FEMT, Open Source Tools for Solving Large Systems of Equations in Parallel

Miguel Vargas-Felix, Salvador Botello-Rionda

***Abstract*—We present a new open source library and tools for solving large linear systems of equations with sparse matrices resulting from simulations with finite element, finite volume and finite differences.**

FEMT is a multi-platform software (Windows, GNU/Linux, Mac OS), released as open source (GNU LGPL). FEMT can run from laptops up to clusters of computers. It has been programmed in modern standard C++, also has modules to access it easily from many programming languages, like Fortran, Python, Java, C++, C, etc.

In this work we will describe the tools and the solvers that are used, there are three kind: 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.

The interchange of information is done using pipes (memory allocated files). This makes easy to use the solvers included in FEMT from existing simulation codes without large code changes.

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

***Index Terms*—Numerical Linear Algebra, Sparse Matrices, Parallel Computing, Direct and Iterative Solvers, Schur Substructuration, Finite Element Method, Finite Volume Method.**

I. Introduction

FEMT aims to help people that use finite element, finite differences or isogeometric analisys to easily incorporate a solver for the sparse systems generated. The FEMT package is divided in three parts:

- The library, called also FEMT, was developed in standard C++ using templates extensively. It includes several routines for solving sparse systems of equations, like conjugate gradient, biconjugate gradient, Cholesky and LU factorizations, these were implemented with OpenMP parallelization support. Also, the library includes an implementation of the Schur substructuring method, it was implemented with MPI to run in clusters of computers.
- A set of tools for using the solvers included in the FEMT library in a easy way from any programming language. The motivation for these tools is to adapt the FEMT library to simulation codes developed in languages different to C++, this is common among research groups. There are two kinds of tools, one is to resolve finite element or isogeometric analisys problems (works using elemental matrices), the other is to solve problems from finite volume or finite differences (works using sparse matrices).
- Finite element simulation modules for GiD. GiD is a pre and post-processor developed by CIMNE, with it you can design a geometries, set materials and boundary conditions, mesh it, call a FEMT solver module and visualize the results. The modules (problem types) implemented so far are: linear solid deformation (static and dynamic), heat diffusion (static and dynamic) and electric potential (it calculates also capacitance matrices and sensitivity maps). These problem types use the FEMT library for solving the finite element problems. Several examples with different geometries are included

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

Source code, building instructions, tutorials and extra documentation can be found at:
http://www.cimat.mx/~miguelvargas/FEMT

Miguel Vargas-Felix is with Centre for Mathematical Research (CIMAT). Jalisco Alley w/n, Mineral de Valenciana, Guanajuato, Mexico 36240 (e-mail: miguelvargas@cimat.mx)
Salvador Botello-Rionda is with Centre for Mathematical Research (CIMAT). Jalisco Alley w/n, Mineral de Valenciana, Guanajuato, Mexico 36240 (e-mail: botello@cimat.mx)

II. Parallelization

A. Parallelization on multi-core computers

Tendency in modern computers is to increase the processing units (cores) to process data in parallel. FEMT uses OpenMP, it is a programming model that uses multiple threads to parallelize in multi-core computers. This 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 [1].

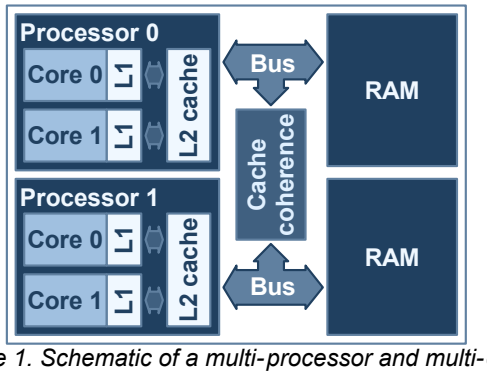


Figure 1. Schematic of a multi-processor and multi-core computer.

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 [2]. Also, it is necessary to consider that cache circuits are designed to be more efficient when reading continuous memory data in an ascendent sequence [2].

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.

The routines in FEMT were programmed to reduce these bottlenecks.

B. Computer clusters and MPI

Using domain decomposition a large system of equations can be divided into smaller problems that can be processed on several computers on a cluster, working concurrently to complete the global solution. A cluster (also known as Beowulf cluster [3]) consists of several multi-core computers (nodes) connected with a high speed network.

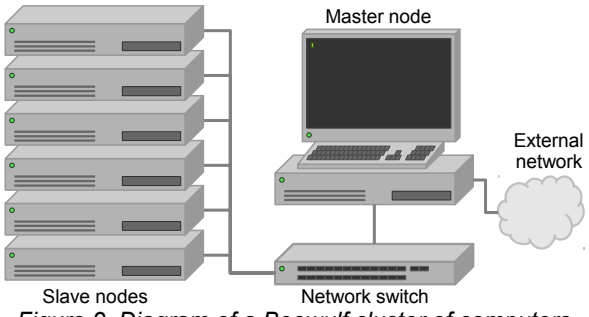


Figure 2. Diagram of a Beowulf cluster of computers.

To parallelize the program and move data among nodes we used the Message Passing Interface (MPI) schema [4], 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 is the standard for scientific applications on clusters.

