Acta Universitaria
ISSN: 0188-6266
actauniversitaria@ugto.mx
Universidad de Guanajuato
México

Vargas-Félix, Miguel; Botello-Rionda, Salvador
Solution of finite element problems using hybrid parallelization with MPI and OpenMP
Acta Universitaria, vol. 22, núm. 7, octubre-noviembre, 2012, pp. 14-24
Universidad de Guanajuato
Guanajuato, México

Solution of finite element problems using hybrid parallelization with MPI and OpenMP

Solución de problemas de elemento finito utilizando paralelización híbrida con MPI y OpenMP

Miguel Vargas-Félix*, Salvador Botello-Rionda*

ABSTRACT

The Finite Element Method (FEM) is used to solve problems like solid deformation and heat diffusion in domains with complex geometries. This kind of geometries requires discretization with millions of elements; this is equivalent to solve systems of equations with sparse matrices and tens or hundreds of millions of variables. The aim is to use computer clusters to solve these systems. The solution method used is Schur substructuration. Using it is possible to divide a large system of equations into many small ones to solve them more efficiently. This method allows parallelization. MPI (Message Passing Interface) is used to distribute the systems of equations to solve each one in a computer of a cluster. Each system of equations is solved using a solver implemented to use OpenMP as a local parallelization method.

RESUMEN

El Método de Elemento Finito (FEM, por sus siglas en inglés) es utilizado para resolver problemas como la deformación de sólidos o la difusión de calor en dominios con geometrías complejas. Este tipo de geometrías requiere de discretizaciones con millones de elementos, lo que equivale a resolver sistemas de ecuaciones con matrices dispersas de decenas o cientos de millones de variables. La meta es utilizar clústeres de computadoras para resolver estos sistemas. El método de solución utilizado es la subestructuración de Schur. Utilizando ésta es posible dividir un sistema grande de ecuaciones en muchos pequeños para resolverse más eficientemente. Este método permite la paralelización. La MPI (*Message Passing Interface*, Interfaz para Paso de Mensajes) es utilizada para distribuir los sistemas de ecuaciones a resolver en cada computadora del cluster. Cada sistema de ecuaciones es resuelta utilizando un *solver* implementado con OpenMP como método de paralelización local.

INTRODUCTION

Solid deformation

It is necessary to calculate linear inner displacements of a solid resulting from forces or displacements imposed on its boundaries. The displacement vector inside the domain is defined as

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

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

Recibido: 28 de junio de 2011
Aceptado: 9 de octubre de 2011

Keywords:
Parallel computing; sparse matrices; linear solvers; partial differential equations.

Palabras clave:
Cómputo en paralelo; matrices dispersas; solvers lineales; ecuaciones diferenciales parciales.

*Computer Science Department. Centre for Mathematical Research (CIMAT). Jalisco alley w/n, Mineral de Valenciana, Zip Code 36240, Guanajuato, Gto., Mexico. E-mails: miguelvargas@cimat.mx, botello@cimat.mx

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

where **u** is the displacement vector, **ε** is the stain and σ is the stress. *D* is called "the constitutive matrix". A differential operator *E* is defined.

Stress vector is defined as $\boldsymbol{\sigma} = \left(\sigma_x, \sigma_y, \sigma_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\right)^{\mathrm{T}}$,

where σ_x, σ_y and σ_z are normal stresses; τ_x, τ_x and τ_x are tangential stresses. Stress and strain are related by

$$\boldsymbol{\sigma} = D\boldsymbol{\varepsilon}, \tag{1}$$

D is called "the constitutive matrix" and depends on Young moduli and Poisson coefficients characteristic of media.

Solution is found using the Finite Element Method (FEM) with the Galerkin weighted residuals. This means that the integral problem in each element is solved using a weak formulation. The integral expression of equilibrium in elasticity problems can be obtained using the principle of virtual work [1]:

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

here **b**, **t** and **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 and u_i is the coordinate of the *i*th node, so it can be had that

$$\mathbf{u} = \sum_{i=1}^{M} N_i u_i\,. \tag{3}$$

Using (3), it is possible to rewrite (1) as

$$\boldsymbol{\varepsilon} = \sum_{i=1}^{M} EN_i u_i,$$

or in a more compact form

$$\boldsymbol{\varepsilon} = \begin{pmatrix} EN_1 & EN_2 & \cdots & EN_M \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_M \end{pmatrix} = B\mathrm{u}.$$

Now it can be expressed (6) as $\boldsymbol{\sigma} = DB\mathbf{u}$ and (2) by

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

By integrating (4), it is obtained a system of equations for each element: $K^e\mathbf{u}^e = \mathbf{f}_b^e + \mathbf{f}_t^e + \mathbf{q}^e$. All systems of equations are assembled in a global system of equations:

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

K is called "the stiffness matrix"; if enough boundary conditions are applied, it will be by construction Symmetric Positive Definite (SPD). Also 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, they will be obtained the displacements of all nodes in the domain.

