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

J. M. Vargas-Felix, S. Botello-Rionda

***Abstract*—Finite element analysis of elastic deformation of three-dimensional complex structures using a fine mesh could require to solve systems of equations with several million variables.**

Domain decomposition is used to separate workload, instead of solving a huge system of equations, the domain is partitioned and for each partition a smaller system of equations is solved, all partitions are solved in parallel. Each partition is solved in a single MPI (Message Passing Interface) process, updates of boundary conditions among processes is done through MPI message routines.

Parallelization of solvers is also done by using OpenMP. Algorithms for solving systems of equations are implemented to run in multi-core processors, we will try two kinds of solvers: iterative (preconditioned conjugate gradient) and direct (Cholesky factorization). We will discuss some strategies to make these solvers parallelizable.

Numerical experiments were done using a program that we created from scratch, it was programed in C++ and tested in a Beowulf cluster with multi-core nodes.

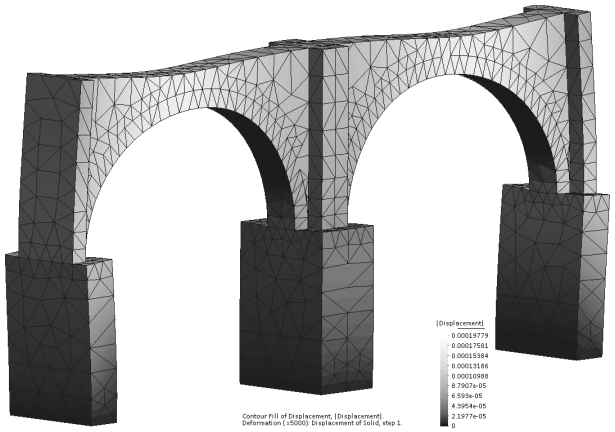
Different configurations for domain decompositions are tested, for instance, using many small partitions each one using a single core, or fewer partitions using several cores that share memory. This kind of experiments becomes very useful when one is looking for an adequate compromise between solution time and memory requirements when a Beowulf cluster is used.

***Index terms* — Parallel computing, domain decomposition, finite element method, sparse systems, linear algebra.**

I. Introduction

This is a high performance/large-scale application case study of the finite element method for solid mechanics. Our goal is to calculate displacements, strain and stress of solids discretized with large meshes (millions of elements) using parallel computing.

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



The displacement vector inside the domain is defined as

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

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

$$\boldsymbol{\varepsilon}=\begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_y \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix}=\begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial w}{\partial z} \\ \frac{\partial u}{\partial y}+\frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial z}+\frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x}+\frac{\partial u}{\partial z} \end{pmatrix}=\begin{pmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix}=\mathbf{E}\,\boldsymbol{u}, \qquad (1)$$

where ε_x, ε_y and ε_z are normal strains; γ_x, γ_x and γ_x are shear strains. We define a differential operator $\mathbf{E}$.

Stress vector is defined as

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

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

Stress an strain are related by

$$\boldsymbol{\sigma}=\boldsymbol{D}\boldsymbol{\varepsilon}; \qquad (2)$$

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

Solution is found using the finite element method with the Galerkin weighted residuals. This means that we solve the integral problem in each element using a weak formulation.

Manuscript received March 31, 2011.

J. Miguel Vargas-Felix and Salvador Botello-Rionda are with the Center of Mathematical Resarch (CIMAT), Guanajuato, Gto. 36240 México (e-mail: miguelvargas@cimat.mx and botello@cimat.mx).

