cases} 1 & \text{if } i=j \text{ o } \left|\left(\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}\right)_{ij}\right| > \text{threshold} \\ 0 & \text{other case} \end{cases},$$

the threshold is a non negative number and the diagonal matrix $\boldsymbol{D}$ is

$$\tilde{D}_{ii} = \begin{cases} |A_{ii}| & \text{if } |A_{ii}| > 0 \\ 1 & \text{other case} \end{cases}.$$

Powers $\tilde{\boldsymbol{A}}^l$ can be calculated combining rows of $\tilde{\boldsymbol{A}}$. Lets denote the k-th row of $\tilde{\boldsymbol{A}}^l$ as $\tilde{\boldsymbol{A}}^l_{k,:}$,

$$\tilde{\boldsymbol{A}}^l_{k,:} = \tilde{\boldsymbol{A}}^{l-1}_{k,:}\tilde{\boldsymbol{A}}.$$

The structure $\boldsymbol{S}_l$ will be the lower triangular part of $\tilde{\boldsymbol{A}}^l$. With this truncated $\tilde{\boldsymbol{A}}^l$, a $\tilde{\boldsymbol{G}}^l$ is calculated using the previous algorithm to create a preconditioner $\boldsymbol{M} = \tilde{\boldsymbol{G}}_l^{\mathrm{T}}\tilde{\boldsymbol{G}}_l$.

The vector $\boldsymbol{q}_k \leftarrow \boldsymbol{M}\,\boldsymbol{r}_k$ is calculated with two matrix-vector products,

$$\boldsymbol{M}\,\boldsymbol{r}_k = \tilde{\boldsymbol{G}}_l^{\mathrm{T}}(\tilde{\boldsymbol{G}}_l \boldsymbol{r}_k).$$

Parallel biconjugated gradient

The biconjugate gradient method is based on the conjugate gradient method, it solves linear systems of equations

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

in this case $\boldsymbol{A} \in \mathbb{R}^{m \times m}$ does not need to be symmetric.

This method requires to calculate a pseudo-gradient $\tilde{\boldsymbol{g}}_k$ and a pseudo-direction of descent $\tilde{\boldsymbol{p}}_k$. The algorithm construcs the pseudo-gradients $\tilde{\boldsymbol{g}}_k$ to be orthogonal to the gradients $\boldsymbol{g}_k$, similarly, the pseudo-directorions of descent $\tilde{\boldsymbol{p}}_k$ to be $\boldsymbol{A}$-orthogonal to the descent directions $\boldsymbol{p}_k$ [Meie94 pp6-7].

If the matrix $\boldsymbol{A}$ is simmetric, then this method is equivalent to the conjugate gradient.

The drawbacks are, it does not assure convergence in n iterations as conjugate gradient does, it requires to do two matrix-vector multiplications.

The algorithm is [Meie92]:

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

This method can also be preconditioned.

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

Preconditioning with incomplete LU factorization

This preconditioner is analog to the incomplete Cholesky preconditioner, this one is formed by two matrices

$$\boldsymbol{M} = \boldsymbol{G}_k \boldsymbol{H}_k,$$

where $\boldsymbol{G}_k$ is a sparse lower triangular matrix, and $\boldsymbol{H}_k$ is an sparse upper triangular matrix. They have a similar structure to the factorization LU of $\boldsymbol{A} \in \mathbb{R}^{n \times n}$.

- The structure of $\boldsymbol{G}_0$ is equal to the structure of the lower triangular part of $\boldsymbol{A}$.
- The structure of $\boldsymbol{H}_0$ is equal to the structure of the upper triangular part of $\boldsymbol{A}$.

- The structure of $\boldsymbol{G}_m$ is equal to the strucure of $\boldsymbol{L}$ (from the LU factorization of $\boldsymbol{A}$.)
- The structure of $\boldsymbol{H}_m$ is equal to the strucure of $\boldsymbol{U}$ (from the LU factorization of $\boldsymbol{A}$.)
- For $0<k<m$ the structure of $\boldsymbol{G}_k$ is created to have a number of entries between $\boldsymbol{G}_0$ and $\boldsymbol{G}_m$, making easy to control the sparsity of the preconditioner. Similarly for $\boldsymbol{H}_k$.

Values of $\boldsymbol{G}_k$ and $\boldsymbol{H}_k$ are filled using the formulae to calculate the LU factorization,

$$H_{ij}=A_{ij}-\sum_{k=1}^{j-1} G_{ik} H_{kj} \quad \text{for } i>j,$$

$$G_{ji}=\frac{1}{H_{ii}}\left(A_{ji}-\sum_{k=1}^{i-1} G_{jk} H_{ki}\right) \text{ for } i>j,$$

$$H_{ii}=A_{ii}-\sum_{k=1}^{i-1} G_{ik} H_{ki},\ G_{ii}=1.$$

Approximate inverse preconditioner

An appoximate inverse for non-simmetric matrices can be constructed using the minimal resiual [Chow98]. The algorithm to build $\boldsymbol{M}$ is:

Let $\boldsymbol{M}=\boldsymbol{M}_0$
for $j \leftarrow 1\ldots n$
  $\boldsymbol{m}_j \leftarrow \boldsymbol{M}\boldsymbol{e}_j$
  for $i \leftarrow 1\ldots s$
    $\boldsymbol{r}_j \leftarrow \boldsymbol{e}_j-\boldsymbol{A}\boldsymbol{m}_j$
    $\alpha_j \leftarrow \dfrac{\boldsymbol{r}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_j}{\left(\boldsymbol{A}\boldsymbol{r}_j\right)^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_j}$
    $\boldsymbol{m}_j \leftarrow \boldsymbol{m}_j+\alpha_j\boldsymbol{r}_j$
    Truncate less significative entries of $\boldsymbol{m}_j$

This is an iterative method, here s is small. If no sparse vectors are used, then this algorithm will have an order of $O\left(n^2\right)$.