III. Tools for solving large systems of equations in parallel

To adapt existing simulation codes, with tens or hundreds of thousands lines of code, to a new library with solvers, could imply to have a step learning curve and to rewrite a large amount of sections of code. The tools included with FEMT offer an alternative, add a single function to the code, this function saves the needed information into an archive (with a simple format). This archive is read by a program (FEMSolver and EqnSolver), process the information, solves the system of equations using parallelized routines, and return the solution in other archive. This archive is read by the same function in the simulation code.

To make the interchange of data efficient and fast, FEMT tools uses a kind of archive that is not saved on disk, but into the RAM, these archives are called named pipes. From the point of view of the function that saves and writes data, it is just a normal file. But it has the advantage of being stored on the RAM, making the interchange of data fast. Is a easy and efficient way of communicate two programs, its use is very common on Unix like systems (Linux, Mac OS or BSD). Windows operating system also supports this kind of files, but its usage is less common.

In conclusion, knowing how to write and read archives in any programming language allows to use the tools and solvers included in FEMT. The following sections describe these tools.

A. FEMSolver

FEMSolver is a program that assembles and solves finite element problems in parallel using the FEMT library on multi-core computers. It uses a very simple interface using pipes.

The sequence is as follows, first FEMSolver is executed, it will create the named pipes for data interchange, then it will wait until the simulation program sends data.

The simulation program only needs to write to the data pipe (default name for this file is */tmp/FEMData*) the information that describes the finite element problem: connectivity matrix, elemental matrices, vector of independent terms, and vector of fixed conditions. FEMSolver reads this data, assembles the global matrix, makes the reduction of this using the fixed conditions and calls the solver routine. After solving the system of equations FEMSolver returns the result vector in another pipe (by default */tmp/FEMResult*).

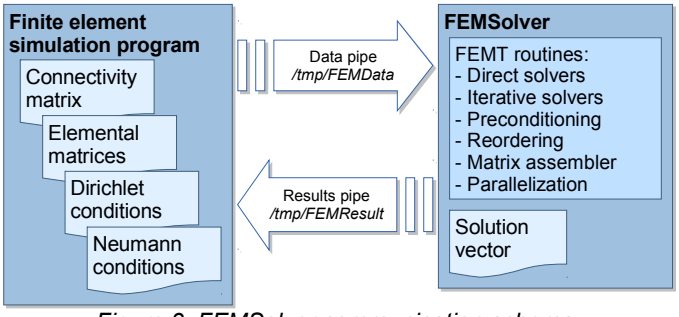


Figure 3. FEMSolver communication schema.

Before running FEMSolver the user can specify which kind of solver to use, the preconditioner type and the number of threads (cores) to use for parallelization.

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

In multi-step problems, where the matrix remains constant a direct solver can be used, FEMSolver can be used to efficiently solve problems like linear dynamic deformations, transient heat diffusion, etc.

B. FEMSolver.Schur

FEMSolver.Schur 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.

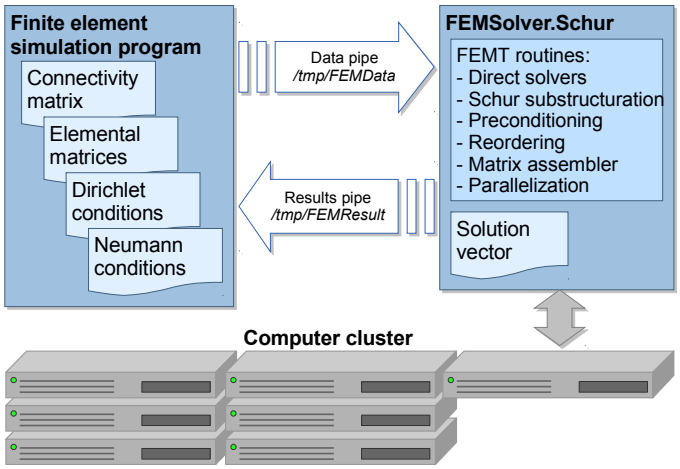


Figure 4. FEMSolver.Schur communication schema.

This tool uses the Schur substructuring method to divide the solution of the system of equations into several small parts. It uses the MPI technology to handle communication between nodes in the cluster. This makes high performance computing (HPC) easy to use to solve very large systems of equations. Below in the document we will present some numerical experiments with matrices with more than one hundred million equations solved in a mid-size cluster.

C. 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.

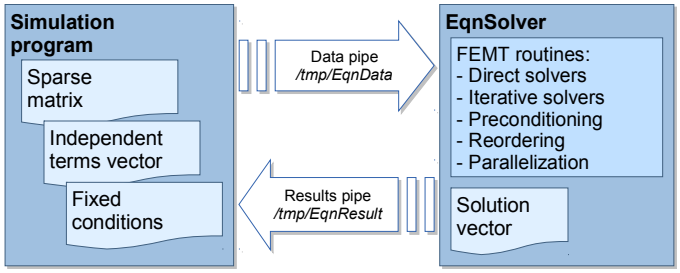


Figure 5. EqnSolver communication schema.

D. EqnSolver.Schur

It uses the same input data as EqnSolver, but uses a cluster of computers parallelize the solution of the system of equations, it uses the Schur substructuring method to do so.

