Towards parallel solution of continuous problems by means of a general finite/spectral–element oriented C/C++ framework

Javier Quinteros[1,2] and Alejandro D. Otero[2,3,4]
javier@gfz-potsdam.de aotero@fi.uba.ar

[1] Deutsches GeoForschungsZentrum GFZ, Postdam, Germany
[2] FCEyN, Universidad de Buenos Aires, Argentina
[3] FI, Universidad de Buenos Aires, Argentina
[4] CONICET, Argentina

Abstract. In this work, we present the design and implementation of a highly modular and flexible software framework to implement numerical models based on the finite element method (FEM) and its extension to deal with distributed problems. This work improves the current implementation by the adition of parallel calculations capabilities by means of the substructure technique applied to solve problems by the FEM in clusters of computers using the MPI protocol. We considered the solution of a general Poisson problem as a test case to conduct experiments in order to evaluate the scaling capabilities of our code. Conclusions are extracted with focus on future lines of development.

1 Introduction

The design of a numerical model goes through different stages until it can be considered functional and satisfactory results are obtained. A correct formulation of the problem is needed for numerical models based on the finite element method (FEM) in order to assess, from a mathematical point of view, that a proper solution will be obtained. The equations, their discretization and the element employed are an integral part of these. However, the initial design stage includes not only the mathematical formulation, but also the *software design*.

As in many numerical problem, one of the most important features is the ability to solve bigger problems with finer meshes. As the amount of resources needed (namely memory and time) grow exponentially, the idea of dividing it into smaller sub-problems was studied by means of different approaches.

2 General Purpose Finite Element Framework

Work on parallel FEM implementation based on Object-Oriented Programming technics can be traced back, for example, to the work of Modak and Sotelino [1], Sonzogni *et al.* [2] and references therein. Here, we present the design and implementation of a highly modular and flexible software framework to implement

numerical models based on the FEM and its extension to deal with distributed problems. Its potentiality is based on proper abstractions. No underlying elemental formulation needs to be known in order to code the resolution of a problem. In addition, the implementation of algebraic operations is isolated from the elemental formulation.

This work improves the implementation already presented by Quinteros *et al.* [3, 4] by means of the substructure technique to solve problems by the FEM in clusters of computers using the MPI protocol.

2.1 Proposed Design

Identification of Stages and Entities. The finite element method is widely employed to solve partial differential equations (PDE). It is well known that, although many types of elements with their own properties exist, the FEM theory does not depend on a particular element. The FEM can be considered a framework in which many specific aspects of the formulations can vary.

From a high level point of view, the following stages can be identified:

- domain description,
- domain discretization,
- element matrices calculation,
- numerical integration in the specified Gauss points,
- assemblage of global matrices/vectors,
- imposition of boundary conditions and
- resolution of the equation system, among others.

Every stage is defined as independent from the others. Usually, the variations in the formulations can be achieved by means of appropriate parameters; thus, isolating the different processes. As in a *sequential* analysis of the operations many independent and generic stages can be defined, different layers can be identified inside the domain. The spatial concepts related to the discretization, like *domain*, *element*, *boundary condition*, *node* and *Gauss point*, introduce a new abstraction level, not in a sequential sense, but from an *entity* point of view. One can see in figure 1 an example of a domain that is formed by two elements. The elements are defined by four nodes located at their corners. The numerical interpolation should be calculated in four Gauss points.

Based on these concepts, the class diagram that can be seen in Fig. 2 is proposed where only the most important attributes of each class are included.

The class `Domain` is the one that includes all the information needed to describe the model by means of a set of elements and nodes stored in the attributes `Elements` and `Nodes`. Once the domain is known, every boundary condition is stored in an instance of the class `Boundary` and the set of all the boundary conditions that determine the problem is stored in another attribute of the `Domain` class (`Boundaries`).

It is important to note that the element is determined by a set of nodes that are connected. However, these nodes do not belong to the element because many

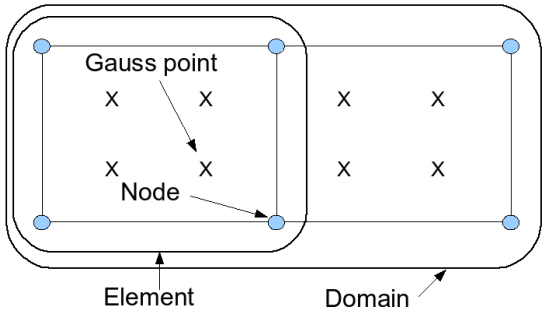

Fig. 1. Conceptual view of a domain composed by two elements. The elements are defined by four corner nodes and the numerical integration is calculated in four Gauss points. It is shown only as an example, as it is one of many element types that can be employed.