To initialize the algorithm, we can use

$$\boldsymbol{M}_0=\alpha\boldsymbol{G},$$

with

$$\alpha=\frac{\operatorname{tr}\left(\boldsymbol{A}\boldsymbol{G}\right)}{\operatorname{tr}\left(\boldsymbol{A}\boldsymbol{G}\left(\boldsymbol{A}\boldsymbol{G}\right)^{\mathrm{T}}\right)},$$

$\boldsymbol{G}$ can be selected as $\boldsymbol{G}=\boldsymbol{I}$ or $\boldsymbol{G}=\boldsymbol{A}^{\mathrm{T}}$.

$\boldsymbol{M}_0=\alpha\boldsymbol{I}$ is the faster to calculate, but $\boldsymbol{M}_0=\alpha\boldsymbol{A}^{\mathrm{T}}$ produces a better initial approximation in some cases.

The next step is to use sparse vectors. Let $\boldsymbol{b}$ be a sparse vector, $\eta\left(\boldsymbol{b}\right)$ indicates the number of non-zero entries in $\boldsymbol{b}$. A technique to select the most significative entries of $\boldsymbol{m}_j$ is truncate entries in the descent-direction of the minimal residual. Approximate inverse via minimal residual iteration with dropping in the search direction algorithm is shown next.

Let $\boldsymbol{M}=\boldsymbol{M}_0$
for $j \leftarrow 1\ldots n$
  Let $\boldsymbol{m}_j \leftarrow \boldsymbol{M}\,\boldsymbol{e}_j$ be an sparse vector
  Let $\boldsymbol{d}_j$ a sparse vector with the same strucure than $\boldsymbol{m}_j$
  for $i \leftarrow 1\ldots s$
    $\boldsymbol{r}_j \leftarrow \boldsymbol{e}_j-\boldsymbol{A}\,\boldsymbol{m}_j$
    $\boldsymbol{d}_j \leftarrow \boldsymbol{r}_j$ only take the entries of $\boldsymbol{r}_j$ with index in $\boldsymbol{d}_j$
    $\alpha_j \leftarrow \dfrac{\boldsymbol{r}_j^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{d}_j}{\left(\boldsymbol{A}\,\boldsymbol{d}_j\right)^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{d}_j}$
    $\boldsymbol{m}_j \leftarrow \boldsymbol{m}_j+\alpha_j\boldsymbol{d}_j$
    if $\eta\left(\boldsymbol{m}_j\right)<lfil$
      add $\max\left|\left(\boldsymbol{r}_j\right)_k\right|$, such that k does not exists in the structure of $\boldsymbol{m}_j$

Cons:

- The final structure of $\boldsymbol{M}$ is not symetric.
- $\boldsymbol{M}$ can be singular if s is small.

Schur sustructuring 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 paralle.

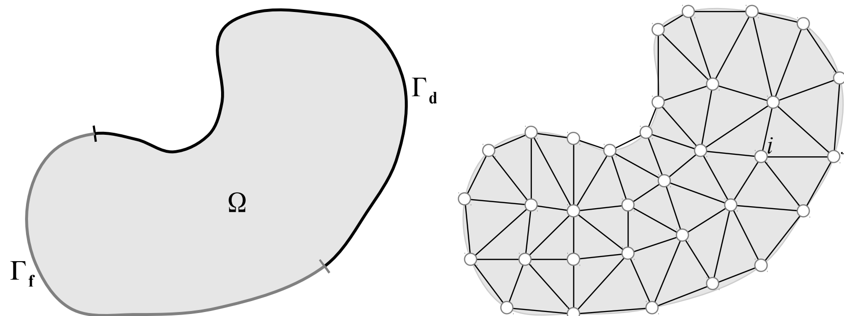


Figure 7. Finite element domain (left), domain discretization (center), partitioning (right).

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

$$\mathbf{K}\,\mathbf{d}=\mathbf{f}, \tag{4}$$

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

Partitioning

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 8. Partitioning example.

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

$$\begin{pmatrix} \mathbf{K}_1^{\mathrm{II}} & & \mathbf{0} & & & \mathbf{K}_1^{\mathrm{IB}} \\ & \mathbf{K}_2^{\mathrm{II}} & & & & \mathbf{K}_2^{\mathrm{IB}} \\ \mathbf{0} & & \mathbf{K}_3^{\mathrm{II}} & & & \mathbf{K}_3^{\mathrm{IB}} \\ \vdots & & & \ddots & & \vdots \\ & & & & \mathbf{K}_p^{\mathrm{II}} & \mathbf{K}_p^{\mathrm{IB}} \\ \mathbf{K}_1^{\mathrm{BI}} & \mathbf{K}_2^{\mathrm{BI}} & \mathbf{K}_3^{\mathrm{BI}} & \cdots & \mathbf{K}_p^{\mathrm{BI}} & \mathbf{K}^{\mathrm{BB}} \end{pmatrix} \begin{pmatrix} \mathbf{d}_1^{\mathrm{I}} \\ \mathbf{d}_2^{\mathrm{I}} \\ \mathbf{d}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{d}_p^{\mathrm{I}} \\ \mathbf{d}^{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1^{\mathrm{I}} \\ \mathbf{f}_2^{\mathrm{I}} \\ \mathbf{f}_3^{\mathrm{I}} \\ \vdots \\ \mathbf{f}_p^{\mathrm{I}} \\ \mathbf{f}^{\mathrm{B}} \end{pmatrix}. \qquad (5)$$

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 de boundary and nodes inside the partition.