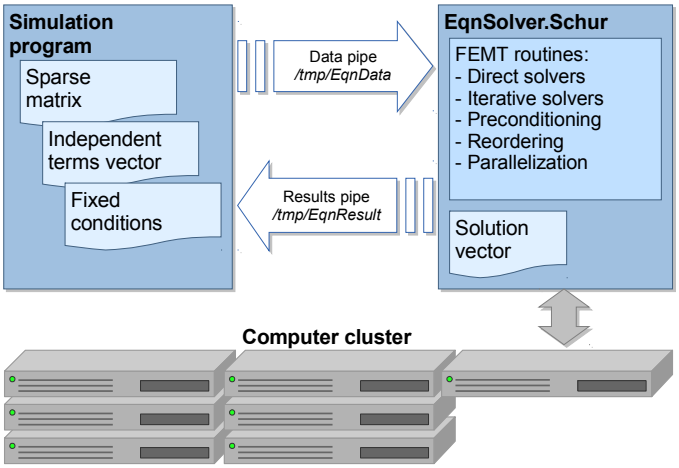


Figure 6. EqnSolver.Schur communication schema.

E. MatSolver

Another simple way to access the FEMT library 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.

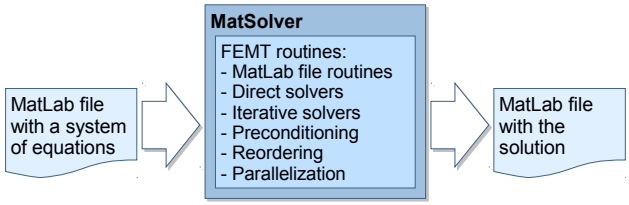


Figure 7. MatSolver, it uses MatLab files as input and output formats.

IV. Sparse matrices

In problems simulated 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.

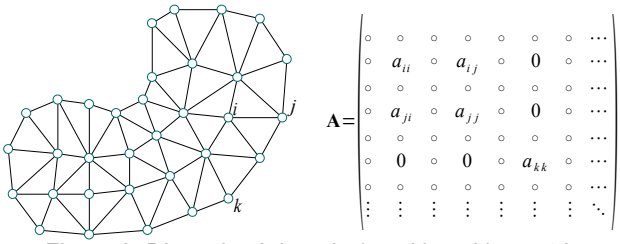


Figure 8. Discretized domain (mesh) and its matrix representation.

Lets define the notation $\eta(\boldsymbol{A})$, it indicates the number of non-zero entries of $\boldsymbol{A}$. For example figure 9, $\boldsymbol{A}\in\mathbb{R}^{556\times 556}$ has 309,136 entries, with $\eta(\boldsymbol{A})=1810$, this means that only the 0.58% of the entries are non zero.

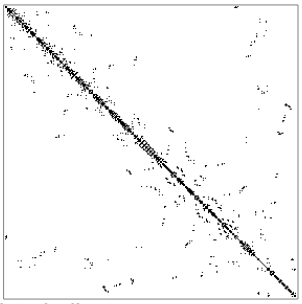


Figure 9. Black dots indicates a non zero entries in the matrix.

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

A. Matrix storage

An efficient method to store and operate matrices of this kind of problems is the Compressed Row Storage (CRS) [5]. 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. An simple example for this storage method is shown in figure 10.

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

8	4	
1	2	
1	3	
3	4	
2	1	7
1	3	5
9	3	1
2	3	6
5		
6		

$\boldsymbol{v}_4^{\boldsymbol{A}}=(9,3,1)$

$\boldsymbol{j}_4^{\boldsymbol{A}}=(2,3,6)$

Figure 10. Example for compressed row storage method.

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

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

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 [2].

V. 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

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

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

A. 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 $\boldsymbol{L}$.

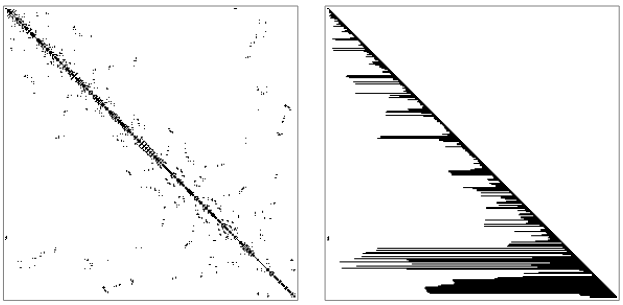


Figure 11. Left: non-zero entries of $\boldsymbol{A}$. Right: non-zero entries of $\boldsymbol{L}$ (Cholesky factorization of $\boldsymbol{A}$)

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.

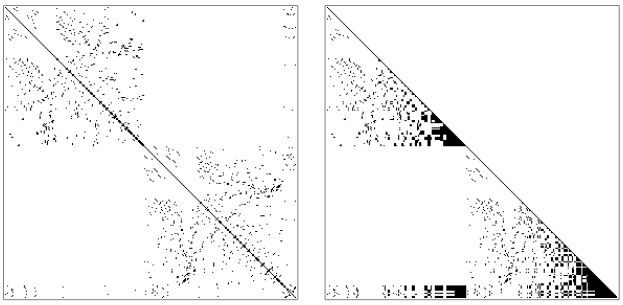


Figure 12. Left: non-zero entries of reordered $\boldsymbol{A}$. Right: non-zero entries of $\boldsymbol{L}$.

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. Calculating the optimum ordering that minimizes the number the fill-in is an NP-complete problem [6], but there are heuristics that generate an acceptable ordering in a reduced time. The most common reordering heuristic to reduce fill-in is the minimum degree algorithm, the basic version is presented in [7], more advanced versions can be found in [8].

