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 kinds: direct, iterative and domain decomposition. Direct and iterative solvers are designed to run in parallel in multi-core computers. 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 one hundred million degrees of freedom.


I. INTRODUCTION

FEMT aims to help people that use finite element, finite differences or isogeometric analysis to easily incorporate a solver for the sparse systems generated. The FEMT package is divided into 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 analysis 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

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.](image)

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 processors have several caches that are organized by levels (L1, L2, etc), L1 cache is next to the core. It is important to consider 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:

<table>
<thead>
<tr>
<th>Access to</th>
<th>CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU registers</td>
<td>&lt;=1</td>
</tr>
<tr>
<td>L1 cache</td>
<td>3</td>
</tr>
<tr>
<td>L2 cache</td>
<td>14</td>
</tr>
<tr>
<td>RAM</td>
<td>240</td>
</tr>
</tbody>
</table>

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 exist 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 ascending 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.](image)

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.](image)

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.

```
Finite element simulation program
- Connectivity matrix
- Elemental matrices
- Dirichlet conditions
- Neumann conditions
Figure 4. FEMSolver.Schur communication schema.

Computer cluster

FEMSolver.Schur
- FEMT routines:
  - Direct solvers
  - Schur substructuring
  - Preconditioning
  - Reordering
  - Matrix assembler
  - Parallelization

Results pipe
/tmp/FEMResult

Simulation program
- Sparse matrix
- Independent terms vector
- Fixed conditions

Data pipe
/tmp/FEMData

Equation vector
/tmp/EqnResult

Solution vector
/tmp/EqnData

MatSolver
- MatLab routines:
  - Direct solvers
  - Iterative solvers
  - Preconditioning
  - Reordering
  - Paralellization

Data pipe
/tmp/FEMResult

Results pipe
/tmp/EqnResult

MatSolver.Schur
- FEMT routines:
  - Direct solvers
  - Iterative solvers
  - Preconditioning
  - Reordering
  - Parallelization

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.

```

IV. Sparse matrices

In problems simulated with finite element or finite volume methods it is common to have to solve linear system of equations \( Ax = b \).

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.

```
\[
A = \begin{pmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & a_{ij} & a_{ij} & 0 & \cdots \\
\vdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\vdots & \cdots & \cdots & \cdots & a_{jj} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{pmatrix}
\]
```

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

Lets define the notation \( \eta(A) \), it indicates the number of non-zero entries of \( A \). For example figure 9, \( A \in \mathbb{R}^{556 \times 556} \) has 309,136 entries, with \( \eta(A) = 1810 \), this means that only the 0.58% of the entries are non zero.
Cholesky factorization could be implemented in parallel. To maintain memory and time requirements near to those of symbolic factorization, we could use reordering strategies and store factor matrices using CRS, identifying non-zero entries using vectors \( v^A_i \) with their respective column indexes. An example for this storage method is shown in figure 10.

\[
A = \begin{bmatrix}
8 & 4 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 3 & 0 & 0 \\
2 & 0 & 1 & 0 & 7 & 0 \\
0 & 9 & 3 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 5 \\
\end{bmatrix}, \quad v^A_4 = (9, 3, 1), \quad j^A_4 = (2, 3, 6)
\]

Figure 10. Example for compressed row storage method.

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

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

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 an efficient processor cache usage [2].

V. CHOLESKY FACTORIZATION FOR SPARSE MATRICES

The cost of using Cholesky factorization \( A=LL^T \) 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.

Formulæ to calculate \( L \) entries are

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

\[
L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{i-1} L_{jk}^2}.
\]

A. 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 \( A \) and the non zero entries of its Cholesky factorization \( L \).

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

By reordering we have a new factorization with \( \eta(L)=3215 \), 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 \( L \), a deep explanation could be found in [10].
For an sparse matrix $A$, we define
\[ a_j \triangleq \{ k > j \mid A_{kj} \neq 0 \}, \quad j = 1 \ldots n, \]
as the set of non zero entries of column $j$ of the strictly lower triangular part of $A$.