Figure 9. Substructuring example with three partitions.

Thus, the sistem 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 partitioning the mesh we used the METIS library [Kary99].

Schur complement method

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

$$\mathbf{d}_i^{\mathrm{I}} = \left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1} \left(\mathbf{f}_i^{\mathrm{I}} - \mathbf{K}_i^{\mathrm{IB}} \mathbf{d}^{\mathrm{B}}\right). \qquad (6)$$

After applying Gaussian elimination by blocks on (5), the reduced system of equations becomes

$$\left(\mathbf{K}^{\mathrm{BB}}-\sum_{i=1}^{p}\mathbf{K}_i^{\mathrm{BI}}\left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1}\mathbf{K}_i^{\mathrm{IB}}\right)\mathbf{d}^{\mathrm{B}}=\mathbf{f}^{\mathrm{B}}-\sum_{i=1}^{p}\mathbf{K}_i^{\mathrm{BI}}\left(\mathbf{K}_i^{\mathrm{II}}\right)^{-1}\mathbf{f}_i^{\mathrm{I}}. \tag{7}$$

Once the vector $\mathbf{d}^{\mathrm{B}}$ is computed using (7), we can calculate the internal unknowns $\mathbf{d}_i^{\mathrm{I}}$ with (6).

It is not necessary to calculate the inverse in (7). 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 proced column by column using an extra vector $\mathbf{t}$, and solving for $c=1\ldots n$

$$\mathbf{K}_i^{\mathrm{II}}\mathbf{t}=\left[\mathbf{K}_i^{\mathrm{IB}}\right]_c, \tag{8}$$

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

$$\mathbf{K}_i^{\mathrm{II}}\mathbf{t}=\mathbf{f}_i^{\mathrm{I}}, \tag{9}$$

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 (7), this can be written as

$$\left(\mathbf{K}^{\mathrm{BB}}-\sum_{i=1}^{p}\bar{\mathbf{K}}_i^{\mathrm{BB}}\right)\mathbf{d}^{\mathrm{B}}=\mathbf{f}^{\mathrm{B}}-\sum_{i=1}^{p}\bar{\mathbf{f}}_i^{\mathrm{B}}, \tag{10}$$

once (10) is solved, we can calculate the inner results of each partition using (6).

Since $\mathbf{K}_i^{\mathrm{II}}$ is sparse and has to be solved many times in (8), a efficient way to proceed is to use a Cholesky factorization of $\mathbf{K}_i^{\mathrm{II}}$. To reduce memory usage and increce speed a sparse Cholesky factorization has to be implemented, this method is explained below.

In case of (10), $\mathbf{K}^{\mathrm{BB}}$ 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 preconditioner is used. This algorithm is described below.

One benefit of this method is that the condition number of the system is reduced when solving (10), this decreases the number of iterations needed to converge.

Parallelization

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].

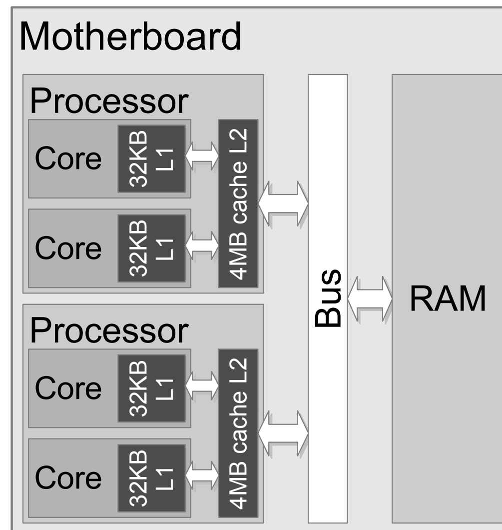


Figure 10. Schematic of a multi-processor and multi-core computer.

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 necesary 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.

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.