Heat diffusion

The other problem to solve is the stationary case of the heat diffusion. It is modeled using the Poisson equation:

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

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

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

Boundary conditions could be Dirichlet $\varphi(x,y) = \bar{\varphi}(x,y)$ en Γ_φ, or Neumann $\mathbf{q}(x,y) = \bar{\mathbf{q}}(x,y)$ en Γ_φ.

In complex domains, it is complicated to obtain a solution $\varphi(x, y)$ that satisfies (5). It is necessary to look for an approximate solution φ that satisfies $f\left(\varphi(x,y)\right) = 0$, in the sense of a weighted integral, like

$$\int_\Omega W f\left(\varphi(x,y)\right)dxdy = 0,$$

where $W = W(x, y)$ is a weighting function.

Reformulating the problem as a weighted integral

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

Integrating by parts

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

using the definition of flux vector

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

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

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

MATERIALS AND METHODS

Schur complement method

This is a domain decomposition method with no overlapping [2]. The basic idea is to split a large system of equations into smaller systems that can be solved independently in different computers in parallel.

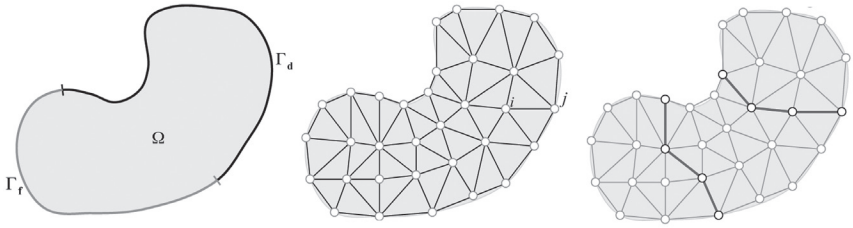


Figure 1. Finite element domain (left), domain discretization (center), partitioning (right).

This paper starts with a system of equations resulting from a finite element problem

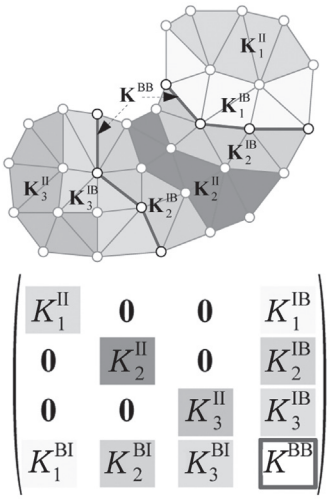
$$K\mathbf{d}=\mathbf{f}, \tag{6}$$

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

For this, it is useful to arrange (reorder variables) of the system of equations to have the following form

$$\begin{pmatrix} K_1^{\mathrm{II}} & & 0 & & & K_1^{\mathrm{IB}} & d_1^{\mathrm{I}} \\ & K_2^{\mathrm{II}} & & & & K_2^{\mathrm{IB}} & d_2^{\mathrm{I}} \\ 0 & & K_3^{\mathrm{II}} & & & K_3^{\mathrm{IB}} & d_3^{\mathrm{I}} \\ \vdots & & & \ddots & & \vdots & \vdots \\ & & & & K_p^{\mathrm{II}} & K_p^{\mathrm{IB}} & d_p^{\mathrm{I}} \\ K_1^{\mathrm{BI}} & K_2^{\mathrm{BI}} & K_3^{\mathrm{BI}} & \cdots & K_p^{\mathrm{BI}} & K^{\mathrm{BB}} & d^{\mathrm{B}} \end{pmatrix}=\begin{pmatrix} f_1^{\mathrm{I}} \\ f_2^{\mathrm{I}} \\ f_3^{\mathrm{I}} \\ \vdots \\ f_p^{\mathrm{I}} \\ f^{\mathrm{B}} \end{pmatrix}. \tag{7}$$

The superscript $^{\mathrm{II}}$ denotes entries that capture the relationship between nodes inside a partition. $^{\mathrm{BB}}$ is used to indicate entries in the matrix that relate nodes on the boundary. Finally, $^{\mathrm{IB}}$ and $^{\mathrm{BI}}$ are used for entries with values dependent of nodes in the boundary and nodes inside the partition.



$$\begin{pmatrix} K_1^{\mathrm{II}} & 0 & 0 & K_1^{\mathrm{IB}} \\ 0 & K_2^{\mathrm{II}} & 0 & K_2^{\mathrm{IB}} \\ 0 & 0 & K_3^{\mathrm{II}} & K_3^{\mathrm{IB}} \\ K_1^{\mathrm{BI}} & K_2^{\mathrm{BI}} & K_3^{\mathrm{BI}} & K^{\mathrm{BB}} \end{pmatrix}$$

Figure 2. Substructuring example with three partitions.