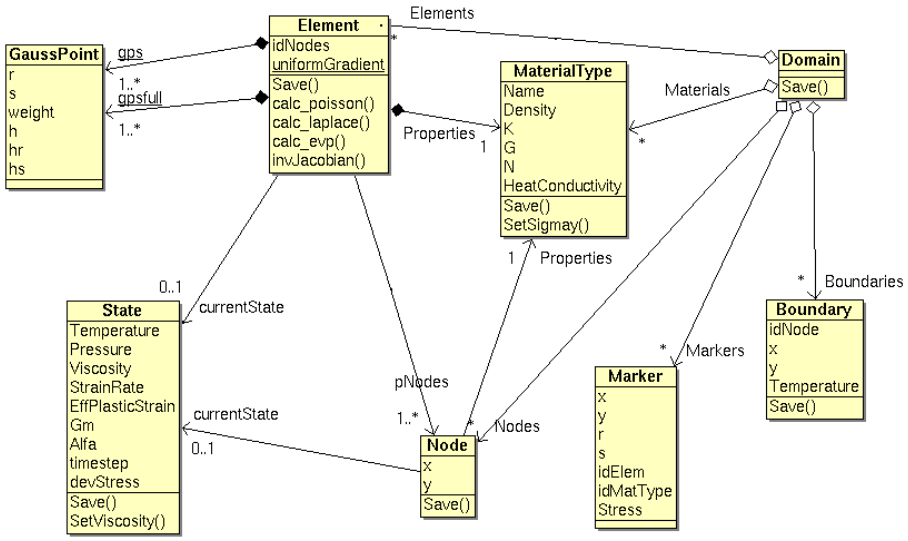

Fig. 2. Partial class diagram of the model.

of them are located on the border and are shared with another element. Thus, only the references to the nodes (`pNodes`) as well as their global id (`idNodes`) are stored in the class `Element`.

To calculate the stiffness matrix of each element, a numerical integration in the Gauss points determined by the order of interpolation is required. The position of the Gauss points is identical for every element because the integration is actually made in the master undeformed element. Thus, this set of Gauss points is stored in the attribute `gps` in `Element`, `gps` being declared as a `static` variable.

This way of storing the Gauss points is very flexible at the moment of the implementation of a selective reduced integration [5], which can be defined including only the ones that will be employed to integrate and modifying its weight accordingly. For the operations that cannot be computed without the whole set of Gauss points, another attribute called `gpsfull` is included in `Element` to be able to perform a full integration.

The `State` class stores variables (e.g. stress, strain, effective plastic strain) that are calculated by the model and must be stored arbitrarily in the `Element` or the `Node`. One can see in figure 2 that the `State` class is related to the `Element` *and* the `Node` class, by means of a pointer called `currentState`. Once the type of element is defined, as a part of the discretization passed as a parameter, it is clear for the model where the `State`s will be stored.

Interface with *Solvers* and Mathematical Libraries. It is known that the stiffness matrix associated with every instance of the `Element` class is computed by means of the multiplication of small dense matrices related to geometrical and compositional properties of the element. There are many thoroughly tested libraries for linear algebra operations that can be employed to address the matrices operations, being Lapack [6] a standard *de-facto*.

The class `Matrix` is designed to store all the dense matrices and vectors that take part of the framework. It also isolates the model from the particular Lapack library by means of an interface that provides a basic set with the most common operations for matrices related to the FEM.

This class provides a simple, natural and practical way of manipulating matrices. Overloading of operators and polymorphism were heavily used in order to achieve an intuitive notation, preventing obscure coding techniques.

In problems with a sufficient number of elements, the global mass and stiffness matrices will be very sparse and their size in memory would be of the order of the square of the number of total nodes. Another class called `SparseMatrix` was designed for this type of matrices, in order to improve the storage in memory and the time needed to process some operations. This class works as an interface that isolates the framework from the sparse matrix library which is actually used to solve the global system of equation.

At present the `SparseMatrix` class can interface with three different libraries: PARDISO [7], MUMPS [8] and SuperLU [9]. Thus, we have several options for solving systems of equation sequentially and parallelly, both in shared and dis-

tributed memory computers. These libraries provide fast and tested procedures to solve linear systems stored in sparse matrices.

2.2 Elemental Level Abstraction

Up to this moment, no mention has been made of the type of element employed, mainly because the generic schema of resolution does not need to know the element implemented to solve the problem. By means of the proposed design and interfaces, a nearly total freedom from the element implementation can be obtained. In the proposed design the `Element` class is defined as a base class that provides a common specification that the different types of element must implement. It also includes the specification of the functions to solve the different types of equations. All the element methods that cannot be defined in a generic way must be implemented in a separated class that inherits the specifications from `Element`.