There are more complex algorithms that perform better in terms of time and memory requirements, the nested dissection algorithm developed by Karypis and Kumar [9] included in METIS library gives very good results.

B. Symbolic Cholesky factorization

This algorithm identifies non zero entries of $\boldsymbol{L}$, a deep explanation could be found in [10].

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

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

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

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

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:

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

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

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

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

Figure 13. 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 [7].

C. Filling entries in parallel

Once non zero entries are determined we can rewrite (1) and (2) as

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

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

The resulting algorithm to fill non zero entries is [11]:

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 [12]. Calculus of each column is divided among cores.

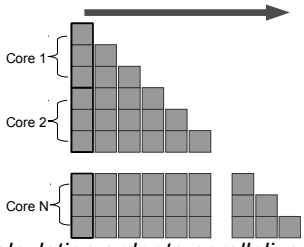


Figure 14. Calculation order to parallelize the Cholesky factorization.

Cholesky solver is particularly efficient because the stiffness matrix is factorized once.

D. LDL' factorization

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

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

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

Using sparse matrices, we can use the following

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

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

E. Numerical experiments

The next charts and table shows results solving a 2D Poisson equation problem, comparing Cholesky and conjugate gradient with Jacobi preconditioning. Several discretizations where used, from 1,000 equations up to 10,000,000 equations.

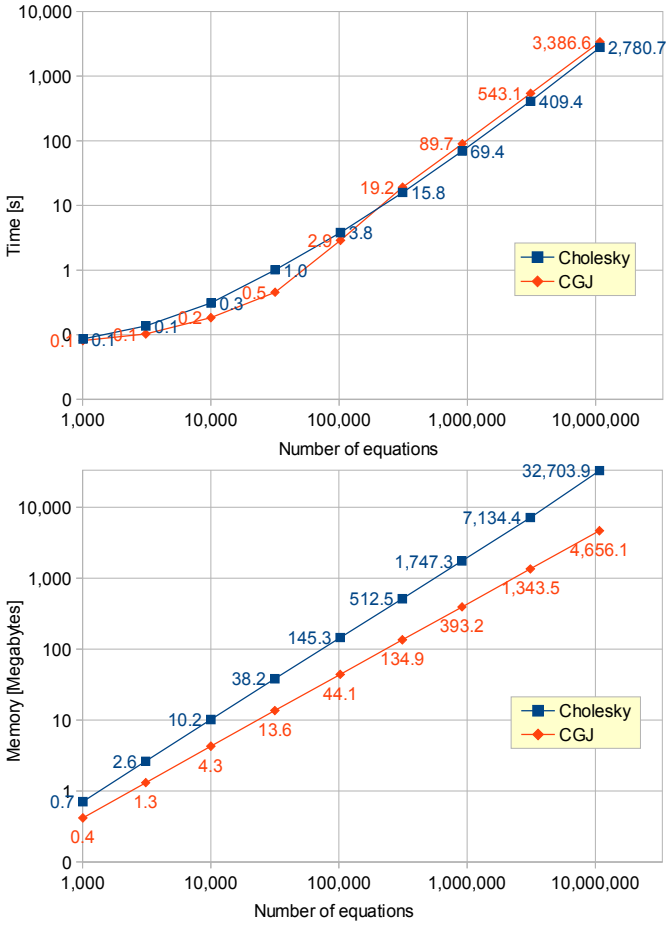


Figure 15. Number of equations vs. time (top) and number of equations vs. memory (bottom).

The tests were run in a computer with 8 Intel Xeon E5620 cores running at 2.40GHz and with 32GB of memory.

Equations	nnz(A)	nnz(L)	Cholesky time [s]	CGJ time [s]
1,006	6,140	14,722	0.09	0.08
3,110	20,112	62,363	0.14	0.10
10,014	67,052	265,566	0.31	0.18
31,615	215,807	1'059,714	1.01	0.45
102,233	705,689	4'162,084	3.81	2.89
312,248	2'168,286	14'697,188	15.82	19.17
909,540	6'336,942	48'748,327	69.35	89.66
3'105,275	21'681,667	188'982,798	409.37	543.11
10'757,887	75'202,303	743'643,820	2780.73	3386.61

VI. 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

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

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

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

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.

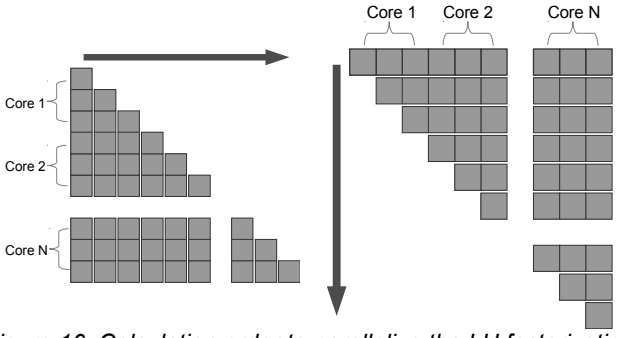


Figure 16. Calculation order to parallelize the LU factorization.