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

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

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

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

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

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

Once the vector $\mathbf{d}^{\mathrm{B}}$ is computed using (9), they are calculated the internal unknowns $\mathbf{d}_i^{\mathrm{I}}$ with (8). It is not necessary to calculate the inverse in (9). In order to define $\bar{K}_i^{\mathrm{BB}}=K_i^{\mathrm{BI}}\left(K_i^{\mathrm{II}}\right)^{-1}K_i^{\mathrm{IB}}$ and calculate it [3], it was proceeded column by column using an extra vector $\mathbf{t}$, and solving for $c = 1\ldots n$

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

(note that many $\left[K_i^{\mathrm{IB}}\right]_c$ are null). Next, K_i^{BB} can be completed with $\left[\bar{K}_i^{\mathrm{BB}}\right]_c=K_i^{\mathrm{BI}}\mathbf{t}$.

The definition of $\bar{\mathbf{f}}_i^{\mathrm{B}} = K_i^{\mathrm{BI}}\left(K_i^{\mathrm{II}}\right)^{-1}\mathbf{f}_i^{\mathrm{I}}$ comes next. In this case, only one system has to be solved

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

and then $\bar{\mathbf{f}}_i^{\mathrm{B}} = K_i^{\mathrm{BI}}\mathbf{t}$.

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

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

Once (12) is solved, they can calculated the inner results of each partition using (8).

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

In case of (12), K^{BB} is sparse but $\overline{K}_i^{\mathrm{BB}}$ is not. To solve this system of equations, a sparse version of conjugate gradient was implemented. The matrix $\left(K^{\mathrm{BB}} - \sum_{i=1}^{p}\overline{K}_i^{\mathrm{BB}}\right)$ 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 the present implementation, each $\overline{K}_i^{\mathrm{BB}}$ is maintained in their respective computer, the multiplication is done in a distributed way and the resulted vector is formed with contributions from all partitions. To improve the convergence of the conjugate gradient, a Jacobi preconditioned is used. This algorithm is described below.

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

Sparse matrices

Considering all elements, it was assembled a system of equations (with certain Dirichlet or Neumann boundary conditions) to solve a linear system of equations $A\mathbf{x} = \mathbf{b}$. 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.

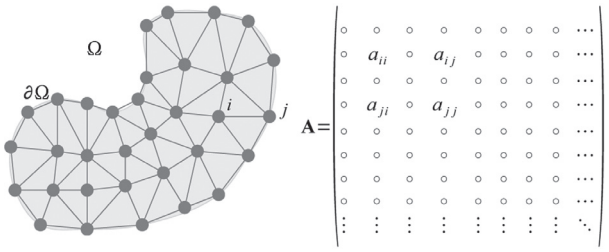


Figure 3. Discretized domain (mesh) and its matrix representation.

In this instance, it is turn to define the notation $\eta(A)$ (it indicates the number of non zero entries of A). For example, $A \in \mathbb{R}^{556\times556}$ has 309 136 entries with $\eta(A)$ = 1 810, this means that only the 0,58% of the entries are non zero.

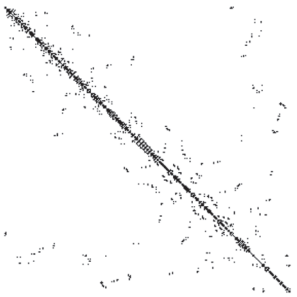


Figure 4. Black dots indicates a non zero entry in the matrix.

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

▫ Matrix storage

An efficient method to store and operate matrices of this kind of problems is the Compressed Row Storage (CRS) [4]. This method is suitable when it is wanted to access entries of each row of a matrix $\boldsymbol{A}$ sequentially.

For each row i of A, they will be had two vectors: a vector $\boldsymbol{v}_i^A$ that will contain the non zero values of the row and a vector $\boldsymbol{j}_i^A$ with their respective column indexes. This is an example of a matrix A and its CRS representation

$$A = \begin{pmatrix} 8 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 2 & 0 & 1 & 0 & 7 & 0 \\ 0 & 9 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}, \quad \begin{matrix} 8\ 4 \\ 1\ 2 \\ 1\ 3 \\ 3\ 4 \\ 2\ 1\ 7 \\ 1\ 3\ 5 \\ 9\ 3\ 1 \\ 2\ 3\ 6 \\ 5 \\ 6 \end{matrix} \quad \begin{matrix} \boldsymbol{v}_4^A = (9,3,1) \cdot \\ \\ \\ \\ \\ \\ \\ \\ \boldsymbol{j}_4^A = (2,3,6) \end{matrix}$$

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

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

The main advantage of using CRS is when data in each row is stored continuously and accessed in a sequential way. This is important because an efficient processor cache usage will be had [5].