The `Liu` class that can be seen in Fig. 3 is an example of a particular element implementation. In its simplest form, the solution of partial derivative equations by means of the FEM implies, for each element, the evaluation of matrix products (that depends on the problem) in some points called *Gauss points*. These evaluations are multiplied by a weight factor and summed to obtain the elemental stiffness matrix. From all these tasks, only the evaluation in a specific Gauss point needs information about the number of nodes that the element have. That is the main reason to implement in the `Element` class a method called `calc_equation`, where all the necessary steps to get the element stiffness matrix related to the problem (`equation`) are implemented with the exception of the specific evaluation in the Gauss points, that must be defined in a method called `eval_equation` implemented in the derived class (`Liu` in this case) and that receives the `Gausspoint` where the integration will take place as a parameter. The schematic diagram of an equation resolution is shown in Fig. 3. In this particular case it is the Poisson equation. However, many different equations can be calculated in the same way.

It can be seen that the class that implements an specific element (`Liu`) passes the messages to the `Element` class, where all the common code was actually implemented. The only method that is executed in `Liu` is `eval_poisson`, which evaluates the integration in a specific Gauss point.

2.3 Implemented Elements

As an example of the flexibility of the developed model, four types of two-dimensional quadrilateral element with different features have been implemented.

Four-Node Isoparametric Element with Reduced Integration. A two-dimensional, quadrilateral element with four nodes proposed by [10] was implemented. This element uses selective reduced integration to avoid the volumetric and shear locking and to reduce the computational time needed. This element

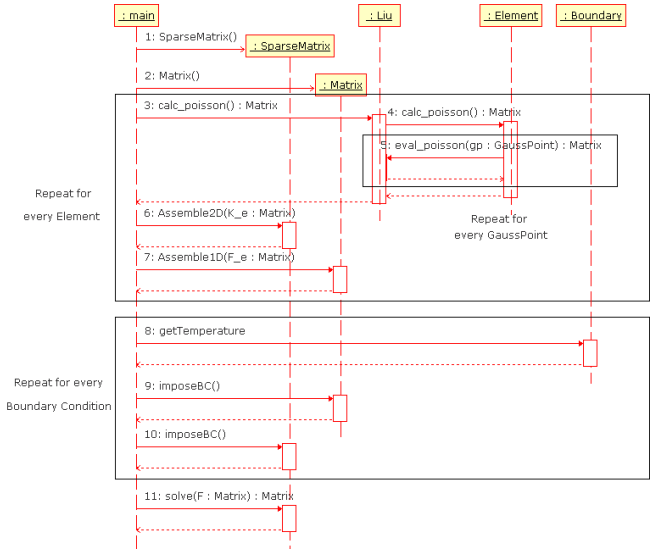

Fig. 3. Schematic diagram from the resolution of the Poisson equation.

was designed to be applied in elasto-plastic solid mechanic problems. It uses a Taylor expansion to compute the strain rate, that is split into its deviatoric and volumetric part. To diminish the volumetric locking, selective reduced integration is applied [5]. The elemental formulation asserts the diminishing of the volumetric locking, even if it is integrated at just one Gauss point. However, this is usually not enough in the case that a plastic deformation front should be accurately detected in an elasto-plastic problem. That is the main reason to integrate at two Gauss points.

Eight-Node Isoparametric Element. It is known that the classical quadrilateral four noded element does not pass the Brezzi-Babuška condition [11], which is necessary to assert the solution convergence and avoid collateral effects like *checkerboard* pressure distribution. That is why other elements were implemented, as the two-dimensional quadrilateral eight noded element, without the need of special characteristic.

Strain rate is computed in the standard way by means of the gradient matrix that includes the shape functions derivatives. No expansion in a Taylor series is made and the integration is performed in the usual full set of Gauss points related to the second order of interpolation inside the element [11].

Nine-Node Isoparametric Element. In order to increase the richness of the variety of available elements in the framework, a two-dimensional bi-quadratic nine-node isoparametric element [11] was implemented. This element presents a better behavior than the eight-node element presented above due to the presence

of the "bubble" function associated with the center node. Thus, the extra node results in a more precise but also a more expensive element.

Variable-Order Spectral Isoparametric Element. Another type of element implemented is a parametric spectral isoparametric element [12, 13]. In these elements the order of the interpolation functions is set by means of a parameter giving raise to a family of elements of variable order. The main characteristic in this family is that nodes are not uniformly distributed inside the element, but on a Lobatto grid. Besides, all the integrals are computed numerically using the Gauss-Lobatto-Legendre integration rule which uses those same points as integration points. In our case we have used spectral elements with order ranging from 4 to 20. The particular case of the spectral element of order 2 (biquadratic) corresponds to the classical nine-node element that uses the usual Gauss-Legendre integration rule.

3 Improvements for parallel execution

The `Domain` class was improved in order to read only the subdomain that should represent instead of the whole domain. In addition, a new class, called `Subdomain`, was implemented to deal with the multiplicity of mappings between internal and external DoF of the subdomain itself and also, of the subdomains that are related to it in a tree-structured communication [14].