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 [11]:

```
for j ← 1…n
· U_jj ← A_jj
· For q ← 1…(|V_j(L)|−1)
· · U_jj ← U_jj − V_j^q(L) V_j^q(U)
· L_jj ← 1
· U_jj^T ← U_jj
· parallel for q ← 2…|J_j(L^T)|
· · i ← J_j^q(L^T)
· · L_ij ← A_ij
· · U_ji^T ← A_ji
· · r ← 1; ρ ← J_i^r(L)
· · s ← 1; σ ← J_j^s(L)
· · repeat
· · · while ρ<σ
· · · · r ← r+1; ρ ← J_i^r(L)
· · · while ρ>σ
· · · · s ← s+1; σ ← J_j^s(L)
· · · while ρ=σ
· · · · if ρ=j
· · · · · exit repeat loop
· · · · L_ij ← L_ij − V_i^r(L) V_j^s(U^T)
· · · · U_ji^T ← U_ji^T − V_j^s(L) V_i^r(U^T)
```

$r \leftarrow r+1$
$\rho \leftarrow \boldsymbol{J}_i^r(\boldsymbol{L})$
$s \leftarrow s+1$
$\sigma \leftarrow \boldsymbol{J}_j^s(\boldsymbol{L})$
$L_{ij} \leftarrow \frac{L_{ij}}{U_{jj}}$
$L_{ji}^{\mathrm{T}} \leftarrow L_{ij}$
$U_{ji} \leftarrow U_{ij}^{\mathrm{T}}$

VII. 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.

A. Preconditioning

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 $\|\boldsymbol{r}_k\|>\varepsilon$
$\alpha_k \leftarrow -\frac{\boldsymbol{r}_k^{\mathrm{T}}\boldsymbol{q}_k}{\boldsymbol{p}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_k}$
$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k$
$\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k-\alpha_k\boldsymbol{A}\boldsymbol{p}_k$
$\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{M}\boldsymbol{r}_{k+1}$
$\beta_k \leftarrow \frac{\boldsymbol{r}_{k+1}^{\mathrm{T}}\boldsymbol{q}_{k+1}}{\boldsymbol{r}_k^{\mathrm{T}}\boldsymbol{q}_k}$
$\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{q}_{k+1}+\beta_k\boldsymbol{p}_k$
$k \leftarrow k+1$

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,

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

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 [13].

B. Jacobi preconditioner

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

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

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

C. Incomplete Cholesky factorization preconditioner

This preconditioner has the form

$$\mathbf{M}=\mathbf{H}_k\mathbf{D}\mathbf{H}_k^{\mathrm{T}},$$

where $\mathbf{H}_k$ is a lower triangular sparse matrix that have structure similar to the Cholesky factorization of $\mathbf{A}$. he structure of $\mathbf{H}_0$ is equal to the structure of the lower triangular form of $\mathbf{A}$. For $0<k<n$, k diagonals will be added to the preconditioner using the symbolic factorization, for $k=n$, the structure of $\mathbf{H}_k$ will be equal to the structure of $\mathbf{L}$ (Cholesky factorization of $\mathbf{A}$). The values of $\mathbf{H}_k$ will be filled using the formulas for the Cholesky factorization,

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

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

This preconditioner could not be SPD [14], to avoid this problem the algorithm of Munksgaard [15] is used, it consists of two strategies:

- Add a perturbation to the diagonal of $\mathbf{A}$ with a factor α,

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

  this will make the preconditioner to be SPD. The value of α can be found by try and error.
- Create a perturbation in the pivots to increase stability if they are negative or near zero,

  if $D_{jj}\leq u\left(\sum_{k\neq j}|a_{jk}|\right)$, then

$$D_{jj}=\begin{cases}\sum_{k\neq j}|a_{jk}| & \text{if } \sum_{k\neq j}|a_{jk}|\neq 0\\ 1 & \text{if } \sum_{k\neq j}|a_{jk}|=0\end{cases}.$$

An adequate value for u is 0.01.

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 $\mathbf{H}_k$. Unfortunately the dependency of values makes these substitutions very hard to parallelize.

D. 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.

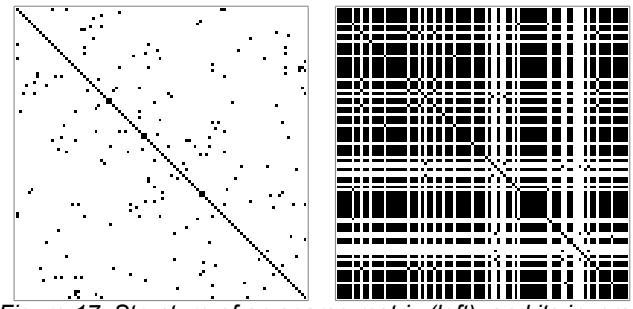


Figure 17. Structure of an sparse matrix (left), and its inverse (right).