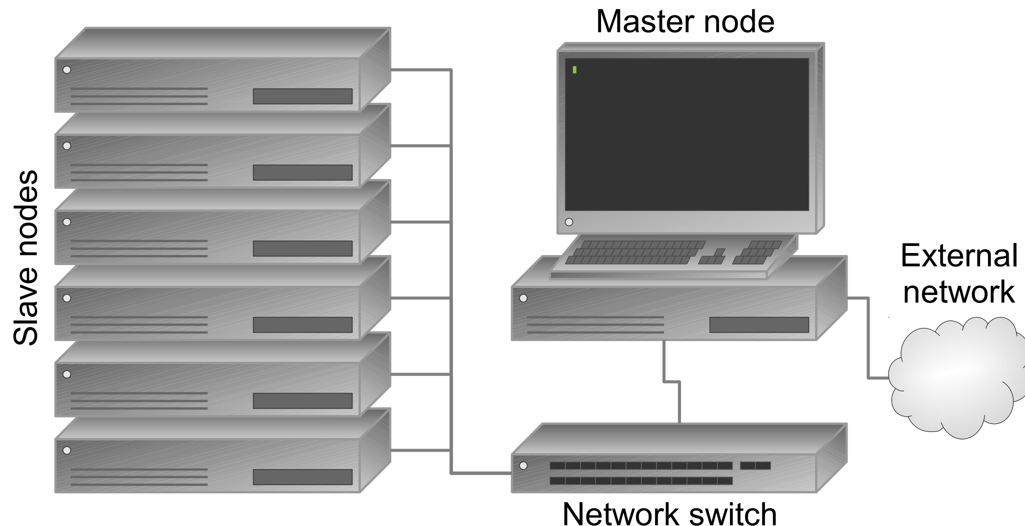


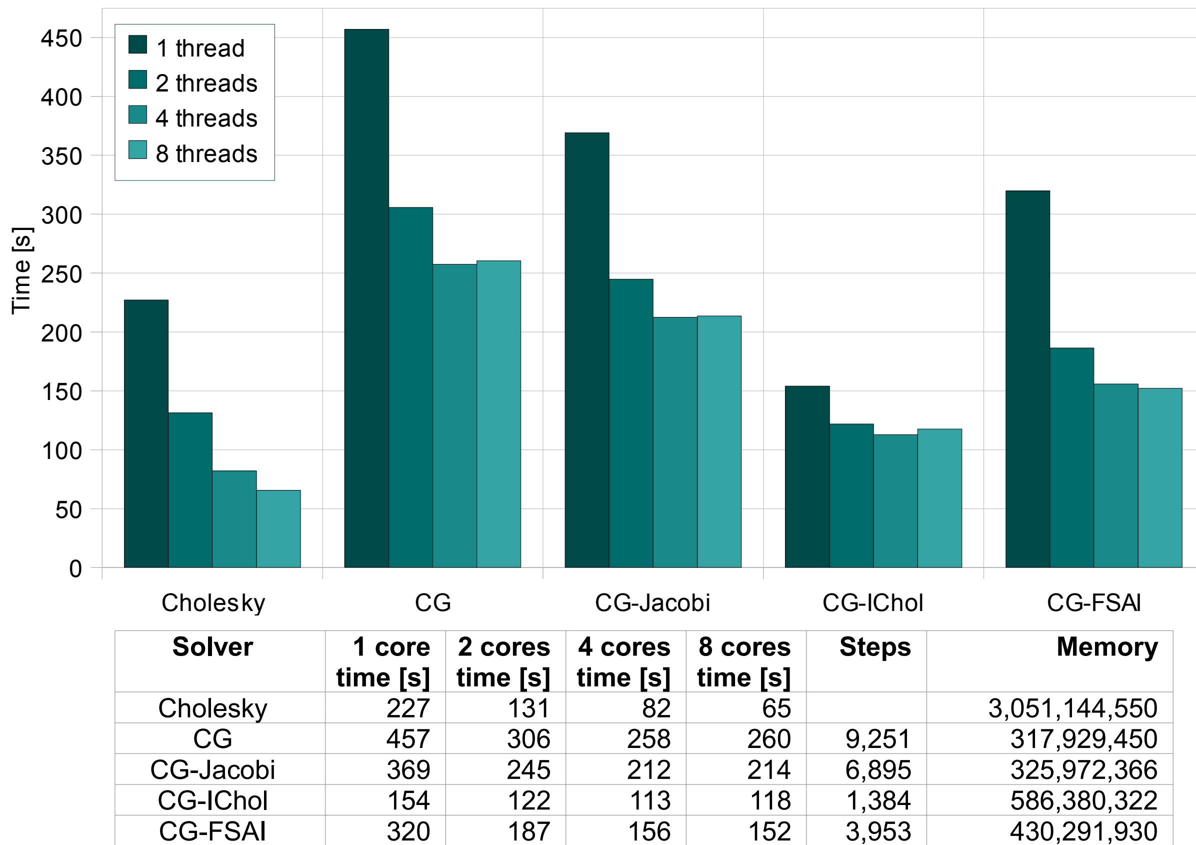
Figure 11. Diagram of a Beowulf cluster of computers.

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.