The communication between a pair of nodes in the cluster is solved in a very simple way, from the point of view of the programmer who handles the instances. All the classes in the framework include the method `Save`, which stores the instance into an output stream. These same classes have a constructor, with an input stream as the parameter, which reads the instance from the stream and creates it in memory. No more methods than these are needed to move instances in a transparent way between computers in the cluster. The bytes located at the output stream are given to the MPI_Send command and the same is done with the bytes received by MPI_Receive, which are given to the constructor of the class.

The assemblage of the global stiffness matrix is modified in order to skip the external or internal nodes when necessary. Namely, the degrees of freedom are assembled in only one of the four global stiffness submatrices related to the Schur complement technique (see later subsection 4.3).

4 Mathematical Model

4.1 Mathematical Problem

As a test case we considered the following version of the Poisson problem [15] of finding the solution ϕ in a domain $\Omega = [0, 1] \times [0, 1]$ to the equation

$$-\nabla^2 \phi = f, \tag{1}$$

with homogeneous Dirichlet boundary conditions.

The right hand side function f in equation 1 was chosen such that the exact solution is

$$\phi = e^{xy} \sin(\pi x) \sin(\pi y),$$

which gives

$$\begin{aligned} f = - e^{xy} \left\{ \left(y^2 - 2\pi^2 + x^2\right) \left[\sin(\pi x) \sin(\pi y)\right] + \right. \\ \left. 2\pi \left[y \cos(\pi x) \sin(\pi y) + x \sin(\pi x) \cos(\pi y)\right] \right\}. \end{aligned}$$

This way, we knew a priori the exact solution of the problem to which we could compare the results obtained from our numerical simulations.

The partial differential problem of equation 1 can be reformulated in its weak form [16, 17] as: *find a function $\phi \in H$ such that:*

$$\iint_\Omega \boldsymbol{\nabla}\psi \cdot \boldsymbol{\nabla}\phi \, d\Omega = \iint_\Omega \psi \, f \, d\Omega \quad \forall \psi \in H. \tag{2}$$

This is the formulation required to implement the FEM to find an approximation of equation 1. In the remaining of the section we will describe the discretization procedure followed in order to arrive to the final system of equations by means of finite elements disregarding which type of element is used. In our implementation we have been working with the elements presented in section 2.3

4.2 Discretization

In the FEM the analysis domain is decomposed into subdomains called *elements*; in such a way that integral in equation 2 can be divided in several integrals over each element. The unknown functions are written in terms of interpolating functions which are particular of the type of element implemented. The collection of all the shape functions of the elements in a discretization form a basis of functions over the whole domain which generates a functional space the approximate solution will belong to.

In the general case, the unknown function ϕ is expressed inside an element as

$$\phi = h_i(r, s), \; \phi^i \qquad i = 1..N, \tag{3}$$

where h_i is the interpolation function associated with elemental node i, r and s are the coordinates in the *master or* natural square element with $-1 \leq r, s \leq 1$, ϕ^i is the value of the function ϕ in node i and N is total the number of nodes in the element. Here, and in the rest of the paper, we adopt the Einstein summation convention over repeated indices. Equation 3 can also be written in matrix form as

$$\phi = \mathbf{H}\,\boldsymbol{\Phi}^e,$$

where $\mathbf{H} = \begin{bmatrix} h_1 \; h_2 \cdots \; h_N \end{bmatrix}$ and $\boldsymbol{\Phi}^e = \begin{bmatrix} \phi^1 \; \phi^2 \cdots \; \phi^N \end{bmatrix}^T$. The spatial derivatives of the unknown function can be arranged in matrix form as

$$\boldsymbol{\nabla}\phi = \begin{bmatrix} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{bmatrix} = \mathbf{B}\,\boldsymbol{\Phi}^e,$$

where

$$\mathbf{B} = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_2}{\partial x} & \cdots & \frac{\partial h_N}{\partial x} \\ \frac{\partial h_1}{\partial y} & \frac{\partial h_2}{\partial y} & \cdots & \frac{\partial h_N}{\partial y} \end{bmatrix}.$$

This way, interpolating the test functions ψ such as the unknown ϕ and mapping the actual element in physical space to the natural element, the integral equation 2 can be split into elemental contributions such that the left hand side takes the form

$$\iint_{\Omega_e} \boldsymbol{\nabla}\psi \cdot \boldsymbol{\nabla}\phi \, d\Omega_e = \int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\Psi}^{eT} \, \mathbf{B}^T \, \mathbf{B} \, \boldsymbol{\Phi}^e J \, dr \, ds, \tag{4}$$

where J is the determinant of the jacobian matrix $\mathbf{J}$ of the transformation from the physical space coordinates (x, y) to the natural element coordinates (r, s). The matrix $\mathbf{J}$ is also used to calculate the elements of matrix $\mathbf{B}$ from the derivatives of the interpolation functions with respect to the natural coordinates. The right hand side of equation 2 can be written as

$$\iint_{\Omega} \psi \, f \, d\Omega = \int_{-1}^{1}\int_{-1}^{1} \boldsymbol{\Psi}^{eT} \, \mathbf{H}^T \, f \, J \, dr \, ds. \tag{5}$$