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

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

The Frobenius norm is defined as

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

It is possible to separate (3) into decoupled sums of 2-norms for each column [16],

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

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 [17] 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

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

$\boldsymbol{S}_{\boldsymbol{L}}$ 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 $\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{A}}^2, \ldots, \tilde{\boldsymbol{A}}^l$,

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

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

the threshold t 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 $\tilde{\boldsymbol{A}}^l_{k,:}$,

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

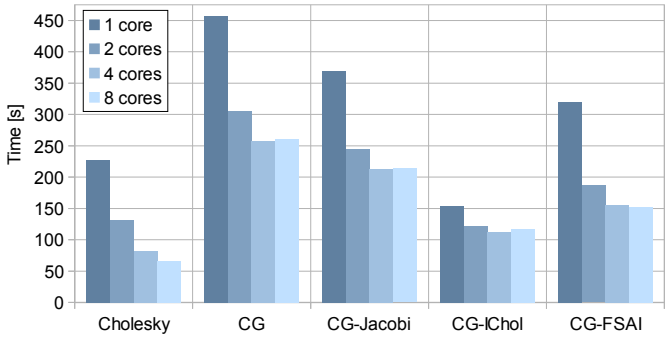
The structure $\boldsymbol{S}_l$ will be the lower triangular part of $\tilde{\boldsymbol{A}}^l$. With this truncated $\tilde{\boldsymbol{A}}^l$, a $\tilde{\boldsymbol{G}}^l$ is calculated using the previous algorithm to create a preconditioner $\boldsymbol{M}=\tilde{\boldsymbol{G}}_l^{\mathrm{T}}\tilde{\boldsymbol{G}}_l$.

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

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

E. Numerical experiments

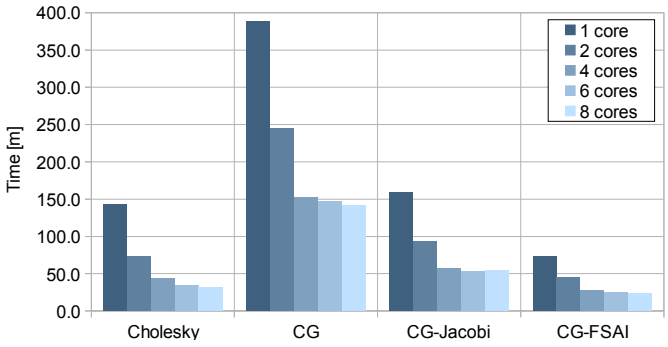
First we will show results for the parallelization of solvers with OpenMP. The next example is a 2D solid deformation with 501,264 elements, 502,681 nodes. A system of equations with 1'005.362 variables is formed, the number of non zero entries are $\eta(\boldsymbol{K})=18'062{,}500$, $\eta(\boldsymbol{L})=111'873{,}237$.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



Solver	1 core [s]	2 cores [s]	4 cores [s]	8 cores [s]	Steps	Memory [bytes]
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 [m]	2 cores [m]	4 cores [m]	6 cores [m]	8 cores [m]	Memory [bytes]
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.

VIII. 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-directions of descent $\tilde{\boldsymbol{p}}_k$ to be $\boldsymbol{A}$-orthogonal to the descent directions $\boldsymbol{p}_k$ [18].

If the matrix $\boldsymbol{A}$ is symmetric, 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 [18]:

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

This method can also be preconditioned.

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

Preconditioners for this solver are Jacobi, incomplete LU factorization [19] and factorized sparse approximate inverse for non-symmetric matrices [20]. These are counterparts of the preconditioners for the symmetric case.

IX. Schur sustructuring method

This is a domain decomposition method with no overlapping [21], the basic idea is to split a large system of equations into smaller systems that can be solved independently in different computers in parallel.

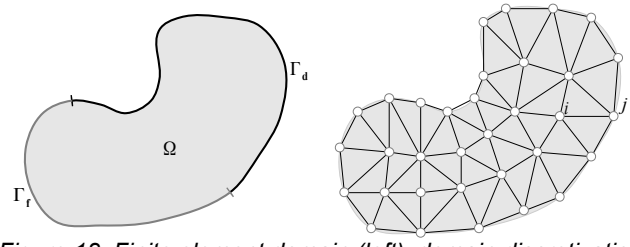


Figure 18. Finite element domain (left), domain discretization (center), partitioning (right).

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 symmetric positive definite matrix of size $n\times n$.

A. 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 [22].

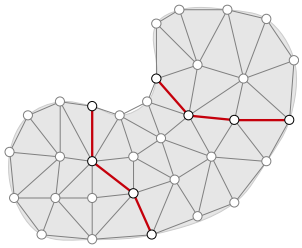


Figure 19. Partitioning example.

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

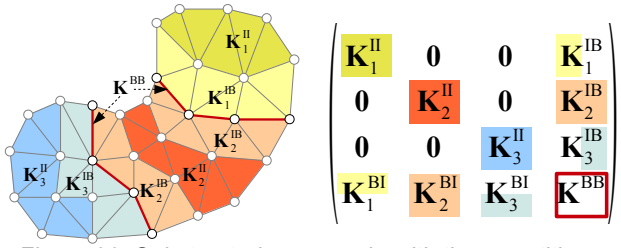


Figure 20. Substructuring example with three partitions.

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 and are used for entries with values dependent of nodes in the boundary and nodes inside the partition.

On a more general example

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

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 partitioning the mesh we used the METIS library [9].

B. Schur complement method

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

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

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 [23], we proceed column by column using an extra vector $\mathbf{t}$, and solving for $c=1\ldots n$

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

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 increase 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.

C. Numerical experiments

We present 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.

