Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 3045-3061 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

GENERALIZED FINITE ELEMENT METHOD COMPUTATION: PARALLELIZATION USING PYTHON MULTIPROCESSING PACKAGE

Dorival Piedade Neto, Manoel Dênis Costa Ferreira, Sergio Persival Baroncini Proença

Structural Engineering Department, São Carlos Engineering School, São Paulo University, Av Trabalhador Sãocarlense, 400, São Carlos, SP, Brazil, dpiedade@sc.usp.br, mdenis@sc.usp.br, persival@sc.usp.br, http://www.set.eesc.usp.br/public/main/

Keywords: Generalized Finite Element Method; Parallel Processing; Multiprocessing; Python.

Abstract. The Generalized Finite Element Method (GFEM) is a partition of unity (PU) based approach that explores a mesh of elements to construct shape functions by enrichment of the PU using polynomial or special purpose functions. The shape functions are attached to a node, center of a domain ('cloud') defined by the elements sharing that node. In spite of the nodal aspect, it is possible to verify that the GFEM programming follows the same framework of the Finite Element Method (FEM) programming, i.e., the governing equations system of a problem is assembled by element contributions. Even though the required meshes in the GFEM are in general coarser than the ones used in the FEM, the computing of the enriched element stiffness matrix is usually time expensive. Nevertheless, the computing of each local matrix is totally independent and so parallelization of these computations is straightforward and can be naturally explored. The hereby presented article describes a simple tool for Python codes parallelization by means of its standard library package 'Multiprocessing', which provides high level functions for parallel programming. The parallelization of the GFEM local stiffness matrix computations for two dimensional analyses is described, showing that it requires just a few changes in the original sequential code. The performance of the parallelization tool when tested on different computer architectures are shown to be very attractive. One concludes that the use of Python and Multiprocessing package is an efficient alternative for parallel programming.

1 INTRODUCTION

Python is a very high level, multi paradigm programming language, presenting a clean and simple syntax. Even though it is a scripting language, and therefore is not as fast as compiled languages as FORTRAN and C, it is being adopted as a convenient solution for scientific computation, due to the existence of numerical libraries like NumPy (The SciPy Community, 2010) and SciPy (The SciPy Community, 2011). These libraries are mainly written in compiled languages and provide a convenient access to its numerical routines trough Python, turning it recommended for this kind of application, as they provide a way to associate Python’s natural flexibility to very powerful numerical solution algorithms.

The association of Python to these numerical libraries has been adopted as a programming tool to create a two-dimensional Generalized Finite Element Method (GFEM) code developed to be used in our scientific research, and proved to be very attractive. Even though the observed time performance was not as good as it could be if it was developed in FORTRAN or C, it was good enough for the aimed purpose. The sparse linear system solver provided by the ‘scipy.sparse’ module, and the matrix computation routines provided by NumPy are essential for the performance improvement.

Despite of this fact, a great amount of the computations are performed in pure-Python, like the ones needed to assemble the local stiffness matrices and local force vectors, and showed to be very time expensive. For instance, in the developed code, the equation system assembly takes around 90% of the total time spent, as it is shown in Figure 1.

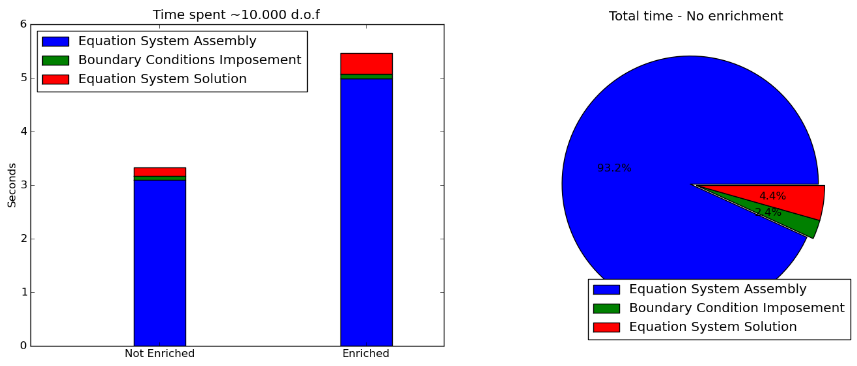


Figure 1: Contribution of each task group in the total processing time.