Numerical experiments

Solutions with OpenMP

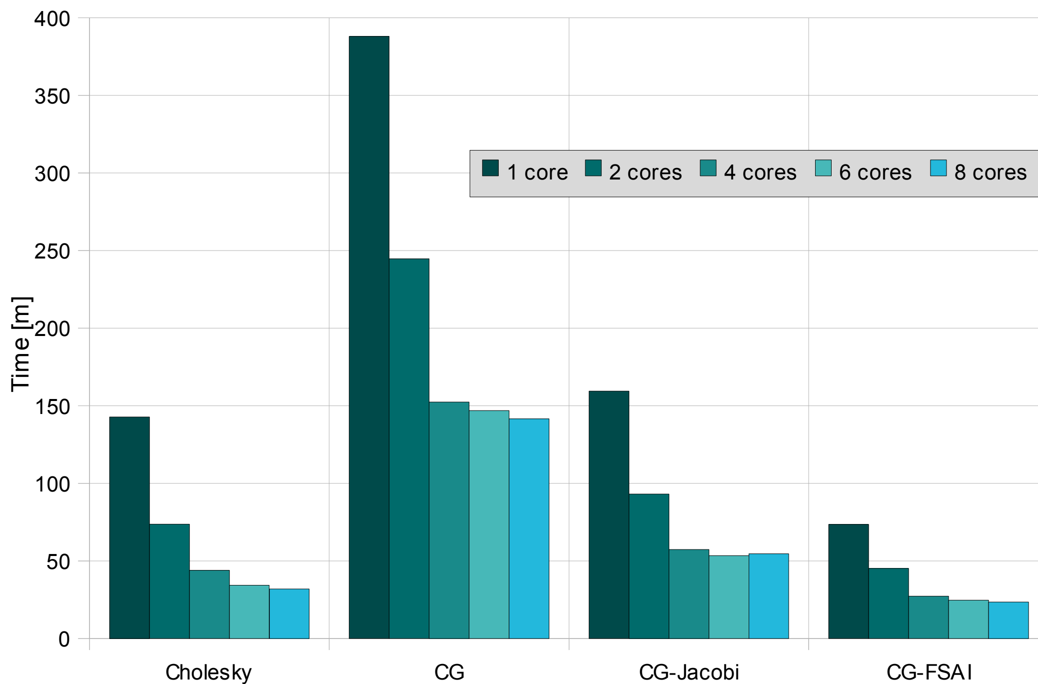
First we will show results for the parallelization of solvers with OpenMP. The next example is a 2D solid deformation with 501,264 elements, 502,681 nodes. A system of equations with 1’005.362 variables is formed, the number of non zero entries are $\eta(\boldsymbol{K})=18'062{,}500$, $\eta(\boldsymbol{L})=111'873{,}237$. Tolerance used in CG methods is $\|\boldsymbol{r}_k\|\geq 1\times 10^{-5}$.



Solver	1 core time [s]	2 cores time [s]	4 cores time [s]	8 cores time [s]	Steps	Memory
Cholesky	227	131	82	65		3,051,144,550
CG	457	306	258	260	9,251	317,929,450
CG-Jacobi	369	245	212	214	6,895	325,972,366
CG-IChol	154	122	113	118	1,384	586,380,322
CG-FSAI	320	187	156	152	3,953	430,291,930

The next example is a 3D solid model of a building that sustain deformation due to self-weight. Basement has fixed displacements.

The domain was discretized in 264,250 elements, 326,228 nodes, 978,684 variables, $\eta(\boldsymbol{K})=69\,'255{,}522$.



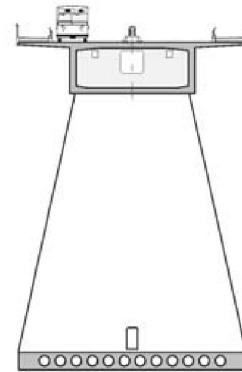
Solver	1 core time [m]	2 cores time [m]	4 cores time [m]	6 cores time [m]	8 cores time [m]	Memory
Cholesky	143	74	44	34	32	19,864,132,056
CG	388	245	152	147	142	922,437,575
CG-Jacobi	160	93	57	54	55	923,360,936
CG-FSAI	74	45	27	25	24	1,440,239,572

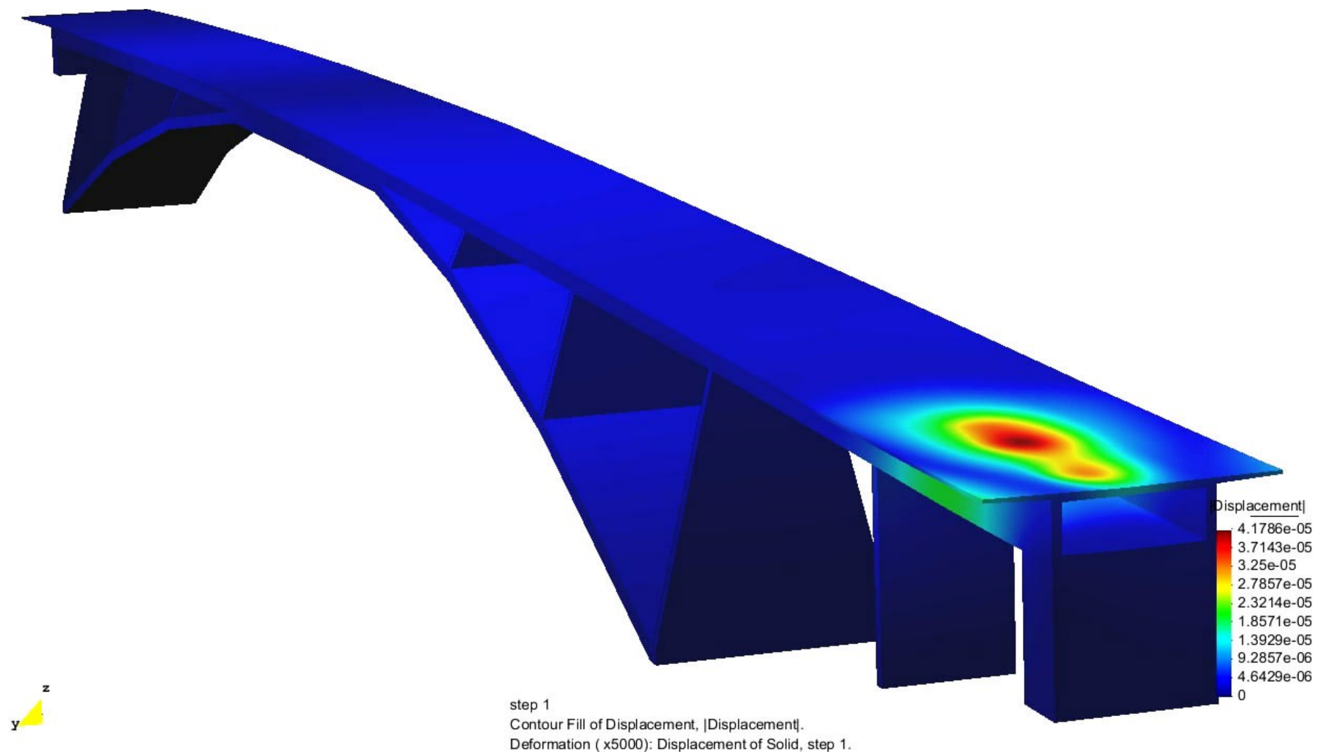
In this model, conjugate gradient with incomplete Cholesky factorization failed to converge.