The integral expression of equilibrium in elasticity problems can be obtained using the principle of virtual work [Zien05 pp65-71],

$$\int_V \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\boldsymbol{\sigma}\,dV=\int_V \delta\boldsymbol{u}^{\mathrm{T}}\boldsymbol{b}\,dV+\oint_A \delta\boldsymbol{u}^{\mathrm{T}}\boldsymbol{t}\,dA+\sum_i \delta\boldsymbol{u}_i^{\mathrm{T}}\boldsymbol{q}_i, \tag{3}$$

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

$$\boldsymbol{u}=\sum_{i=1}^{M} N_i\boldsymbol{u}_i. \tag{4}$$

Using (4), we can rewrite (1) as:

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

or in a more compact form

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

Now we can express (2) as $\boldsymbol{\sigma}=\boldsymbol{D}\boldsymbol{B}\boldsymbol{u}$, and then (3) by

$$\underbrace{\int_{V^e}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{B}\,dV^e}_{\boldsymbol{K}^e}\boldsymbol{u}=\underbrace{\int_{V^e}\boldsymbol{b}\,dV^e}_{\boldsymbol{f}_b^e}+\underbrace{\oint_{A^e}\boldsymbol{t}\,dA^e}_{\boldsymbol{f}_t^e}+\boldsymbol{q}^e. \tag{5}$$

By integrating (5) we obtain a system of equations for each element,

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

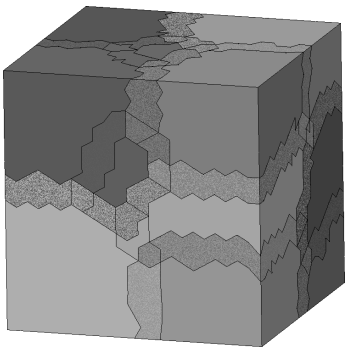
All systems of equations are assembled in a global system of equations,

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

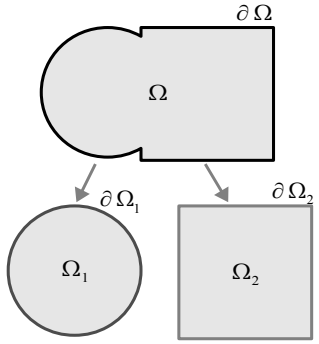
$\boldsymbol{K}$ is called the stiffness matrix, if enough boundary conditions are applied, it will be symmetric positive definite (SPD). By construction it is sparse with storage requirements of order $O(n)$, where n is the total number of nodes in the domain. By solving this system we will obtain the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

II. Domain decomposition

We can separate a huge finite element problem into smaller problems by partitioning the mesh to create sub-domains. In this case we will use domain decomposition with overlapping, these methods were first studied by Schwarz [Tose05]. For each sub-domain a system of equations with a SPD matrix is assembled thus we can solve it using solver algorithms that are implemented to run in parallel, such as Cholesky factorization or preconditioned conjugate gradient.

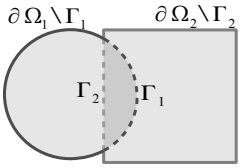


The domain decomposition algorithm we used is parallel Schwarz alternating method [Smit96], this is an iterative algorithm. We start with a domain Ω with boundary $\partial\Omega$.



Let $\boldsymbol{L}$ be a differential operator such that $\boldsymbol{L}\,\boldsymbol{x}=\boldsymbol{y}$ in Ω. Dirichlet conditions $\boldsymbol{x}=\boldsymbol{b}$ are applied on $\partial\Omega$. The domain is divided in two partitions Ω_1 and Ω_2 with boundaries $\partial\Omega_1$ and $\partial\Omega_2$ respectively.

Partitions are overlapped, now $\Omega=\Omega_1\cup\Omega_2$ *and* $\Omega_1\cap\Omega_2\neq\emptyset$.



We define artificial boundaries Γ_1 and Γ_2, these are part of Ω_1 and Ω_2, and are inside Ω.

Schwarz alternating method consists on solving each partition independently, fixing Dirichlet conditions in artificial boundaries with values from adjacent partition resulting from previous iteration.