Since equation 2 must be true for every function in the space generated by the set of interpolation function we can chose the vector $\boldsymbol{\Psi}^e$ to be formed by all zeros except one element at a time getting as many equations as nodes in the element. Also, as the vector $\boldsymbol{\Phi}^e$ is independant of the coordinates we can take it out of the integrals of equations 4 and 5 giving rise to the elemental system of equations

$$\mathbf{K}^e \, \boldsymbol{\Phi}^e = \mathbf{F}^e, \tag{6}$$

where

$$\mathbf{K}^e = \int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^T \, \mathbf{B} \, J \, dr \, ds \quad \text{and} \quad \mathbf{F}^e = \int_{-1}^{1}\int_{-1}^{1} \mathbf{H}^T \, f \, J \, dr \, ds.$$

Matrices $\mathbf{K}^e$ and $\mathbf{F}^e$ are calculated by numerical integration at every element and then assembled into corresponding global matrices $\mathbf{K}$ and $\mathbf{F}$ according to the correspondence between local and global node numbering, arriving to the final system of equations

$$\mathbf{K} \, \boldsymbol{\Phi} = \mathbf{F}. \tag{7}$$

4.3 Substructuring by means of the Schur Complement Technique

The idea of the substructuring technique is to divide the problem of solving the equation 1 over the whole domain Ω into several sub-problems to be solved locally in every processing unit. Then, the contribution of every sub-problem to the global "stiffness" is calculated and the new (and much smaller) global problem is solved.

Mathematically this technique can be presented as follows. The global domain is to be divided into sub-domains composed by several elements each. Considering one of these sub-domains, its nodes (and correspondingly its degrees of freedom (DoF)) can be divided between those belonging only to elements of the current sub-domain, i.e. internal nodes, and those shared with elements in different sub-domains, i.e. external nodes. Considering the system in equation 7, obtained by the assemblage of the elements of the sub-domain, we can rearrange the equations and unknowns in a way that all the interior nodes appear altogether and the same for the exterior nodes. Then, indicating with the subscript e the set of exterior DoF and with i the set of the interior ones, we can rewrite equation 7 for the sub-domain s as

$$\begin{bmatrix} \mathbf{K}^s_{ee} & \mathbf{K}^s_{ei} \\ \mathbf{K}^s_{ie} & \mathbf{K}^s_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{\Phi}^s_e \\ \mathbf{\Phi}^s_i \end{bmatrix} = \begin{bmatrix} \mathbf{F}^s_e \\ \mathbf{F}^s_i \end{bmatrix}. \tag{8}$$

Now, we can obtain the *Schur complement* of block $\mathbf{K}^s_{ii}$ in matrix $\mathbf{K}^s$ to find the contribution of sub-domain s to the global problem stiffness $\mathbf{K}^s_{Sch} = \mathbf{K}^s_{ee} - \mathbf{K}^s_{ei}\,{\mathbf{K}^s_{ii}}^{-1}\,\mathbf{K}^s_{ie}$. The system of equation 8 can then be expressed only in terms of the exterior DoF as

$$\mathbf{K}^s_{Sch}\,\mathbf{\Phi}^s_e = \mathbf{F}^s_{Sch} \tag{9}$$

where $\mathbf{F}^s_{Sch} = \mathbf{F}^s_e - \mathbf{K}^s_{ei}\,{\mathbf{K}^s_{ii}}^{-1}\,\mathbf{F}^s_i$ is the corresponding load term.

The matrices $\mathbf{K}^s_{Sch}$ and vectors $\mathbf{F}^s_{Sch}$ of every sub-domain are assembled into global reduced matrix and vector to form a system of equations in terms of DoF only associated to nodes shared by elements in different sub-domains. Once this system is solved we can recover the interior nodes of each sub-domain from the second line of equation 8.

When applying this technique there are two stages of solving systems of equations whose matrices are sparse. One is the local solution leading to equation 9 and the recovering of the interior DoF. This solution is carried on concurrently for all the sub-domains locally in each processing unit. The second stage is global and only one system of equations is solved at a time. Due to this characteristic, different solvers could be used for each stage.