In similar way, for matrix $L$ we define the set
\[ l_j \triangleq \{ k > j \mid L_{kj} \neq 0 \}, \quad j = 1 \ldots n. \]

We also use sets define sets $r_j$ that will contain columns of $L$ which structure will affect the column $j$ of $L$. The algorithm is:

\[
\begin{align*}
  r_j &\leftarrow \emptyset, \quad j \leftarrow 1 \ldots n \\
  r_j &\leftarrow A_j, \quad \text{for } j \leftarrow 1 \ldots n \\
  l_j &\leftarrow a_j, \quad \text{for } j \leftarrow 1 \ldots n \\
  \text{for } i \leftarrow r_j \\
  \quad l_j &\leftarrow l_j \cup l_i \setminus \{ j \} \\
  p &\leftarrow \min\{ j \in l_i \} \quad \text{if } l_i \neq \emptyset \\
  \quad r_p &\leftarrow r_p \cup \{ j \} \\
  \text{other case }
\end{align*}
\]

For the next example matrix column 2, $a_2$ and $l_2$ will be:

\[
\begin{align*}
A &= \begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
  a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46}
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
L &= \begin{pmatrix}
  l_{11} & l_{12} & l_{13} & l_{14} & l_{15} & l_{16} \\
  l_{21} & l_{22} & l_{23} & l_{24} & l_{25} & l_{26} \\
  l_{31} & l_{32} & l_{33} & l_{34} & l_{35} & l_{36} \\
  l_{41} & l_{42} & l_{43} & l_{44} & l_{45} & l_{46}
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
1_j &= \{ 3, 4 \}
\end{align*}
\]

Figure 13. Example matrix, showing how $a_2$ and $l_2$ are formed.

This algorithm is very efficient, time and memory usage has an order of $O(\eta(L))$. 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

\[
\begin{align*}
L_{ij} &= \frac{1}{L_{jj}} \left( A_{ij} - \sum_{k \in \eta(l_i^j)} L_{ik} L_{jk} \right), \quad \text{for } i > j; \\
L_{ij} &= \sqrt{A_{jj} - \sum_{k \in \eta(l_i^j)} L_{ik}^2}, \quad \text{for } i \leq j.
\end{align*}
\]

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

\[
\begin{align*}
\text{for } j &\leftarrow 1 \ldots n \\
\quad L_{jj} &\leftarrow A_{jj} \\
\quad \text{for } q &\leftarrow 1 \ldots |l_i^j| \\
\quad \quad L_{jj} &\leftarrow L_{jj} - (v_i^j) q (v_i^j)_q \\
\quad L_{ij} &\leftarrow \sqrt{L_{jj}} \\
\quad L_{i}^T &\leftarrow L_{ij} \\
\quad \text{parallel for } q &\leftarrow 1 \ldots |l_i^j| \\
\quad \quad i &\leftarrow \{ l_{i}^j \} q \\
\quad L_{ij} &\leftarrow A_{ij} \\
\quad r &\leftarrow 1; \rho &\leftarrow \{ j^T \} r \\
\quad s &\leftarrow 1; \sigma &\leftarrow \{ j^T \} s \\
\quad \text{repeat }
\end{align*}
\]

while $\rho < \sigma$

\[
\begin{align*}
\quad r &\leftarrow r + 1; \rho &\leftarrow \{ j^T \} r \\
\quad \text{while } \rho \geq \sigma \\
\quad s &\leftarrow s + 1; \sigma &\leftarrow \{ j^T \} s \\
\quad \text{if } \rho = \sigma \\
\quad \text{exit repeat } \\
\quad L_{ij} &\leftarrow L_{ij} - (v_i^j) q (v_i^j)_q \\
\quad r &\leftarrow r + 1; \rho &\leftarrow \{ j^T \} r \\
\quad s &\leftarrow s + 1; \sigma &\leftarrow \{ j^T \} s \\
\quad \text{for } k \in \eta(l_i^j) \\
\quad \quad L_{ij} &\leftarrow L_{ij} - \sum_{\kappa \in \eta(l_i^j) \cap \eta(l_k^j)} L_{ik} L_{jk} D_{\kappa}, \quad \text{for } i > j \\
\quad D_j &\leftarrow A_{jj} - \sum_{k \in \eta(l_i^j)} L_{ik}^2 D_{\kappa}.
\end{align*}
\]