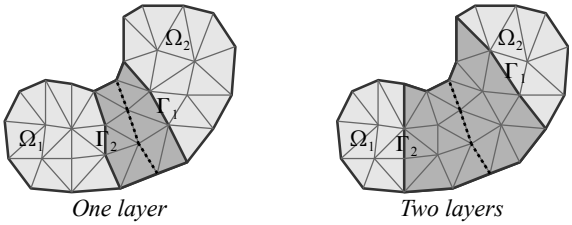
$\boldsymbol{x}_1^0$, $\boldsymbol{x}_2^0$, initial approximations
ε tolerance
$i \leftarrow 0$
while $\left\|\boldsymbol{x}_1^i-\boldsymbol{x}_1^{i-1}\right\|>\varepsilon$ or $\left\|\boldsymbol{x}_2^i-\boldsymbol{x}_2^{i-1}\right\|>\varepsilon$

solve		solve	
$\boldsymbol{L}\,\boldsymbol{x}_1^i=\boldsymbol{y}$	in Ω_1	$\boldsymbol{L}\,\boldsymbol{x}_2^i=\boldsymbol{y}$	in Ω_2
with $\boldsymbol{x}_1^i=\boldsymbol{b}$	on $\partial\boldsymbol{\Omega}_1\backslash\boldsymbol{\Gamma}_1$	with $\boldsymbol{x}_2^i=\boldsymbol{b}$	on $\partial\boldsymbol{\Omega}_2\backslash\boldsymbol{\Gamma}_2$
$\boldsymbol{x}_1^i\leftarrow\boldsymbol{x}_2^{i-1}\big\vert_{\Gamma_1}$	on $\boldsymbol{\Gamma}_1$	$\boldsymbol{x}_2^i\leftarrow\boldsymbol{x}_1^{i-1}\big\vert_{\Gamma_2}$	on $\boldsymbol{\Gamma}_2$

$i\leftarrow i+1$

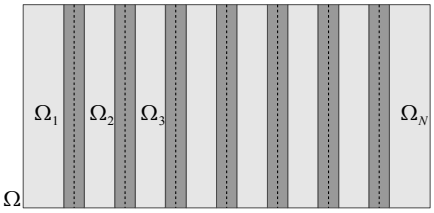
When the $\boldsymbol{L}$ operator has a matrix representation, alternating Schwarz algorithm corresponds (due to overlapping) to the iterative Gauss-Seidel by blocks [Smit96 p13].

In finite element problems, overlapping is adding to each partition one or several layers of elements adjacent to the boundary between partitions.



A. Convergence speed

There is a degradation on the convergence speed when the number of partitions raise [Smith96 p53]. Next image shows a pathological case of domain decomposition.



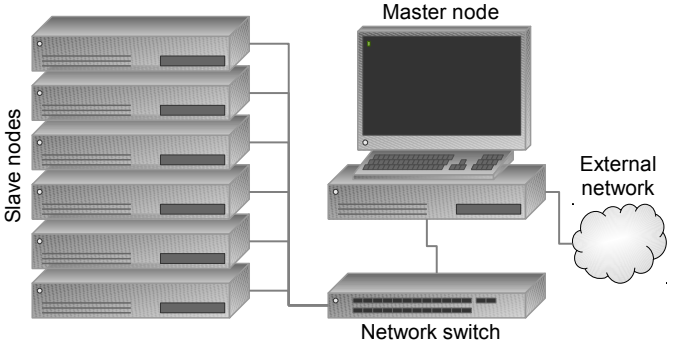
In each iteration of the alternating Schwarz method information is only transmitted to adjacent partitions. Therefore, if we have a boundary condition different to zero in the boundary of Ω_1, and we start in iteration 0, it will take N iterations for the local solution of partition Ω_N to be different to 0.

The Schwarz algorithm typically converge at a speed that is independent (or slightly independent) of mesh density when the overlapping is large enough [Smith96 p74].

A deeper study of theory of Schwarz algorithms can be found in [Tose05].

III. Computer clusters and MPI

We developed a software program that runs in parallel in a Beowulf cluster [Ster95]. A Beowulf cluster consists of several multi-core computers (nodes) connected with a high speed network.