Dynamic problems

Simulation of a 18 wheels 36 metric tons truck crossing the Infante D. Henrique Bridge. Pre and post-process where made using GiD (http://gid.cimne.upc.es).

Nodes	337,195
Elements	1'413,279
Element type	Tetrahedron
Time steps	372
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 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

Solutions with domain decomposition using MPI

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 1×10^{-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.

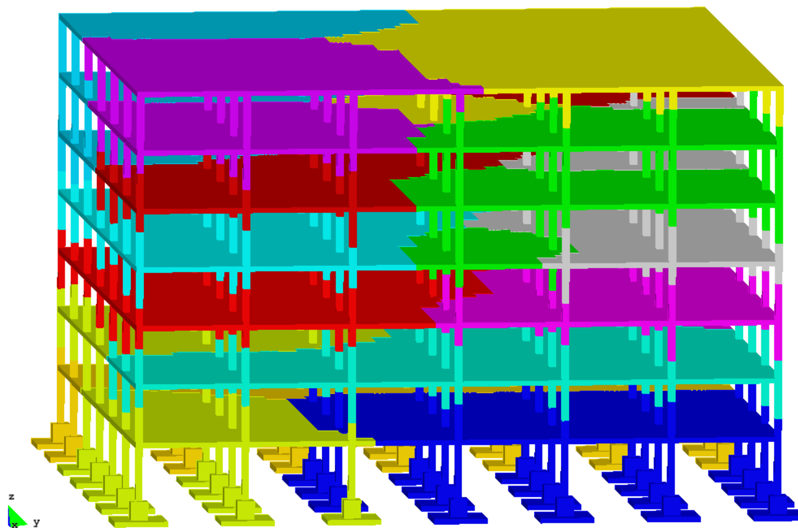
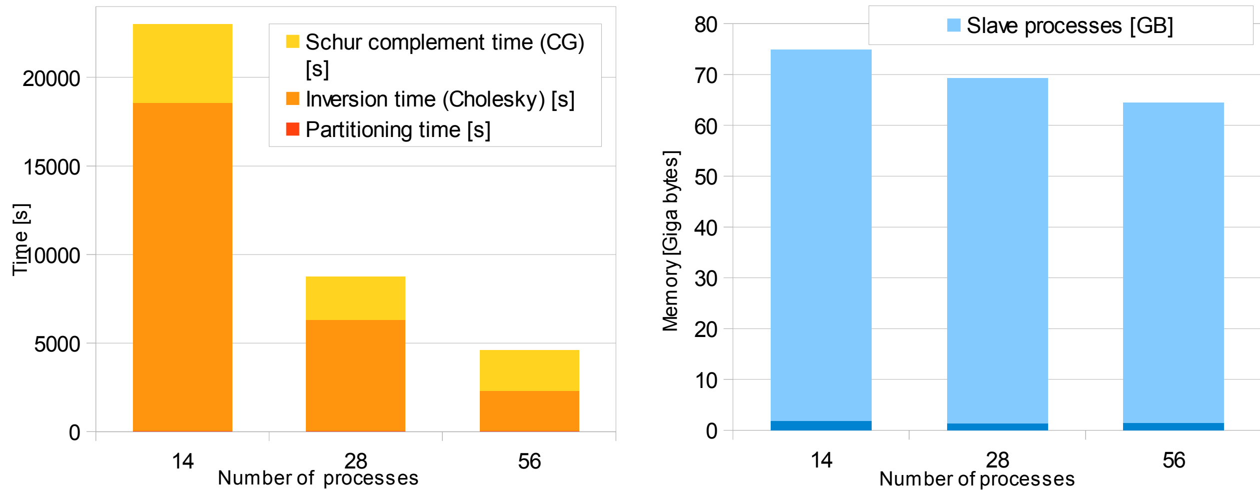


Figure 12. Substructuration of the domain.

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

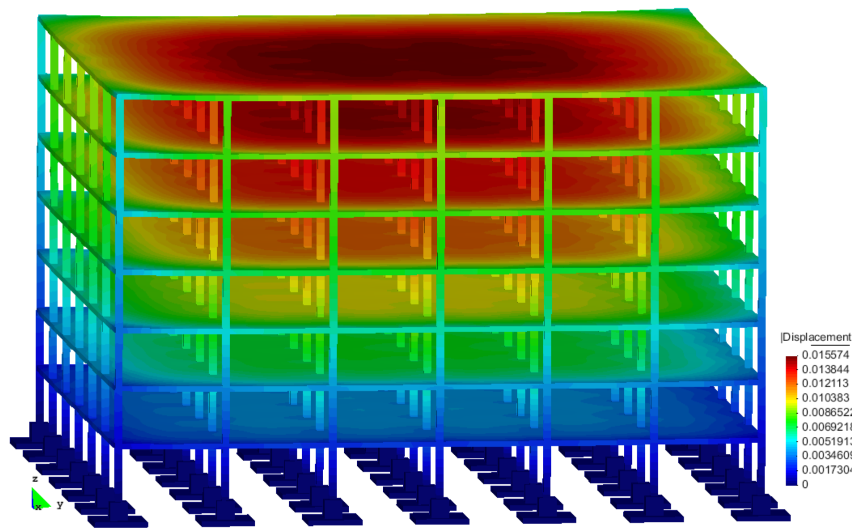


Figure 13. Resulting deformation.