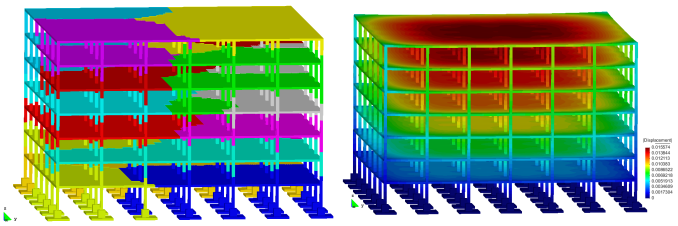
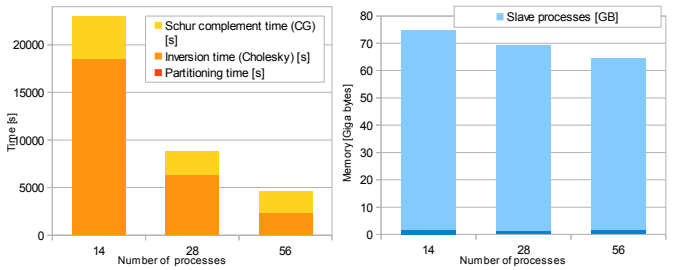


Figure 21. Substructuration of the domain (left) ad the resulting deformation (right)

Number of processes	Partition time [s]	Inversion time [s]	Schur c. time [s]	CG steps	Total time [s]
14	47.6	18521	4445	6927	23025
28	45.7	6270	2445	8119	8772
56	44.1	2257	2296	9627	4609



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

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

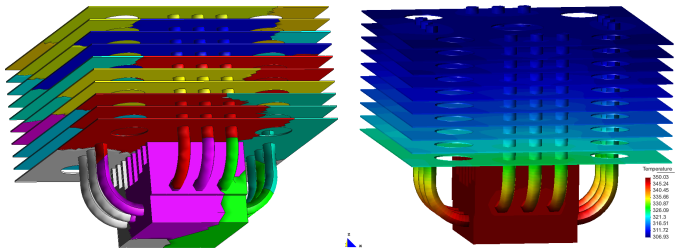
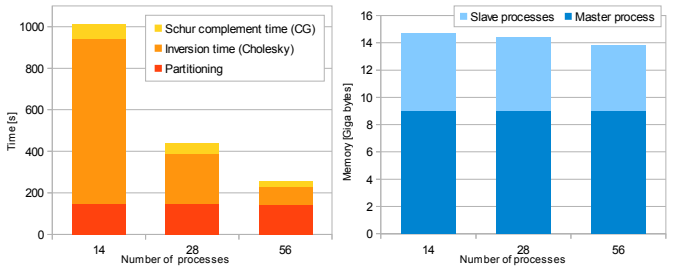


Figure 22. Sub-structuration of the domain (left) ad the resulting temperature (right)

Number of processes	Partition time [s]	Inversion time [s]	Schur c. time [s]	CG steps	Total time [s]
14	144.9	798.5	68.1	307	1020.5
28	146.6	242.0	52.1	348	467.1
56	144.2	82.8	27.6	391	264.0



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

D. Larger systems of equations

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

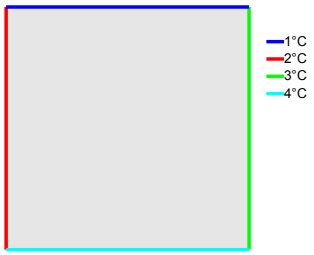


Figure 23. Domain and boundary conditions.

The domain was discretized using quadrilaterals with nine nodes, the discretization made was from 25 million nodes up to 800 million nodes. In all cases we divided the domain into 960 partitions. In this case we used a cluster with 81 nodes, each one with two CPU E5-2620 processors (with 6 cores per processor), a total of 972 cores. A node is used as a master process that loads the geometry and the problem parameters and splits the systems of equations. Tolerance used was $1x10^{-10}$.

Number of equations	Time [hours]	Memory [GB]
25,010,001	00:10:00	38,127,911,928
50,027,329	00:22:21	82,291,561,272
75,012,921	00:34:24	128,852,868,872
100,020,001	00:46:14	176,982,703,608
125,014,761	00:59:20	224,876,901,752
150,038,001	01:10:32	275,868,154,968
200,081,025	01:37:42	380,487,437,704
250,050,969	02:03:05	485,244,957,896
300,086,329	02:29:11	598,995,145,840
400,040,001	03:24:50	812,439,074,088
500,103,769	04:26:51	1,034,046,442,776
600,103,009	05:16:07	1,263,423,250,648
700,078,681	06:15:05	1,451,719,027,176
800,154,369	07:18:15	1,690,025,398,632

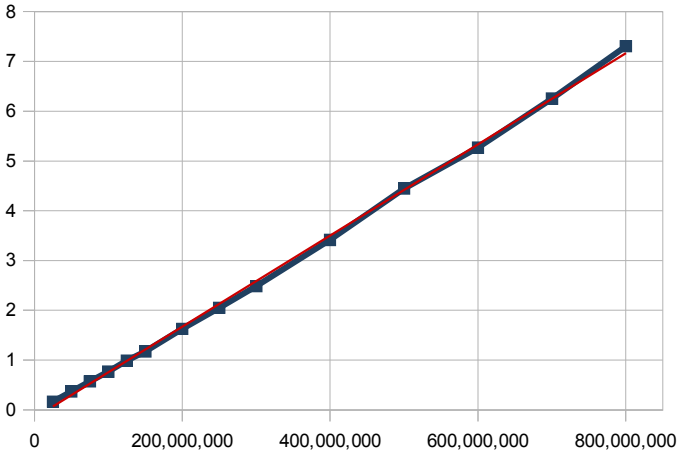


Figure 24. Number of equations vs. solution time (in hours).