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

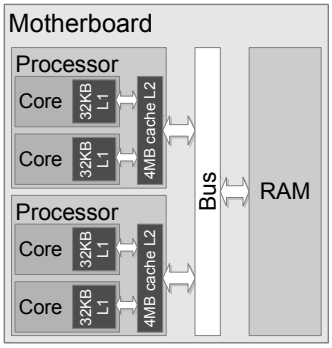
For partitioning the mesh we used the METIS library [Kary99].

IV. OpenMP

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

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

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



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

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

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

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

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

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

Algorithms to solve our system of equations should take care of these strategies.

V. Matrix storage

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

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

$$\boldsymbol{A}=\begin{pmatrix} 8 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 2 & 0 & 1 & 0 & 7 & 0 \\ 0 & 9 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}, \quad \begin{array}{|c|c|}\hline 8 & 4 \\ \hline 1 & 2 \\ \hline \end{array} \quad \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 3 & 4 \\ \hline \end{array} \quad \begin{array}{|c|c|c|}\hline 2 & 1 & 7 \\ \hline 1 & 3 & 5 \\ \hline \end{array} \quad \begin{array}{|c|c|c|}\hline 9 & 3 & 1 \\ \hline 2 & 3 & 6 \\ \hline \end{array} \quad \begin{array}{|c|}\hline 5 \\ \hline 6 \\ \hline \end{array}$$

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

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

The size of the 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 [Drep07].

VI. Parallel Cholesky 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 this strategies we could maintain memory and time requirements near to $O(n)$. Also Cholesky factorization could be implemented in parallel.

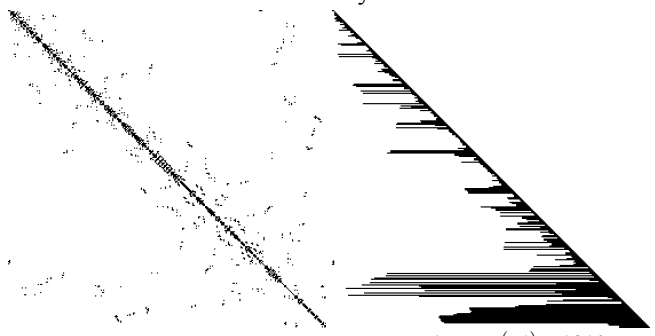
Formulae to calculate $\boldsymbol{L}$ entries are

$$L_{ij}=\frac{1}{L_{jj}}\left(A_{ij}-\sum_{k=1}^{j-1}L_{ik}L_{jk}\right), \text{ for } i>j; \tag{6}$$

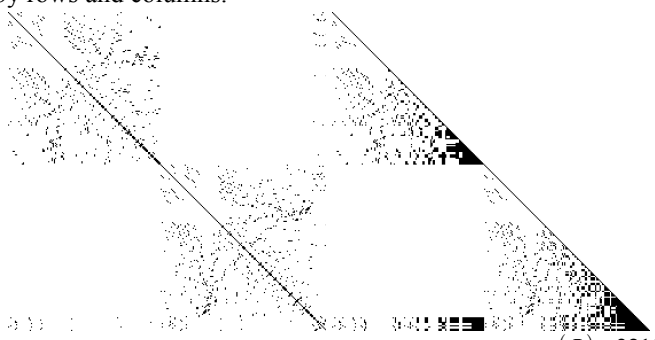
$$L_{jj}=\sqrt{A_{jj}-\sum_{k=1}^{j-1}L_{jk}^2}. \tag{7}$$

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 $\boldsymbol{A}\in\mathbb{R}^{556\times556}$ and the non zero entries of its Cholesky factorization $\boldsymbol{L}$.



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 reordering by rows and columns.



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

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

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

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

There are more complex algorithms that perform better in terms of time and memory requirements, the nested dissection

algorithm developed by Karypis and Kumar [Kary99] included in METIS library gives very good results.