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

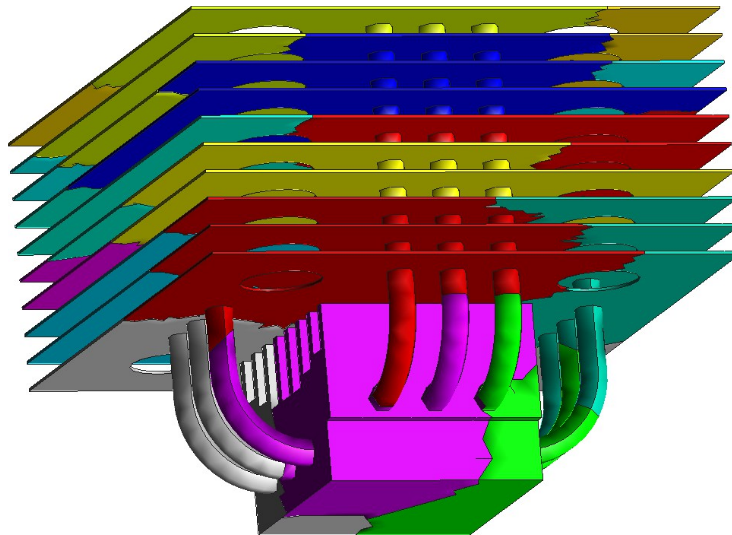
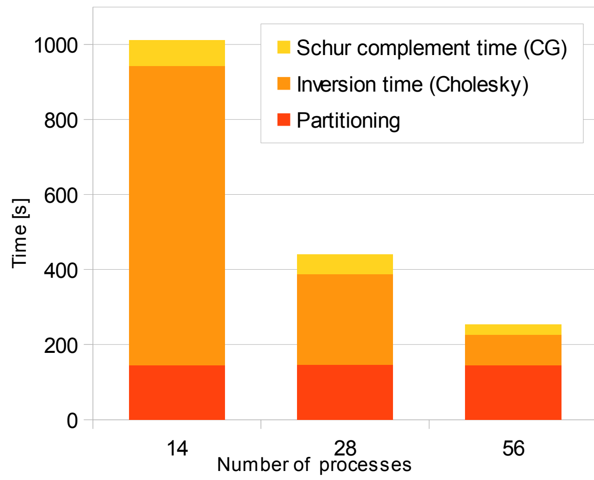
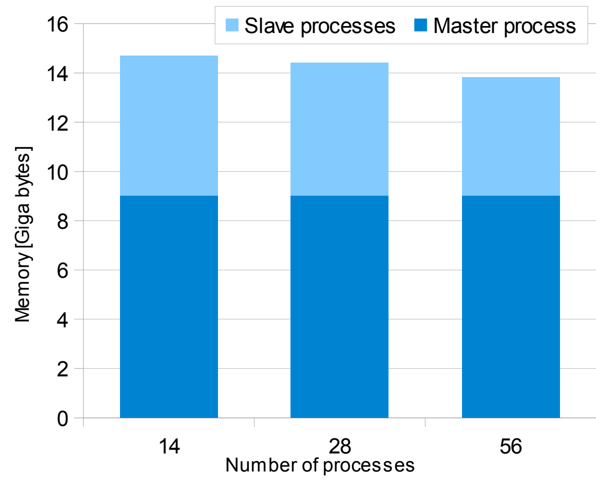


Figure 14. Substructuration of the domain.

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





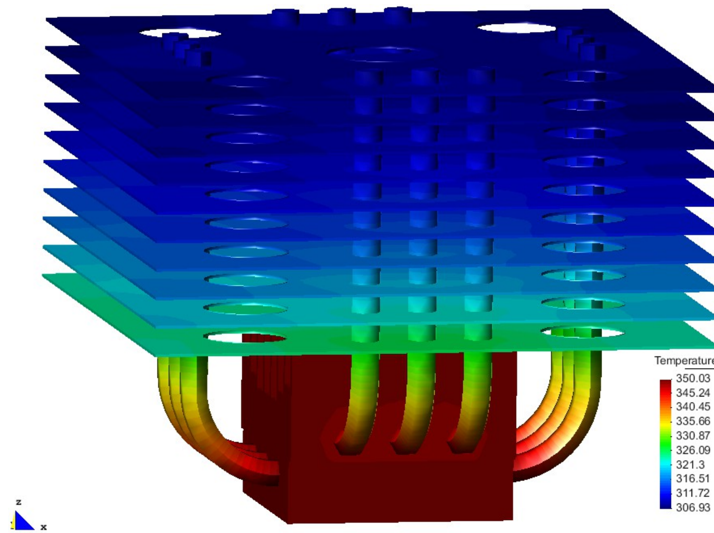


Figure 15. Resulting temperature distribution.

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 metalic square with Dirichlet conditions on all boundaries.