Cholesky factorization for sparse matrices

The cost of using Cholesky factorization $A = LL^T$ is expensive if the researcher wants to solve systems of equations with full matrices. But, for sparse matrices, he could reduce this cost significantly if he uses reordering strategies and store factor matrices using CRS identifying non zero entries using symbolic factorization. With these strategies, he could maintain memory and time requirements near to $O(n)$. Also Cholesky factorization could be implemented in parallel.

Formulae to calculate L entries are

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

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}\,. \quad (14)$$

▫ Reordering rows and columns

By reordering the rows and columns of a SPD matrix A, it is possible to reduce the fill-in (the number of non-zero entries) of L. The next images show the non zero entries of $A \in \mathbb{R}^{556\times556}$ and the non zero entries of its Cholesky factorization L.

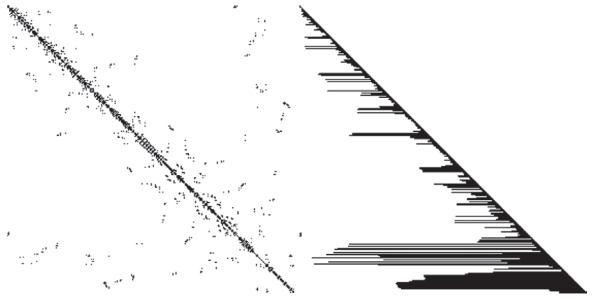


Figure 5. Left: non-zero entries of A. Right: non-zero entries of L (Cholesky factorization of A).

The number of non zero entries of A is $\eta(A) = 1\,810$ and for L is $\eta(L) = 8\,729$. The next images show A with an efficient reordering by rows and columns.

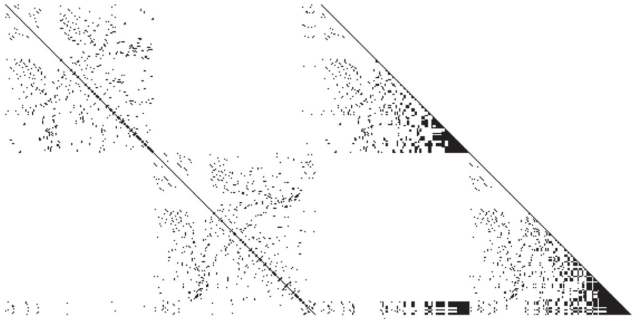


Figure 6. Left: non-zero entries of reordered A. Right: non-zero entries of L.

By reordering, there is a new factorization with $\eta(L) = 3\,215$, reducing the fill in to 0,368 of the size of the not reordered version. Both factorizations allow solving 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 [6]:

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

More advanced versions of this algorithm can be consulted in [7].

There are more complex algorithms that perform better in terms of time and memory requirements; the nested dissection algorithm developed by Karypis and Kumar [8] included in METIS library gives very good results.

▫ Symbolic Cholesky factorization

This algorithm identifies non zero entries of L. A deep explanation could be found in [9].

For an sparse matrix A, it is defined

$$a_j \overset{\text{def}}{=} \{k > j \mid A_{kj} \neq 0\},\ j = 1 \ldots n,$$

as the set of non zero entries of column j of the strictly lower triangular part of A. In similar way, for matrix it is defined the set

$$l_j \overset{\text{def}}{=} \{k > j \mid L_{kj} \neq 0\},\ j = 1 \ldots n.$$

Also, it is possible to use sets that define sets $\boldsymbol{r}_j$ and that will contain columns of L which structure will affect the column j of L. The algorithm is:

$r_j \leftarrow \emptyset$, $j \leftarrow 1 \ldots n$
for $j \leftarrow 1 \ldots n$
  $l_j \leftarrow a_j$
  for $i \in r_j$
    $l_j \leftarrow l_j \cup l_i \setminus \{j\}$
  end_for
  $p \leftarrow \begin{cases} \min\{i \in l_j\} & \text{if } l_j \neq \emptyset \\ j & \text{another case} \end{cases}$
  $r_p \leftarrow r_p \cup \{j\}$
end_for

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

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

$$a_2=\{3,4\} \qquad l_2=\{3,4,6\}$$

Figure 7. Example matrix showing how a_2 and l_2 are formed.

This algorithm is very efficient. Complexity in time and memory usage has an order of O(η(*L*)). Symbolic factorization could be seen as a sequence of elimination graphs [6].