Using sparse matrices, we can use the following

\[
\begin{align*}
\text{for } i \geq j \\
\quad D_j &\leftarrow A_{jj} - \sum_{k \in \eta(l_i^j) \cap \eta(l_k^j)} L_{ik} L_{jk} D_{\kappa}.
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Equations</th>
<th>nnz(A)</th>
<th>nnz(L)</th>
<th>Cholesky time (s)</th>
<th>CGJ time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,006</td>
<td>6,140</td>
<td>14,722</td>
<td>0.09</td>
<td>0.08</td>
</tr>
<tr>
<td>3,110</td>
<td>20,112</td>
<td>62,363</td>
<td>0.14</td>
<td>0.10</td>
</tr>
<tr>
<td>10,014</td>
<td>67,052</td>
<td>265,566</td>
<td>0.31</td>
<td>0.18</td>
</tr>
<tr>
<td>31,615</td>
<td>215,807</td>
<td>1'059,714</td>
<td>1.01</td>
<td>0.45</td>
</tr>
<tr>
<td>102,233</td>
<td>705,689</td>
<td>4'162,084</td>
<td>3.81</td>
<td>2.89</td>
</tr>
<tr>
<td>312,248</td>
<td>2'168,286</td>
<td>14'697,188</td>
<td>15.82</td>
<td>19.17</td>
</tr>
<tr>
<td>909,540</td>
<td>6'336,942</td>
<td>48'748,327</td>
<td>69.35</td>
<td>89.66</td>
</tr>
<tr>
<td>3'105,275</td>
<td>21'681,667</td>
<td>188'962,798</td>
<td>409.37</td>
<td>543.11</td>
</tr>
<tr>
<td>10'757,887</td>
<td>75'202,303</td>
<td>743'643,820</td>
<td>2780.73</td>
<td>3386.61</td>
</tr>
</tbody>
</table>

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 L and U^T will have the same structure.

Formulae to calculate L and U (using Doolittle’s algorithm) are:

\[ L_{ij} = \frac{1}{U_{ii}} \left( A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj} \right) \text{ for } i > j, \]

\[ U_{ii} = A_{ii} - \sum_{k=1}^{i-1} L_{ik} U_{ik}, \quad L_{ii} = 1. \]

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

\[ L_{ij} = \frac{1}{U_{ii}} \left( A_{ij} - \sum_{k \in \{ j \}} L_{ik} U_{kj} \right) \text{ for } i > j, \]

\[ U_{ii} = A_{ii} - \sum_{k \in \{ j \}} L_{ik} U_{ik}, \quad L_{ii} = 1. \]

To parallelize the algorithm, the fill of L must be done row by row, each row filled in parallel. The sequence to fill L and U in parallel is shown in the following figures.

![Figure 15. Number of equations vs. time (top) and number of equations vs. memory (bottom).](image)

![Figure 16. Calculation order to parallelize the LU factorization.](image)