B. Symbolic Cholesky factorization

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

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

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

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:

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

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

C. Filling entries in parallel

Once non zero entries are determined we can rewrite (6) and (7) as

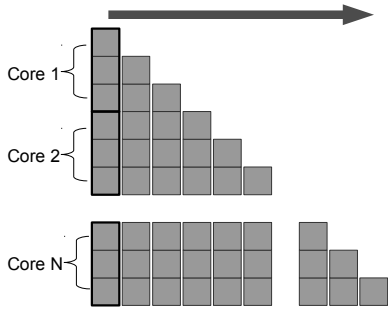
$$L_{ij}=\frac{1}{L_{jj}}\left(A_{ij}-\sum_{\substack{k\in \boldsymbol{j}_i^{\boldsymbol{L}}\cap \boldsymbol{j}_j^{\boldsymbol{L}} \\ k<j}} L_{ik}L_{jk}\right),\ \text{for } i>j;$$

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

The resulting algorithm to fill non zero entries is:

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
        $L_{ij} \leftarrow L_{ij}-\left(\boldsymbol{v}_i^{\boldsymbol{L}}\right)_r\left(\boldsymbol{v}_j^{\boldsymbol{L}}\right)_s$
        $r \leftarrow r+1;\ \rho \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_r$
        $s \leftarrow s+1;\ \sigma \leftarrow \left(\boldsymbol{j}_i^{\boldsymbol{L}}\right)_s$
    $L_{ij} \leftarrow \dfrac{L_{ij}}{L_{jj}}$
    $L_{ji}^{\mathrm{T}} \leftarrow L_{ij}$

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



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

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.

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 $\|\cdot\|_2$,

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

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 will talk about three kinds of preconditioners suitable for sparse systems with SPD matrices:

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

For the first two preconditioners, $\boldsymbol{M}$ is not constructed, instead $\boldsymbol{M}^{-1}$ is defined and we have to solve a system of equations in each step to obtain $\boldsymbol{q}_k$

$$\boldsymbol{M}^{-1}\,\boldsymbol{q}_k = \boldsymbol{r}_k.$$

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

A. Jacobi preconditioner

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.

B. Incomplete Cholesky factorization preconditioner

This preconditioner has the form

$$\boldsymbol{M}^{-1} = \boldsymbol{G}_l\,\boldsymbol{G}_l^{\mathrm{T}},$$

where $\boldsymbol{G}_l$ is a lower triangular sparse matrix that have structure similar to the Cholesky factorization of $\boldsymbol{A}$.

- The structure of $\boldsymbol{G}_0$ is equal to the structure of the lower triangular form of $\boldsymbol{A}$.
- The structure of $\boldsymbol{G}_m$ is equal to the structure of $\boldsymbol{L}$ (complete Cholesky factorization of $\boldsymbol{A}$).
- For $0 < l < m$ the structure of $\boldsymbol{G}_l$ is creating having a number of entries between $\boldsymbol{L}$ and the lower triangular form of $\boldsymbol{A}$, making easy to control the sparsity of the preconditioner.

Values of $\boldsymbol{G}_l$ are filled using (6) and (7). This preconditioner is not always stable [Golu96 p535].

The use of this preconditioner implies to solve a system of equations in each CG step using a backward and a forward substitution algorithm, this operations are fast given the sparsity of $\boldsymbol{G}_l$. Unfortunately the dependency of values makes these substitutions very hard to parallelize.

C. 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.

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

$$F(\boldsymbol{M}) = \|\boldsymbol{I} - \boldsymbol{A}\,\boldsymbol{M}\|_{\mathrm{F}}^2. \tag{8}$$

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 (8) into decoupled sums of 2-norms for each column [Chow98],

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

where $\boldsymbol{e}_j$ is the j-th column of $\boldsymbol{I}$ and $\boldsymbol{m}_j$ is the j-th column of $\boldsymbol{M}$. With this separation we can parallelize the construction of the preconditioner.