▫ Filling entries in parallel

Once non zero entries are determined, (13) and (14) can be rewritten as

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

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

The resulting algorithm to fill non zero entries is [10]:

```
for j ← 1...n
    L_jj ← A_jj
    for q ← 1...|v_j^L|
        L_jj ← L_jj − (v_j^L)_q (v_j^L)_q
    L_jj ← √L_jj
    L_jj^T ← L_jj
    parallel for q ← 1...|j_j^{L^T}|
        i ← (j_j^{L^T})_q
        L_ij ← A_ij
        r ← 1; ρ ← (j_i^L)_r
        s ← 1; σ ← (j_i^L)_s
        repeat
            while ρ < σ
                r ← r+1; ρ ← (j_i^L)_r
            while ρ > σ
                s ← s+1; σ ← (j_i^L)_s
            while ρ = σ
                if ρ = j
                    exit repeat
                L_ij ← L_ij − (v_i^L)_r (v_j^L)_s
                r ← r+1; ρ ← (j_i^L)_r
                s ← s+1; σ ← (j_i^L)_s
        L_ij ← L_ij / L_jj
        L_ji^T ← L_ij
```

This algorithm could be parallelized if it is filled column by column. Entries of each column can be calculated in parallel with OpenMP, because there is no dependence among them [11]. Calculus of each column is divided among cores.

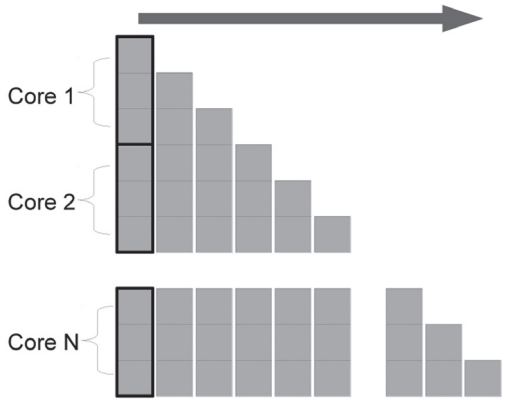


Figure 8. Calculation order to parallelize the Cholesky factorization.

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

Parallel preconditioned conjugate gradient

Conjugate Gradient (CG) is a natural choice to solve systems of equations with SPD matrices. In this article will be discussed 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 $A\in^{m\times m}$, given a norm $\|\cdot\|$ is defined as $\kappa(A) = \|A\| \cdot \|A^{-1}\|$.

For the norm $\|\cdot\|_2$, $\kappa_2(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \dfrac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$, where σ is a singular value of *A*.

For a SPD matrix, $\kappa(A)=\dfrac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$, where λ is an eigenvalue of *A*.

A system of equations $A\mathbf{x} = \mathbf{b}$ 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 *M A* 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 $A\mathbf{x} - \mathbf{b} = 0$, with preconditioning it is solved $M(A\mathbf{x} - \ = 0)$.

The preconditioned conjugate gradient algorithm is:

$\mathbf{x}_0$, initial approximation
$\mathbf{r}_0 \leftarrow \mathbf{b} \quad A\mathbf{x}_0$, initial gradient
$\mathbf{q}_0 \leftarrow M\mathbf{r}_0$
$\mathbf{p}_0 \leftarrow \mathbf{q}_0$, initial descent direction
$\mathbf{k} \leftarrow 0$
while $\|\mathbf{r}_k\| > \varepsilon$

$$\alpha_k \leftarrow \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{q}_k}{\mathbf{p}_k^{\mathrm{T}} A\mathbf{p}_k}$$
$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$$
$$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k \quad \alpha_k A\mathbf{p}_k$$
$$\mathbf{q}_{k+1} \leftarrow M\mathbf{r}_{k+1}$$
$$\beta_k \leftarrow \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{q}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{q}_k}$$
$$\mathbf{p}_{k+1} \leftarrow \mathbf{q}_{k+1} + \beta_k \mathbf{p}_k$$
$$k \leftarrow k+1$$

For large and sparse systems of equations, it is necessary to choose preconditioners that are also sparse. In this work was used the Jacobi preconditioner because it is suitable for sparse systems with SPD matrices. The diagonal part of M^{-1} is stored as a vector: $M^{-1} = (\mathrm{diag}(A))^{-1}$. Parallelization of this algorithm is straightforward, because the calculus of each entry of $\mathbf{q}_k$ is independent.

Parallelization of the preconditioned CG is done using OpenMP. Operations parallelized are matrix-vector, dot products and vector sums. Synchronizing threads has a computational cost, it is possible to modify to CG to reduce this costs maintaining numerical stability [12].

Computer clusters and MPI

It was developed a software program that runs in parallel in a Beowulf cluster [13]. A Beowulf cluster consists of several multi core computers (nodes) connected with a high speed network.

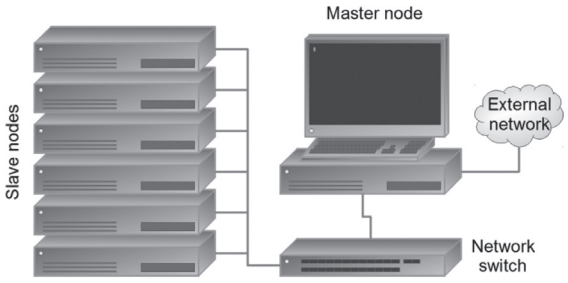


Figure 9. Diagram of a Beowulf cluster of computers.