Similarity to the Cholesky algorithm, to improve performance we will store L, U and U^T matrices using CRS. It is shown in the next algorithm [11]:

```
for j ∈ 1...n
    U_{jj} ← A_{jj}
    L_{jj} ← 0

    for q ∈ 1...(|J_j(L)|−1)
        U_{jj} ← U_{jj}−V_j'(L)V_j(L)'(U)
        L_{jj} ← 1
        U_{jj} ← U_{jj}−V_j'(L)V_j(L)'(U)

    for i ∈ 2...|J_j(L)^T|
        L_{ij} ← A_{ij}
        U_{ij} ← A_{ij}
        r ← 1; p ← J_j'(L)
        s ← 1; σ ← J_j'(L)
        repeat
            while r < σ
                r ← r + 1; p ← J_j'(L)
            while p < σ
                s ← s + 1; σ ← J_j'(L)
            if r == p
                exit repeat loop
            L_{ij} ← L_{ij}−V_j'(L)V_j'(U)^T
            U_{ij} ← U_{ij}−V_j'(L)V_j'(U)^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 \( \kappa \) of a non-singular matrix \( A \in \mathbb{R}^{n \times n} \), given a norm \( \| \cdot \| \), is defined as

\[
\kappa(A) = \|A\| \|A^{-1}\|.
\]

For the norm \( \| \cdot \|_2 \),

\[
\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)},
\]

where \( \sigma \) is a singular value of \( A \).

For a SPD matrix,

\[
\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)},
\]

where \( \lambda \) is an eigenvalue of \( A \).

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

A preconditioner for a matrix \( A \) is another matrix \( M \) such that \( MA \) has a lower condition number

\[
\kappa(MA) < \kappa(A).
\]

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

\[
Ax - b = 0,
\]

with preconditioning we solve

\[
M(Ax - b) = 0.
\]

The preconditioned conjugate gradient algorithm is:

- \( r_0 \leftarrow A \cdot x_0 \), initial approximation
- \( \rho \leftarrow J^T( L) \), initial gradient
- \( s \leftarrow s + 1 \)
- \( \alpha \leftarrow J^T( L) \), initial descent direction
- \( k \leftarrow 0 \)

while \( \| r_0 \| > \epsilon \)

- \( \alpha_k \leftarrow \frac{r_0^T q_k}{p_k^T A p_k} \)
- \( x_{k+1} \leftarrow x_k + \alpha_k p_k \)
- \( r_{k+1} \leftarrow r_{k} - \alpha_k A p_k \)
- \( q_{k+1} \leftarrow M r_{k+1} \)

- \( \beta_k \leftarrow r_{k+1}^T q_{k+1} \)
- \( r_{k+1} \leftarrow r_{k+1} - \beta_k p_k \)
- \( k \leftarrow k + 1 \)

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

We used the Jacobi preconditioner, it is suitable for sparse systems with SPD matrices. The diagonal part of \( M^{-1} \) is stored as a vector, \( M^{-1} = (\text{diag}(A))^{-1} \).

Parallelization of this algorithm is straightforward, because the calculus of each entry of \( q_k \) 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 \( M^{-1} \) is stored as a vector,

\[
(M^{-1})_{i,j} = \begin{cases} \frac{1}{A_{ii}} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}.
\]

Parallelization of this algorithm is straightforward, because the calculus of each entry of \( q_k \) is independent.

C. Incomplete Cholesky factorization preconditioner

This preconditioner has the form

\[
M = H_k \cdot D \cdot H_k^T,
\]

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

\[
H_{ij} = D_{ij} \left( A_{ij} - \sum_{k=1}^{j-1} H_{ik} H_{jk} D_{k} \right), \quad \text{for } i > j;
\]

\[
D_{ij} = A_{ij} - \sum_{k=1}^{i-1} H_{ik} D_{k}.
\]

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 \( A \) with a factor \( \alpha \),

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

this will make the preconditioner to be SPD. The value of \( \alpha \) can be found by try and error.

- Create a perturbation in the pivots to increase stability if they are negative or near zero,

if \( D_{jj} \leq u \left( \sum_{j \neq k} |a_{jk}| \right) \), 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 \( H \). 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 \( M \) to be an approximation of the inverse of \( A \) 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).](image)