5 Numerical Experiments

In this first stage we tested our implementation solving the test case for different regular meshes composed by 8-node 2D quadrilateral elements. We focused on efficiency of data communication trying to evaluate how parallel FE problems could be efficiently implemented following the philosophy of the framework. To this end, we used the interface to PARDISO to perform the system of equations solution sequentially at subdominial as well as global level. In this case we did not make use of PARDISO capabilities of parallelization what we plan to study in detail when analyzing different solvers options.

We tested meshes composed by subdomains with different number of elements ranging from 100 elements per subdomain to 40000 elements per subdomain. The

number of subdomains in each mesh was varied from 1, meaning no subdivision, to 64 subdomains. We divided the solution procedure into stages in order to time each one in detail. These stages are:

1. Assemble of the elemental matrices and load vectors,
2. Factorization of $\mathbf{K}_{ii}^s$ from equation 8,
3. Computation of $\mathbf{K}_{Sch}^s$ and $\mathbf{F}_{Sch}^s$,
4. Reduction and assemble of $\mathbf{K}_{Sch}^s$ and $\mathbf{F}_{Sch}^s$ to form the global system of equations 7,
5. Solution of the reduced global system,
6. Broadcast of the external DoF solution and recovering of the internal DoF solution.

We evaluated the weak and strong scaling capabilities of our code. The total time as the number of subdomains increases for a fixed subdomain size is shown in figure 4.

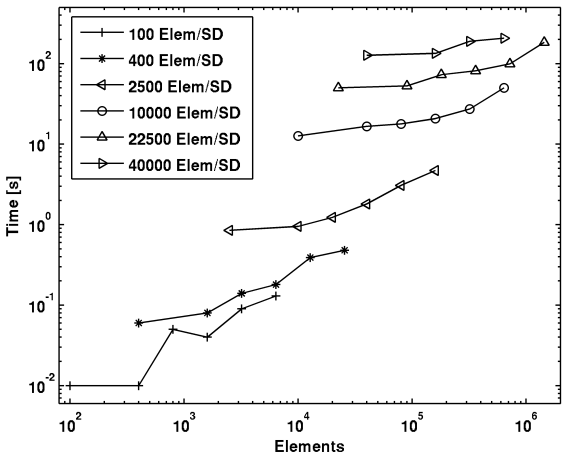

Fig. 4. Total time with respect to problem size for different subdomain sizes.

Weak scaling can be seen as moving along lines of constant number of elements per subdomain, the continuous lines in the figure, while strong scaling may be interpreted as moving along a vertical line keeping the problem total size fixed. This is done in figure 5 for a problem of 160000 elements in total. The dotted line in that figure represents a *linear speedup* situation where the time T_p for solving the same problem using p subdomains is

$$T_p = \frac{T_1}{p},$$

with T_1, the time for solving in 1 subdomain. As shown in the figure our code scales better than in the linear speedup situation. This is not the expected behavior, it could be explained since we made all the testing for this work using only direct sequential solvers for the systems of linear equations arising from the FEM. Those solvers, based on factorizations of the matrix followed by back and forward substitutions are typical algorithms with $\mathcal{O}(n^3)$ complexity. Thus, in this case, it is faster to solve several small problems than few bigger problems despite the cost of communications and global solution.

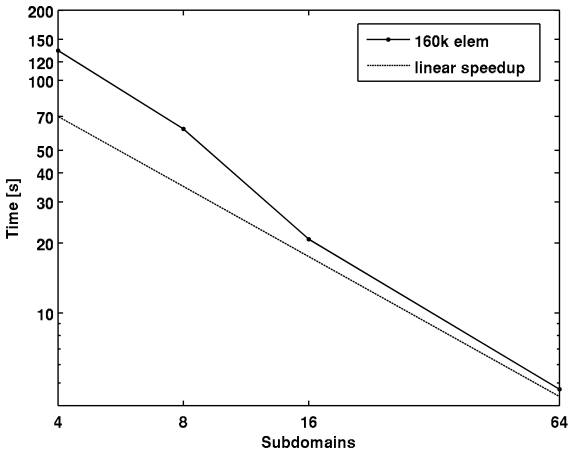

Fig. 5. Time elapsed for a constant problem size of 160000 elements.

In order to study the behavior of our code under weak scaling, i.e. when increasing the number of subdomains with fixed number of elements per subdomain, the speedup of a parallel application in such a situation is defined as

$$S = \frac{p\,T_1}{T_p},$$

where, for a given subdomain size, T_1 is the time required to solve the problem with no subdivision in 1 processor and T_p is the time required to do so in p subdomains, each in one processor. In figure 6 we show the speedup obtained for meshes with increasing number of subdomains in order to test the weak scaling capabilities of our implementation. In this case, linear speedup means that $S = p$ and one can solve p subdomains distributed in p processors in the same time that one solved 1 subdomain sequentially. This case would have resulted in horizontal lines of constant time in figure 4, and the increase in time means a loss in speedup. Linear speedup is represented by the dotted line in figure 6. From this figure two characteristics become evident: the speedup improves with the problem size and the speedup deteriorate as the number of subdomains increase.