In this software implementation, each partition is assigned to one process. To parallelize the program and move data among nodes, it has been used the Message Passing Interface (MPI) schema [14]; 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, it was used the METIS library [8].

Parallelization using multi core computers

Using domain decomposition with MPI, it was possible to have a partition assigned to each node of a cluster and to solve all partitions concurrently. If each node is a multi core computer, also it is givable to parallelizing the solution of the system of equations of each partition. To implement this parallelization, the OpenMP model was used.

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 [15].

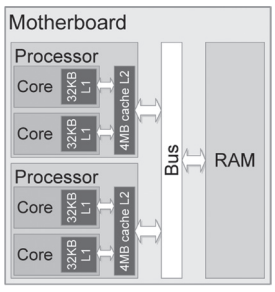


Figure 10. Schematic of a multi processor and multi core computer.

To overcome this, a high speed memory called "cache" exists between the processor and RAM. This cache reads blocks of data from RAM meanwhile the processor is busy, using a 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 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 means 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, thus, to keep consistency is complex and expensive [5]. Also, it is necessary to consider that cache circuits are designed to be more efficient when reading continuous memory data in an ascendant sequence [5].

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 this system of equations should take care of these strategies.

Numerical experiments

In the next lines is going to be presented just a couple examples; these were executed in a cluster with 15 nodes, each one with the following characteristics:

Processor	Two dual core Intel Xeon E5502 (1,87GHz)
Caches	L3 4 MB, L2 512KB, L1 64KB.
Memory	16 GB, DD3 1 066 MHz
Network	Ethernet, 1 Gbit.
RAM	240

Total: 60 cores. A node is used as a master process to load the geometry and the problem parameters, the partition and to split the systems of equations. The other 14 nodes are used to solve the system of equations of each partition. Times described are in seconds (s). Tolerance used is $1x10^{-10}$.

▫ 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. Now, with three degrees of freedom per node, the resulting system of equations has 5 124 819 unknowns.

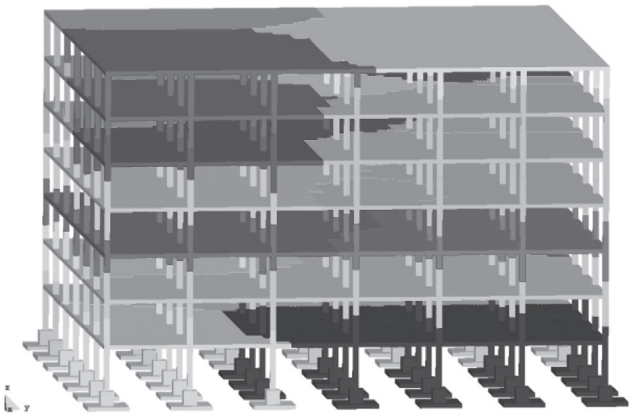
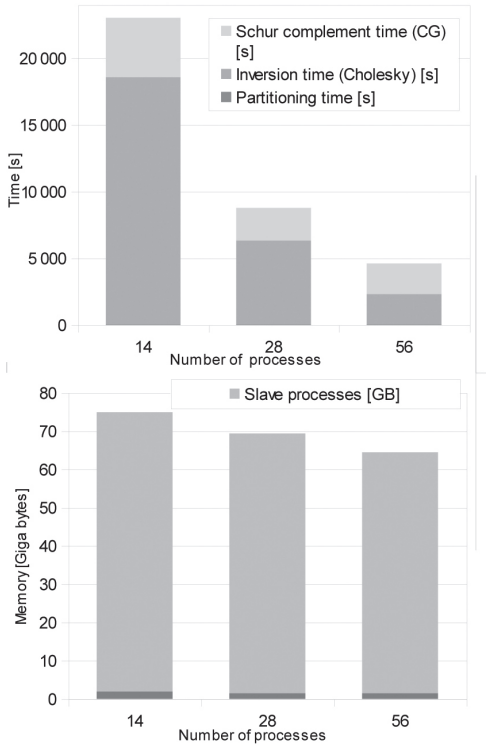


Figure 11. Substructuration of the domain.