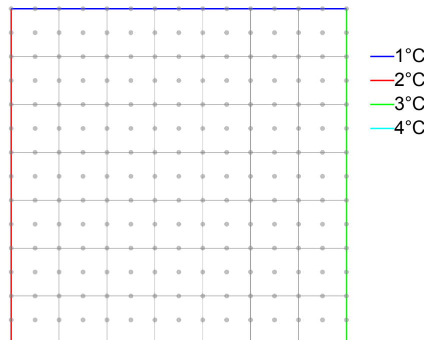
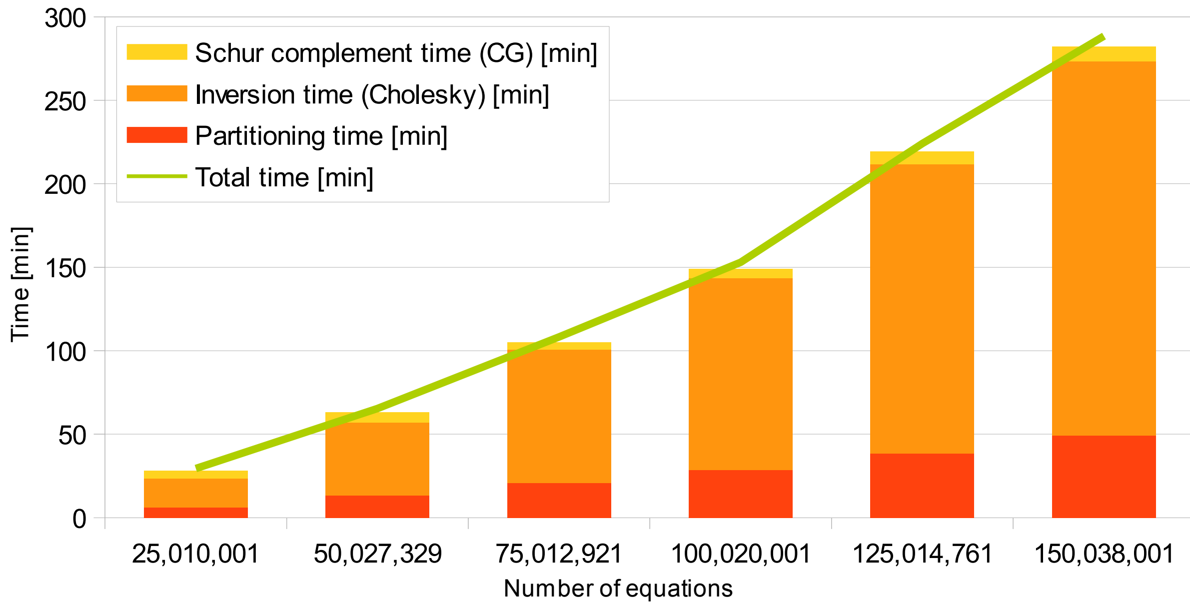


Figure 16. Geometry example.

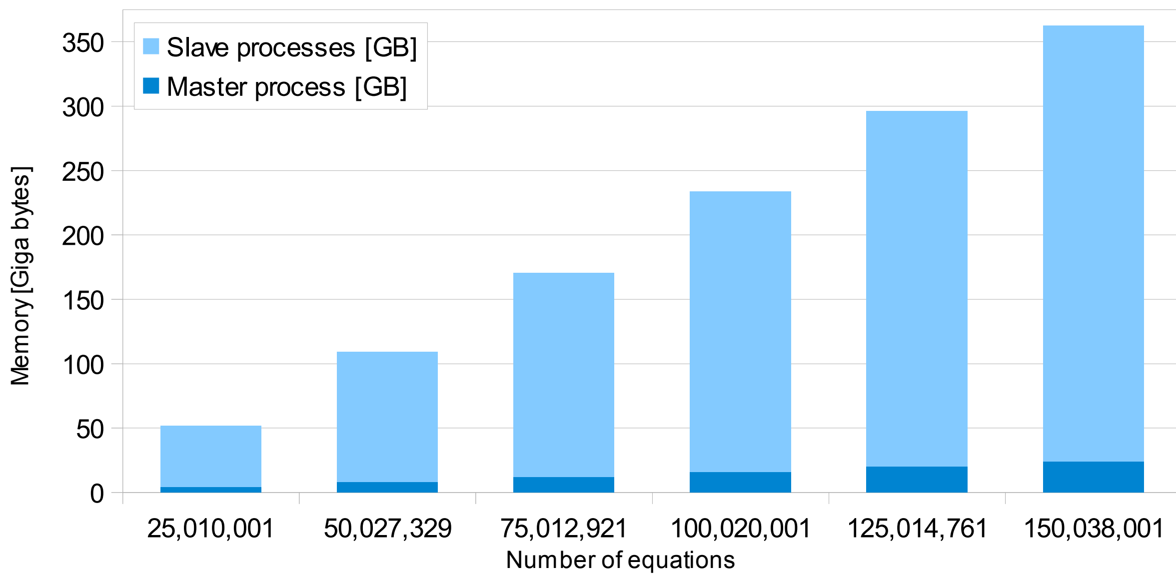
The domain was discretized using quadrilaterals with nine nodes, the discretization made was from 25 million nodes up to 150 million nodes. In all cases we divided the domain into 116 partitions.

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



Library license

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Library General Public License for more details.

You should have received a copy of the GNU Library General Public
License along with this library; if not, write to the Free
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

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.