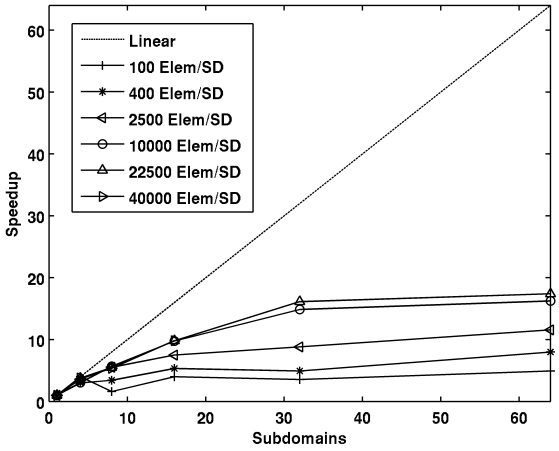

Fig. 6. Speedup for different subdomain sizes in term of the number of subdomains.

In order to explore the reasons for that behavior, in figures 7 and 8 we present measurements of the time elapsed in each stage of the computation with increasing number of subdomains for relatively small and large number of elements per subdomain, respectively. Figure 7 show that time is kept constant in the first 3 stages which seems logical as those imply purely local operations. On the other hand, the assemble of the global system of equations is done by means of a reduce-like operation with communication from every process to the headnode. Although this operation was coded as a tree-structured sending and partial assemble operation, the time increases with the number of subdomains. The most time consuming stage in this case is the solution of the global system of equations. As already said the solution is computed by a sequential solver which means that no parallelization is made at all. Considering that for the bigger cases the time for the system solution represents some 50 % of the total time it turns out clearly what the cause of the speedup loss is. The last stage, solving the interior nodes DoF, is negligible in all cases we studied.

Figure 8 shows essentially the same general figures; but in this case, as each subdomain is bigger, the proportion of time consumed in purely local operations is bigger giving rise to a better speedup. The most time consuming stage is the computation of $\mathbf{K}^s_{Sch}$ which is done by back and forward substitutions with several right hand sides. One point to be highlighted here is that the time of this stage exhibit some jumps for certain increases in the number of subdomains. As pointed out before this stage implies only local operations so those jumps can only be attributed to resource saturation problems. This behavior has appeared when solving other problems with high number of elements per subdomain while when this number is kept medium sized the behavior did not show up.

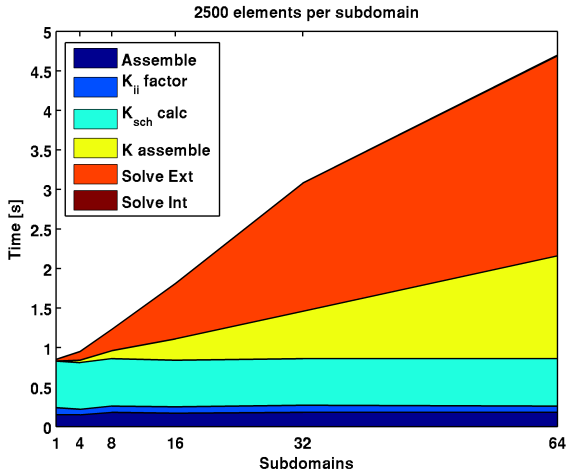

Fig. 7. Time of processing each stage with constant subdomain size of 2500 elements per subdomain.

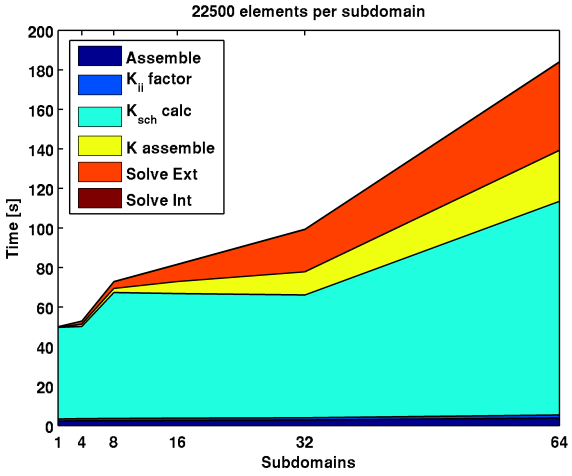

Fig. 8. Time of processing each stage with constant subdomain size of 22500 elements per subdomain.

6 Conclusions and a pathway to future improvements

The main objective of this work was to make a first approach to solve parallel FE problems based on the framework proposed in [3, 4]. This set of experiments and results will help us to identify where to put future efforts in order to improve the performance of this implementation. Considering the results exposed so far it becomes clear that the focus in future research should be on the solution of the system of equations, both solving of the system itself and what now is the assemble of the matrix and load term. All the intrinsic local stages and even the solution of internal DoF which require communication scale well so they should not be problematic, but for problems with increasing size the computation of the Schur complement matrix takes too much time compared with other stages.