A way to create an approximate inverse is to minimize the Frobenius norm of the residual \( I - AM \),

\[ F(M) = \|I - AM\|_F. \tag{3} \]

The Frobenius norm is defined as

\[ \|A\|_F = \left( \sum_{i,j} a_{ij}^2 \right)^{1/2} = \sqrt{\text{tr}(A^	op A)}. \]

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

\[ F(M) = \|I - AM\|_F = \sum_j \|e_j - Am_j\|_2^2, \]

where \( e_j \) is the j-th column of \( I \) and \( m_j \) is the j-th column of \( M \). With this separation we can parallelize the construction of the preconditioner.

The factorized sparse approximate inverse preconditioner [17] creates a preconditioner

\[ M = G_j G_j^\top, \]

where \( G_j \) is a lower triangular matrix such that \( G_j \approx L^{-1}, \)

where \( L \) is the Cholesky factor of \( A \). \( I \) is a positive integer that indicates a level of sparsity of the matrix.

Instead of minimizing (3), we minimize \( \|I - G_j L\|_F^2 \), it is noticeable that this can be done without knowing \( L \), solving the equations

\[ (G_j L L^\top)_j = (L^\top)_j, \quad (i, j) \in S, \]

this is equivalent to

\[ (G_j)_j = (I)_j, \quad (i, j) \in S', \]

\( S \) contains the structure of \( G_j \).

This preconditioner has these features:

- \( M \) is SPD if there are no zeroes in the diagonal of \( G_j \).

The algorithm to construct the preconditioner is parallelizable.

The algorithm to calculate the entries of \( G_j \) is:

Let \( S_j \) be the structure of \( G_j \)

for \( j \leftarrow 1 \ldots n \)

\[ \text{for } \forall (i, j) \in S_j \]

\[ \text{solve } (A G_j)_j = \delta_j \]

Entries of \( G_j \) are calculated by rows. To solve \( (A G_j)_j = \delta_j \) means that, if \( m = \eta((G_j)_j) \) is the number of non zero entries of the column \( j \) of \( G_j \), then we have to solve a small SPD system of size \( m \times m \).

A simple way to define a structure \( S_j \) for \( G_j \) is to simply take the lower triangular part of \( A \).

Another way is to construct \( S_j \) from the structure take from \( \tilde{A}, \tilde{A}', \ldots, \tilde{A}' \),

where \( \tilde{A} \) is a truncated version of \( A \),

\[ \tilde{A}_j = \begin{cases} 1 & \text{if } i = j \text{ or } \|D^{-1/2} AD^{-1/2}\|_2 > t, \\ 0 & \text{other case} \end{cases} \]

the threshold \( t \) is a non negative number and the diagonal matrix \( D \) is

\[ D_j = \begin{cases} |A_j| & \text{if } |A_j| > 0, \\ 1 & \text{other case} \end{cases} \]

Powers \( \tilde{A}' \) can be calculated combining rows of \( \tilde{A} \). Lets denote the \( k \)-th row of \( \tilde{A}' \) as \( \tilde{A}'_k \),

\[ \tilde{A}'_k = \tilde{A}'_{k-1} \tilde{A}. \]

The structure \( S_j \) will be the lower triangular part of \( \tilde{A}' \).

With this truncated \( \tilde{A}' \), a \( \tilde{G}' \) is calculated using the previous algorithm to create a preconditioner \( M = \tilde{G}' \tilde{G}' \).

The vector \( q_k \leftarrow Mr_k \) is calculated with two matrix-vector products,

\[ Mr_k = \tilde{G}' \tilde{G}' (r_k). \]

### 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 \( \eta(K) = 18’062,500, \quad \eta(L) = 111’873,237 \). The tolerance used in CG methods is \( \|r_k\| \geq 1 \times 10^{-5} \).
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, \( \eta(K)=69 \cdot 255,522 \).

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

### VIII. Parallel Biconjugate Gradient

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

\[
Ax = b,
\]

in this case \( A \in \mathbb{R}^{n \times n} \) does not need to be symmetric.

This method requires to calculate a pseudo-gradient \( \tilde{g}_k \) and a pseudo-direction of descent \( \tilde{p}_k \). The algorithm constructs the pseudo-gradients \( \tilde{g}_k \) to be orthogonal to the gradients \( g_k \), similarly, the pseudo-directions of descent \( \tilde{p}_k \) to be \( A \)-orthogonal to the descent directions \( p_k \) [18].