The factorized sparse approximate inverse preconditioner [Chow01] creates a preconditioner

$$\boldsymbol{M} = \boldsymbol{G}_l^{\mathrm{T}}\,\boldsymbol{G}_l,$$

where $\boldsymbol{G}$ is a lower triangular matrix such that

$$\boldsymbol{G}_l \approx \boldsymbol{L}^{-1},$$

where $\boldsymbol{L}$ is the Cholesky factor of $\boldsymbol{A}$. l is a positive integer that indicates a level of sparsity of the matrix.

Instead of minimizing (8), 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

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

this is equivalent to

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

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

This preconditioner has these features:

- $\boldsymbol{M}$ is SPD if there are no zeroes in the diagonal of $\boldsymbol{G}_l$.
- The algorithm to construct the preconditioner is parallelizable.
- This algorithm is stable if $\boldsymbol{A}$ is SPD.

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

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

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

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

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

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

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

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

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

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

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

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

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

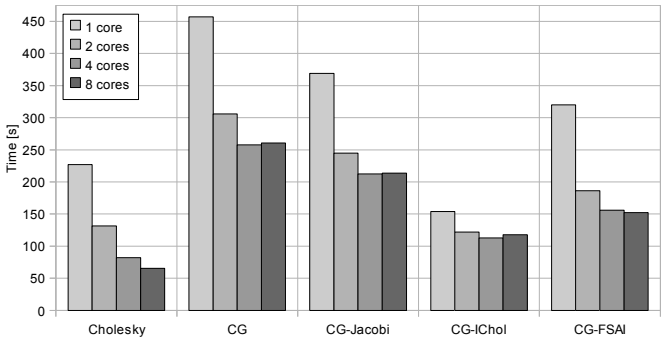
The vector $\boldsymbol{q}_k \leftarrow \boldsymbol{M}\boldsymbol{r}_k$ is calculated with two matrix-vector products,

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

VIII. Numerical Experiments

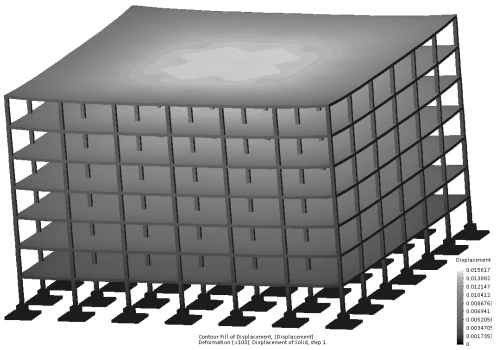
A. Solutions with OpenMP

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

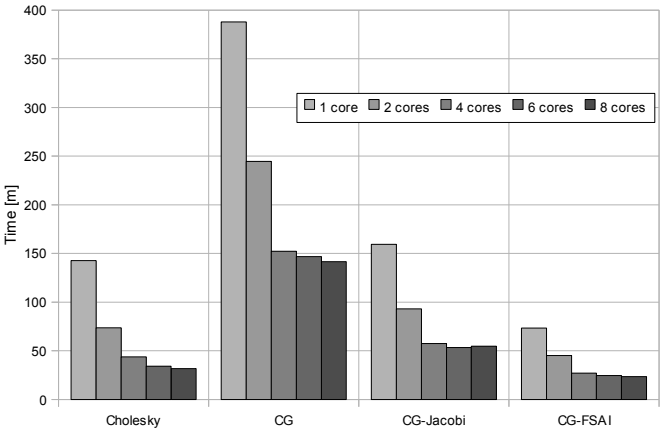


Solver	1 core time [s]	2 cores time [s]	4 cores time [s]	8 cores time [s]	Steps	Memory
Cholesky	227	131	82	65		3,051,144,550
CG	457	306	258	260	9,251	317,929,450
CG-Jacobi	369	245	212	214	6,895	325,972,366
CG-IChol	154	122	113	118	1,384	586,380,322
CG-FSAI	320	187	156	152	3,953	430,291,930