E. Speed-up

To test the speed-up of the Schur substructuring method we used the problem defined in section D with 25,010,001 equa-

tions, using from 4 to 80 parallel processes. Results are shown in the next table, and in figure 25.

Processes	Time [min]	Memory [bytes]
4	300.2	76,957,081,832
8	165.6	67,884,179,856
16	69.4	59,350,255,936
24	41.3	55,276,709,976
32	30.1	53,020,499,256
40	24.1	51,385,832,424
48	21.3	49,783,503,696
56	18.0	48,852,531,880
64	17.5	48,360,392,536
72	15.4	47,395,019,288
80	14.9	47,303,011,304

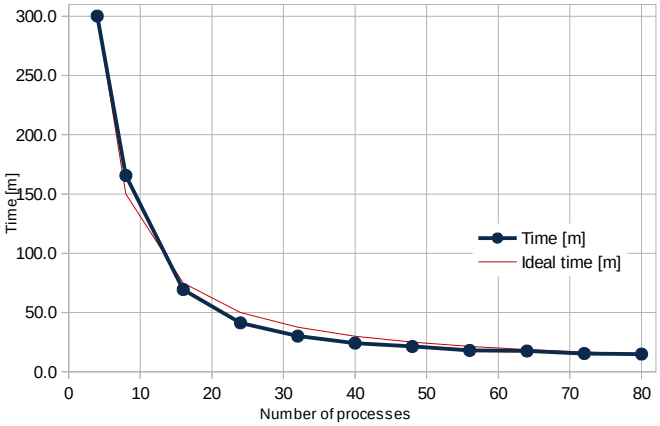


Figure 25.Number of processes vs solution time in minutes.

The Schur substructuring method can not be used with a single partition, thus, to plot the speed-up chart in figure 26 we defined the time taken to solve the system of equations with four processes/partitions as the serial time (t_4), with this

$$\text{speed-up} = \frac{t_4}{t_n}.$$

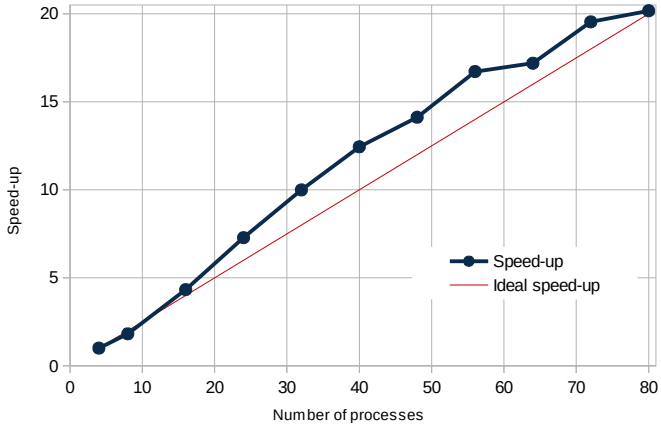


Figure 26.Schur substructuring method speed-up.

The resulting speed-up appears to be better than the ideal, the explanation for this behavior is that with less partitions each parallel process has to handle more memory, and memory access is not linear, many bottlenecks exists, like “page faults”, “cache misses”, “collisions on the bus” on NUMA systems, and others [2].

X. Conclusions and Future Work

We have presented a set of tools for solving large linear systems of equations that is easily adaptable to an existing simulation code in any programming language. This is important, because for large simulation codes, it takes a lot of time and is expensive to translate all code to use a new technology that uses a different storage method. All solvers are designed to run efficiently on multi-core computers.

In the future we will include more solvers, like biconjugate gradient stabilized method, GMRES, and implement more preconditioners. Because the software is released as open source we welcome collaboration from the scientific community to make it better.

XI. FEMT license

XII. References

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

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

[3] 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.

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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

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

[13] 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.

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

[15] N. Munksgaard. Solving Sparse Symmetric Sets of Linear Equations by Preconditioned Conjugate Gradients. ACM Transactions on Mathematical Software, Vol 6-2, pp. 206-219. 1980.
[16] E. Chow, Y. Saad. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. SIAM Journal on Scientific Computing. Vol. 19-3, pp. 995-1023. 1998.
[17] 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.
[18] U. Meier-Yang. Preconditioned conjugate gradient-like methods for nonsymmetric linear systems. University of Illinois. 1994.
[19] V. Eijkhout. On the Existence Problem of Incomplete Factorisation Methods. Lapack Working Note 144, UT-CS-99-435. 1999.
[20] A. Y. Yeremin, A. A. Nikishin. Factorized-Sparse-Approximate-Inverse Preconditionings of Linear Systems with Unsymmetric Matrices. Journal of Mathematical Sciences, Vol. 121-4. 2004.
[21] J. Kruis. “Domain Decomposition Methods on Parallel Computers”. Progress in Engineering Computational Technology, pp 299-322. Saxe-Coburg Publications. Stirling, Scotland, UK. 2004.
[22] M. Vargas-Felix, S. Botello-Rionda. Solution of finite element problems using hybrid parallelization with MPI and OpenMP. Acta Universitaria. Vol. 22-7, pp 14-24. 2012.
[23] 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://hdl.handle.net/10803/6678