If the matrix \( A \) 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]:

- \( \varepsilon \), tolerance
- \( x_0 \), initial coordinate
- \( g_0 \leftarrow Ax_0 - b \), initial gradient
- \( \tilde{g}_0 \leftarrow g_0 \), initial pseudo-gradient
- \( p_0 \leftarrow -g_0 \), initial descent direction
- \( \tilde{p}_0 \leftarrow p_0 \), initial pseudo-direction of descent
- \( k \leftarrow 0 \)

while \( \|g_k\|>\varepsilon \):

- \( w \leftarrow A\tilde{p}_k \)
- \( \tilde{w} \leftarrow \tilde{p}_k A \)
- \( \alpha_k \leftarrow \frac{\tilde{g}_k^T g_k}{\tilde{p}_k^T w} \)
- \( x_{k+1} \leftarrow x_k + \alpha_k \tilde{p}_k \)
- \( g_{k+1} \leftarrow g_k + \alpha_w \tilde{w} \)
- \( \tilde{g}_{k+1} \leftarrow \tilde{g}_k + \alpha_w \tilde{w} \)
- \( q_{k+1} \leftarrow M^{-1} g_{k+1} \)
- \( \tilde{q}_{k+1} \leftarrow M^{-1} g_{k+1} \)
- \( \beta_k \leftarrow \frac{\tilde{g}_{k+1}^T q_{k+1}}{\tilde{g}_k^T q_k} \)
- \( p_{k+1} \leftarrow -q_{k+1} + \beta_{k+1} \tilde{p}_k \)
- \( \tilde{p}_{k+1} \leftarrow -q_{k+1} + \beta_{k+1} \tilde{p}_k \)
- \( k \leftarrow k+1 \)

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).](image-url)
We start with a system of equations resulting from a finite element problem
\[ \mathbf{K} \mathbf{d} = \mathbf{f}, \]  
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 be handled by a computer in the cluster [22].

\[ \text{Figure 19. Partitioning example.} \]

We can arrange (reorder variables) of the system of equations to have the following form
\[ \begin{bmatrix} \mathbf{K}_1^\text{II} & 0 & 0 \\ 0 & \mathbf{K}_2^\text{II} & 0 \\ 0 & 0 & \mathbf{K}_3^\text{II} \end{bmatrix} \begin{bmatrix} \mathbf{d}_1^\text{I} \\ \mathbf{d}_2^\text{I} \\ \mathbf{d}_3^\text{I} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_3 \end{bmatrix}. \]  

\[ \text{Figure 20. Substructuring example with three partitions.} \]

The superscript \( \text{II} \) denotes entries that capture the relationship between nodes inside a partition. \( \text{BB} \) 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
\[ \begin{bmatrix} \mathbf{K}_1^\text{II} & 0 & 0 & \mathbf{K}_1^\text{IB} \\ 0 & \mathbf{K}_2^\text{II} & 0 & \mathbf{K}_2^\text{IB} \\ 0 & 0 & \mathbf{K}_3^\text{II} & \mathbf{K}_3^\text{IB} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{K}_p^\text{II} & \mathbf{K}_p^\text{IB} & \mathbf{K}_p^\text{IB} & \mathbf{K}_p^\text{IB} \end{bmatrix} \begin{bmatrix} \mathbf{d}_1^\text{I} \\ \mathbf{d}_2^\text{I} \\ \vdots \\ \mathbf{d}_p^\text{I} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{bmatrix}. \]  

\[ \text{(5)} \]

Thus, the system can be separated in \( p \) different systems,
\[ \begin{bmatrix} \mathbf{K}_i^\text{BB} & \mathbf{K}_i^\text{IB} \\ \mathbf{K}_i^\text{IB} & \mathbf{K}_i^\text{BB} \end{bmatrix} \begin{bmatrix} \mathbf{d}_i^\text{I} \\ \mathbf{d}_i^\text{II} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i^\text{I} \\ \mathbf{f}_i^\text{II} \end{bmatrix}, \quad i = 1 \ldots p. \]

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

**B. Schur complement method**

For each partition \( i \) the vector of unknowns \( \mathbf{d}_i^\text{I} \) as
\[ \mathbf{d}_i^\text{I} = (\mathbf{K}_i^\text{II})^{-1}(\mathbf{f}_i - \mathbf{K}_i^\text{IB}\mathbf{d}_i^\text{II}). \]  

\[ \text{(6)} \]

After applying Gaussian elimination by blocks on (5), the reduced system of equations becomes
\[ \left( \mathbf{K}_i^\text{BB} - \sum_{j=1}^{p} \mathbf{K}_j^\text{IB}(\mathbf{K}_j^\text{II})^{-1}\mathbf{K}_j^\text{IB} \right) \mathbf{d}_i^\text{II} = \mathbf{f}_i^\text{II} - \sum_{j=1}^{p} \mathbf{K}_j^\text{IB}(\mathbf{K}_j^\text{II})^{-1}\mathbf{f}_j. \]  

\[ \text{(7)} \]

Once the vector \( \mathbf{d}_i^\text{II} \) is computed using (7), we can calculate the internal unknowns \( \mathbf{d}_i^\text{I} \) with (6).\n