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



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



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

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

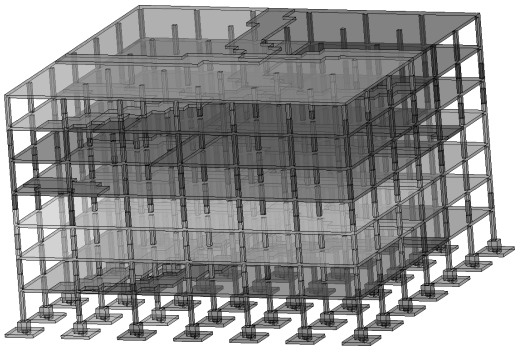
B. Solutions with MPI+OpenMP

Test were executed in a cluster with 14 nodes, each one with two dual core Intel Xeon E5502 (1.87GHz) processors, a

total of 56 cores.

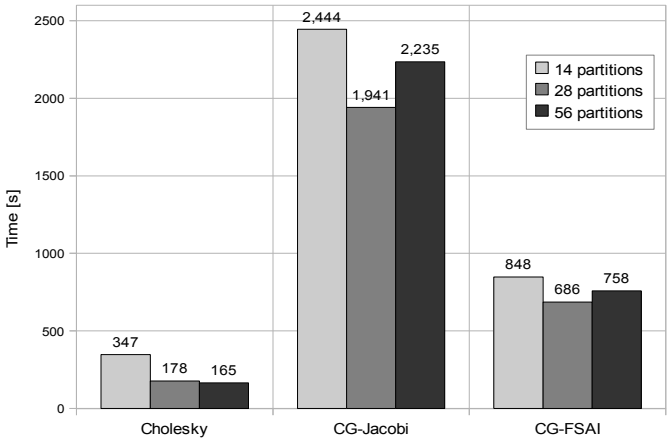
The problem tested is the same 3D solid model of a building. Using domain decomposition we tested the this problem using the following configurations:

- 14 partitions in 14 computers, using 4 cores per solver.
- 28 partitions in 14 computers, using 2 cores per solver.
- 56 partitions in 14 computers, using 1 core per solver.



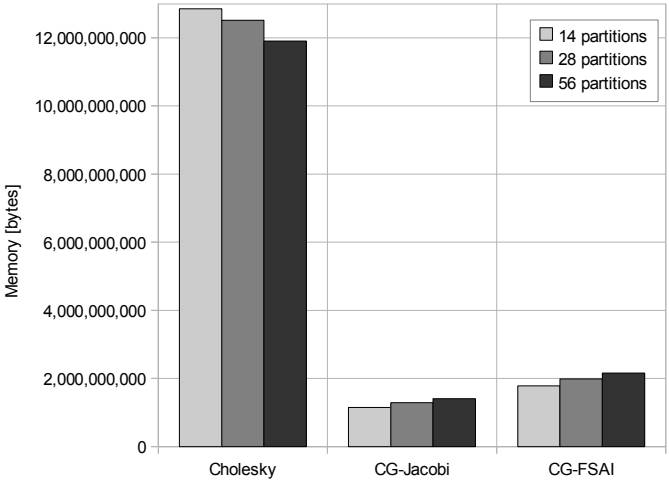
Parallel alternating Schwarz method is set to iterate until a global tolerance of $\|\boldsymbol{u}_i\| \leq 1 \times 10^{-4}$ is reached for all partitions.

Solution times:



Partitions	Cholesky	CG-Jacobi	CG-FSAI
14	347.3	2,444.2	847.5
28	177.5	1,940.5	685.9
56	165.3	2,234.8	757.9

Memory usage:

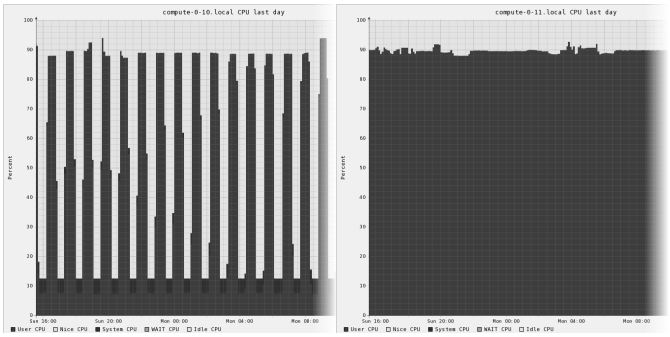


Partitions	Cholesky	CG-Jacobi	CG-FSAI
14	12,853,865,804	1,149,968,796	1,779,394,516
28	12,520,198,517	1,290,499,837	1,985,459,829
56	11,906,979,912	1,405,361,320	2,156,224,760

IX. Conclusions

We found that incomplete Cholesky factorization is unstable for some matrices, it is possible to stabilize the solver making the preconditioner diagonal-dominant, but we have to use a heuristic to do so.

The big issue for domain decomposition with iterative solvers is load balancing. Even thought partitioned meshes had almost the same number of nodes, the condition number of each matrix could vary a lot, making difficult to efficiently balance workload in each Schwarz iteration. The following images show this effect in several iterations. Left image correspond to the workload of the fastest solved partition (less used core), right image shows the workload of the slower solved partition (core used intensively).



It is complex to partition domains in such way that each partition take the same time to be solved. This issue is less noticeable when Cholesky solver is used.

To split the problem using domain decomposition with Cholesky works well, the fastest configuration was using one thread per solver. The obvious drawback is the memory consumption. We still can solve larger systems of equations using CG with FSAI but it will take more time.

For future work, some strategies can be taken to improve convergence:

- Create from the problem mesh a coarse mesh to solve this first and have a two level solution, the coarse solution is used in the Schwartz algorithm and have a better approximation
- It is possible to create the preconditioners from the overlapping of partitions to improve convergence.
- Original alternating Schwarz algorithm does not solve both partitions at the same time, it alternates. Partition coloring could be used to solve in parallel all non adjacent partitions with color 1, and use these solutions as boundary conditions for all partitions with color 2, etc. Several colors could be used.

REFERENCES

[Chow98] E. Chow, Y. Saad. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. SIAM Journal on Scientific Computing. Vol. 19-3, pp. 995-1023. 1998.

[Chow01] E. Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. International Journal of High Performance Computing, Vol 15. pp 56-74, 2001.

[DAze93] E. F. D'Azevedo, V. L. Eijkhout, C. H. Romine. Conjugate Gradient Algorithms with Reduced Synchronization Overhead on Distributed Memory Multiprocessors. Lapack Working Note 56. 1993.

[Drep07] U. Drepper. What Every Programmer Should Know About Memory. Red Hat, Inc. 2007.

[Gall90] K. A. Gallivan, M. T. Heath, E. Ng, J. M. Ortega, B. W. Peyton, R. J. Plemmons, C. H. Romine, A. H. Sameh, R. G. Voigt, Parallel Algorithms for Matrix Computations, SIAM, 1990.

[Geor81] A. George, J. W. H. Liu. Computer solution of large sparse positive definite systems. Prentice-Hall, 1981.

[Geor89] A. George, J. W. H. Liu. The evolution of the minimum degree ordering algorithm. SIAM Review Vol 31-1, pp 1-19, 1989.

[Golu96] G. H. Golub, C. F. Van Loan. Matrix Computations. Third edition. The Johns Hopkins University Press, 1996.

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

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

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

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

[Smit96] B. F. Smith, P. E. Bjorstad, W. D. Gropp. Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, 1996.

[Ster95] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, C. V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. Proceedings of the 24th International Conference on Parallel Processing, 1995.

[Tose05] A. Toselli, O. Widlund. Domain Decomposition Methods - Algorithms and Theory. Springer, 2005.

[Wulf95] W. A. Wulf , S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News, 23(1):20-24, March 1995.

[Zien05] O.C. Zienkiewicz, R.L. Taylor, J.Z. Zhu, The Finite Element Method: Its Basis and Fundamentals. Sixth edition, 2005.