Number of processes	Partitioning time [s]	Inversion time (Cholesky) [s]	Schur complement time (CG) [s]	CG steps	Total time [s]
14	47,6	18 520,8	4 444,5	6 927	2 3025,0
28	45,7	6 269,5	2 444,5	8 119	8771,6
56	44,1	2 257,1	2 296,3	9 627	4608,9



Number of processes	Master process [GB]	Slave processes [GB]	Total memory [GB]
14	1,89	73,00	74,89
28	1,43	67,88	69,32
56	1,43	62,97	64,41

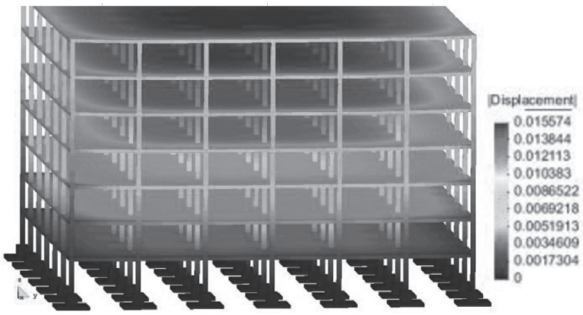


Figure 12. Resulting deformation.

▫ Heat diffusion

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

Number of processes	Partitioning time [s]	Inversion time (Cholesky) [s]	Schur complement time (CG) [s]	CG steps	Total time [s]
14	144,9	798,5	68,1	307	1 020,5
28	146,6	242,0	52,1	348	467,1
56	144,2	82,8	27,6	391	264,0

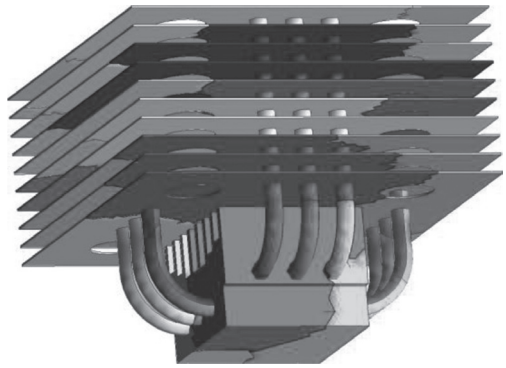
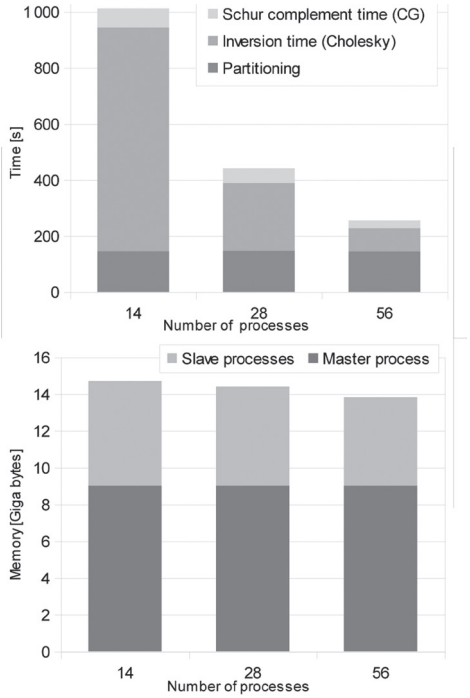


Figure 13. Substructuration of the domain.

Number of processes	Master process [GB]	Slave processes [GB]	Total memory [GB]
14	9,03	5,67	14,70
28	9,03	5,38	14,41
56	9,03	4,80	13,82



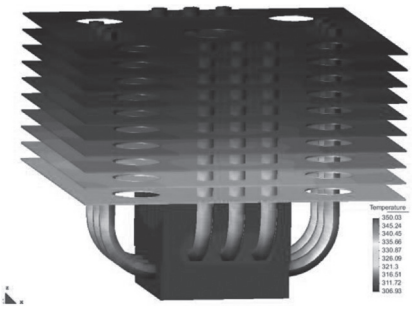


Figure 14. Resulting temperature distribution.

▫ Large systems of equations

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

Figure 15. Geometry example.

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

In this case, it has been used a larger cluster with mixed equipment of 15 nodes, each one with the following characteristics:

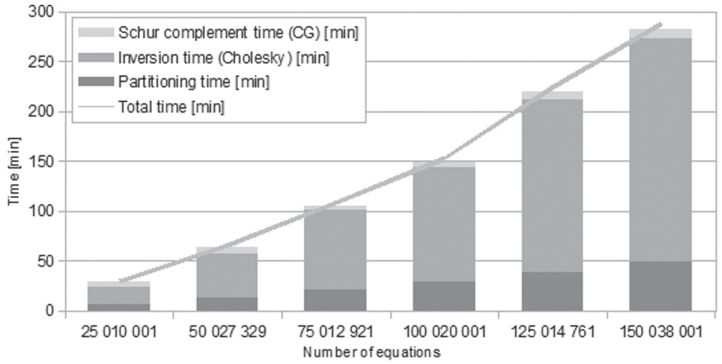
Processor	Two dual core Intel Xeon E5502 (1,87GHz)
Caches	L3 4 MB, L2 512 KB, L1 64 KB
Memory	16 GB, DD3 1 066 MHz
Network	Ethernet, 1 Gbit