It is not necessary to calculate the inverse in (7). Let’s define \( \tilde{\mathbf{R}}_i^\text{BB} = \mathbf{K}_i^\text{BB}(\mathbf{K}_i^\text{II})^{-1}\mathbf{K}_i^\text{IB} \), to calculate it [23], we proceed column by column using an extra vector \( \mathbf{t} \), and solving for \( c = 1 \ldots n \)
\[ \mathbf{K}_i^\text{II}\mathbf{t} = [\mathbf{R}_i^\text{IB}], \]  

\[ \text{(8)} \]

note that many \( \mathbf{K}_j^\text{II} \) are null. Next we can complete \( \mathbf{K}_i^\text{BB} \) with,
\[ [\mathbf{R}_i^\text{IB}] = \mathbf{K}_i^\text{BB}\mathbf{t}. \]

Now lets define \( \tilde{\mathbf{T}}_i^\text{BB} = \mathbf{K}_i^\text{BB}(\mathbf{K}_i^\text{II})^{-1}\mathbf{f}_i \), in this case only one system has to be solved \( \mathbf{K}_i^\text{II}\mathbf{t} = \mathbf{f}_i \), and then \( \tilde{\mathbf{T}}_i^\text{BB} = \mathbf{K}_i^\text{BB}\mathbf{t} \).

Each \( \tilde{\mathbf{T}}_i^\text{BB} \) and \( \tilde{\mathbf{T}}_i^\text{IB} \) holds the contribution of each partition to (7), this can be written as
\[ \left( \mathbf{K}_\text{BB} - \sum_{i=1}^{p} \tilde{\mathbf{R}}_i^\text{BB} \right) \mathbf{d}_i^\text{BB} = \mathbf{f}_i^\text{BB} - \sum_{i=1}^{p} \tilde{\mathbf{T}}_i^\text{BB}, \]  

\[ \text{(10)} \]

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

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

In case of (10), \( \mathbf{K}_\text{BB} \) is sparse, but \( \tilde{\mathbf{R}}_i^\text{BB} \) are not. To solve this system of equations an sparse version of conjugate gradient was implemented, the matrix \( \left( \mathbf{K}_\text{BB} - \sum_{i=1}^{p} \tilde{\mathbf{R}}_i^\text{BB} \right) \) 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 \( \tilde{\mathbf{T}}_i^\text{BB} \) 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 \( 1 \times 10^{-10} \).

**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) and the resulting deformation (right).

D. 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 unitary metallic square with Dirichlet conditions on all boundaries.

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) and the resulting temperature (right).](image)

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>Partition time [s]</th>
<th>Inversion time [s]</th>
<th>Schur c. time [s]</th>
<th>CG steps</th>
<th>Total time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>47.6</td>
<td>18521</td>
<td>4445</td>
<td>6927</td>
<td>23025</td>
</tr>
<tr>
<td>28</td>
<td>45.7</td>
<td>6270</td>
<td>2445</td>
<td>8119</td>
<td>8772</td>
</tr>
<tr>
<td>56</td>
<td>44.1</td>
<td>2257</td>
<td>2296</td>
<td>9627</td>
<td>4609</td>
</tr>
</tbody>
</table>

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 and split the systems of equations. Tolerance used was 1x10^-10.

![Figure 23. Domain and boundary conditions (left) partitioning (right).](image)

<table>
<thead>
<tr>
<th>Number of equations</th>
<th>Time [hours]</th>
<th>Memory [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>25,010,001</td>
<td>0.34</td>
<td>51.79</td>
</tr>
<tr>
<td>50,027,329</td>
<td>0.79</td>
<td>109.31</td>
</tr>
<tr>
<td>75,012,921</td>
<td>1.39</td>
<td>170.68</td>
</tr>
<tr>
<td>100,020,001</td>
<td>2.07</td>
<td>233.71</td>
</tr>
<tr>
<td>125,014,761</td>
<td>2.85</td>
<td>296.29</td>
</tr>
<tr>
<td>150,038,001</td>
<td>3.73</td>
<td>362.60</td>
</tr>
</tbody>
</table>

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 and split the systems of equations. Tolerance used was 1x10^-10.

![Figure 24. Number of equations vs. solution time (in hours).](image)
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. 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

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.

XII. REFERENCES


Figure 25. Number of equations vs. total memory used (in gigabytes).