Regarding the solution of the system of equations there are several steps to follow in order to achieve better scalability and performance. As said before, up to now we tested the code using the PARDISO interface only in a sequential way. The first alternative to be tested is to use a multifrontal parallel direct solver (MUMPS) for which the framework has an interface already available. We hope that the good scaling properties of MUMPS will help to improve the time and speedup of the system solving stage. Further improvement could be achieved using MUMPS capabilities to handle distributed matrices which would eliminate the need of communicating and assembling the global stiffness matrix of the reduced system.

Another line to investigate is the use of iterative solvers which are an frequently followed alternative when solving systems with large number of equations. They usually require much less communication as one can manage the system distributed over each processor. This way you do not need either to communicate and assemble the reduced system from each subdomain. The amount of communication in this case will be dictated by the number of iterations before converging to the solution. To reduce them, good preconditioning techniques should also be studied. Extra time savings could be achieved in this case as it will not be necessary to compute the Schur complement matrix explicitly since its effect on a vector could be evaluated in each step in a more economical way.

As the relative weight of the stages is dependent on the type of particular element used, we plan to compare the effect of different elements. Thanks to the versatility of the framework this can be done with almost no code modification.

Up to now we have tested our implementation only using perfectly regular meshes. To allow for more realistic simulations it has to be able to work with more general ones, in particular non regular meshes with, typically, unbalanced subdomains. We plan to test how this affects the performance of the code and what amount of imbalance is to be tolerated during the partitioning of the domain in order not to penalize the simulation process.

Acknowledgments

Part of this research was supported by founds made available by Project PICT 1581, Agencia Nacional de Promoción Científica y Tecnológica.

References

1. Modak, S., Sotelino, E.: An object-oriented programming framework for the parallel dynamic analysis of structures. Computers & Structures **80**(1) (2002) 77–84
2. Sonzogni, V., Yommi, A., Nigro, N., Storti, M.: A parallel finite element program on a Beowulf cluster. Advances in Engineering Software **33**(7-10) (2002) 427–443
3. Quinteros, J., Jacovkis, P.M., Ramos, V.A.: Diseño flexible y modular de modelos numéricos basados en elementos finitos. In Elaskar, S.A., Pilotta, E.A., Torres, G.A., eds.: Mecánica Computacional. Volume XXVI., Córdoba, Argentina, Asociación Argentina de Mecánica Computacional (Oct 2007) 1724–1740
4. Quinteros, J., Ramos, V.A., Jacovkis, P.M.: An elasto-visco-plastic model using the finite element method for crustal and lithospheric deformation. Journal of Geodynamics **48**(2) (2009) 83–94
5. Hughes, T.J.R.: Generalization of selective integration procedures to anisotropic and nonlinear media. International Journal for Numerical Methods for Engineering **15** (1980) 1413–1418
6. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK users' guide. Third edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
7. Schenk, O., Gaertner, K., Fichtner, W., Stricker, A.: Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. Future Generation Computer Systems **18** (2001) 69–78
8. Amestoy, P.R., Duff, I., L'Excellent, J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Computer Methods in Applied Mechanics and Engineering **184** (2000) 501–520
9. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Analysis and Applications **20**(3) (1999) 720–755
10. Liu, W.K., Hu, Y.K., Belytschko, T.: Multiple Quadrature underintegrated finite elements. International Journal for Numerical Methods in Engineering **37** (1994) 3263–3289
11. Bathe, K.J.: Finite element procedures. Prentice Hall, Englewood Cliffs, New Jersey, USA (1996)
12. Patera, A.T.: A spectral element method for fluid dynamics: laminar flow in a channel expansion. J. Comput. Phys. **54** (1984) 468–488
13. Karniadakis, G.E., Israeli, M., Orszag, S.A.: High-order splitting methods for the incompressible navier-stokes equations. J. Comput. Phys. **97** (1991) 414–443
14. Pacheco, P.: Parallel Programming With MPI. Morgan Kaufmann (October 1996)
15. Toselli, A., Widlund, O.: Domain Decomposition Methods - Algorithms and Theory. Volume 34 of Springer Series in Computational Mathematics. Springer (2004)
16. Becker, E.B., Carey, G.F., Oden, J.T.: Finite Elements: An Introduction. Volume 1 of Texas finite element series. Prentice Hall, Englewood Cliffs, New Jersey, USA (1981)
17. Carey, G.F., Oden, J.T.: Finite Elements: A Second Course. Volume 2 of Texas finite element series. Prentice Hall, Englewood Cliffs, New Jersey, USA (1983)