14 nodes with the following characteristics:

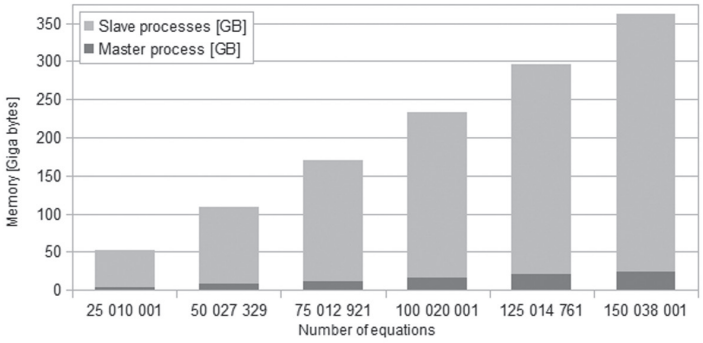
Processor	Two quad core AMD Opteron 2350 (2,8GHz)
Caches	L3 2 MB, L2 2 MB, L1 256 KB
Memory	12 GB, DD2 667 MHz
Network	Ethernet, 1 Gbit

Of the second set, only 4 cores per node were used. That gives a total of 116 cores. An extra node is used as a master process to load the geometry and the problem parameters, the partition and to split the systems of equations. Tolerance used was $1x10^{-10}$.

Equations	Partitioning time [min]	Inversion time (Cholesky) [min]	Schur complement time (CG) [min]	CG steps	Total time [min]
25 010 001	6,2	17,3	4,7	872,0	29,4
50 027 329	13,3	43,7	6,3	1 012,0	65,4
75 012 921	20,6	80,2	4,3	1 136,0	108,3
100 020 001	28,5	115,1	5,4	1 225,0	152,9
125 014 761	38,3	173,5	7,5	1 329,0	224,2
150 038 001	49,3	224,1	8,9	1 362,0	288,5



Equations	Master process [GB]	Average slave processes [GB]	Slave processes [GB]	Total memory [GB]
25 010 001	4,05	0,41	47,74	51,79
50 027 329	8,10	0,87	101,21	109,31
75 012 921	12,15	1,37	158,54	170,68
100 020 001	16,20	1,88	217,51	233,71
125 014 761	20,25	2,38	276,04	296,29
150 038 001	24,30	2,92	338,29	362,60



CONCLUSIONS

There have been presented just a few case studies of the usage of the Schur substructuring method for complex geometries with large number of degrees of freedom.

It is difficult to measure speed up when working with complex geometries. The partitioning routines used in this work [8] have heuristics that try to divide equally the number of nodes, thus the shape of the partitions for each mesh could vary a lot. Nevertheless, results have a linear tendency in reduction of solution times.

In this case, it was used the Jacobi preconditioner, but there are other preconditioners that lead to better convergence that could be interesting to test, like the family of methods called Finite Element Tearing and Interconnect (FETI) [16].

REFERENCES

[1] Zienkiewicz, O. C., Taylor, R. L. and Zhu, J. Z. (2005). T*he Finite Element Method: Its Basis and Fundamentals.* Sixth edition. Butterworth-Heinemann.

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

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

[4] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems.* SIAM.

[5] Drepper, U. (2007). *What Every Programmer Should Know About Memory.* Red Hat, Inc.

[6] George, A. and Liu, J. W. H. (1981). *Computer solution of large sparse positive definite systems.* Prentice-Hall.

[7] George, A. and Liu, J. W. H. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Review* 31(1): pp 1 19.

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

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

[10] Vargas-Felix, J. M. and Botello-Rionda, S. (2010). *Parallel Direct Solvers for Finite Element Problems* (Comunicación del CIMAT N. I-10-08). Available in http://www.cimat.mx/reportes/enlinea/I-10-08.pdf

[11] Heath, M T., Ng, E. and Peyton, B. W. (1991). Parallel Algorithms for Sparse Linear Systems. *SIAM Review* 33(3): pp. 420-460.

[12] D'Azevedo, E. F., Eijkhout, V. L. and Romine, C. H. (2002). Conjugate Gradient Algorithms with Reduced Synchronization Overhead on Distributed Memory Multiprocessors. *Lapack Working Note 56.* Available in http://www.netlib.org/lapack/lawnspdf/lawn56.pdf

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

[14] Message Passing Interface Forum. (2008). MPI: *A Message-Passing Interface Standard.* Version 2.1. University of Tennessee.

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

[16] Farhat, C. and Roux, F. X. (1991). A method of finite element tearing and interconnecting and its parallel solution algorithm. *Internat. J. Numer. Meths. Engrg.* 32: pp. 1205-1227.