In Figure 1, the label ‘Not enriched’ stands for the conventional FEM, while ‘Enriched’ makes reference to the enriched solution attained by means of the local enrichment resource of the GFEM. It is important to note that, even though the meshes used in the GFEM are generally coarser, the GFEM is more time consuming than the Finite Element Method (FEM) for problems containing the same number of unknowns, here referred as degrees of freedom (d.o.f), thus inducing some concerns about the performance of the computational code.

Fortunately, the equation system assemblage time is mainly composed by the computing of the local stiffness matrix and local force vector of each element and these tasks can be performed independently. This characteristic is quite useful since it allows performing these computations at the same time, i.e., to compute them in parallel. Since nowadays most of the computers desktops present more than one processing core, one can take advantage of these resources by parallelizing this part of the code.

On the other hand, parallel programming is not a trivial task in most programming languages, and demands a great theoretical knowledge about the hardware architecture and good programming skills. However, Python presents a ‘user-friendly’ solution available in its Standard Library: the ‘Multiprocessing’ package.

Multiprocessing is a ‘threading’ process-based interface for parallel programming (Rossum, 2011), presenting high level tools that enables one to easily perform parallel computations. In fact, so far none of the authors had a previous experience with this kind of library or even with advanced parallel computing implementation, and in just a few hours we were able to parallelize the local stiffness matrix computations by performing just basic changes in the sequential code.

The main objective of this paper is to describe this simple parallel implementation and to present the conclusions about the resulting benefits based on the achieved speed up and on the reduction of the total processing time.

On what follows, the general GFEM characteristics are briefly described in Section 2, without getting into details about its formulation. Next, the main subject of this text is addressed: the Python’s Standard Library Multiprocessing Package and its application on the existing sequential code. It is shown that its use requires only small changes in the original code, with no loss to the code readability and clearness. Then a simple Solid Mechanics problem is stated in order to evaluate the computational performance achieved. The performance evaluation is conducted by focusing the speed up of the parallelized part of the program and the resulting gain in the total processing time spent to solve the proposed problem. Finally, some conclusions about the proposed parallelization strategy are posted.

2 THE GENERALIZED FINITE ELEMENT METHOD

The Generalized Finite Element Method, (Melenk and Babuška, 1996) and (Duarte and Oden, 1996), is based on the concept of enriched partition of unity (PU) (Babuška and Melenk, 1997). Basically, a set of functions defined in a certain domain constitutes a PU if its sum is equal to one for all points in the domain.

The GFEM explores a mesh of linear Lagrangian elements to provide the PU and to construct shape functions by enrichment of the partition using polynomial or special purpose functions. This shape functions can be used with a Galerkin method to find approximate solutions to boundary value problem as in the standard FEM. Actually, the same FEM programming framework can be followed in order to assemble the GFEM governing equations system.

Each node of the mesh is a vertex for a GFEM shape function. The shape function is then defined over a compact support region given by the area of the elements sharing the vertex node. This region can be referenced as a cloud. A schematic example of a cloud attached to a vertex node of a regular two dimensional mesh of quadrilateral elements is shown at Figure 2.

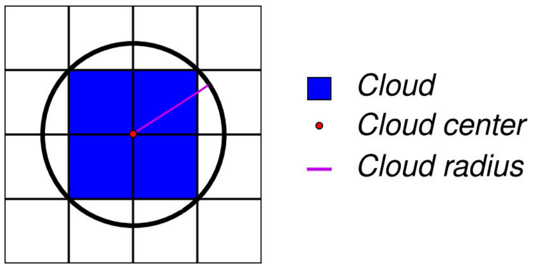


Figure 2: A GFEM mesh: a cloud, its vertex and radius.

The cloud radius depicted in Figure 2 is defined as the radius of the circle centered in the vertex node and circumscribing the cloud. Its value is useful to preserve dimensional consistency to the enrichment functions, as it will be discussed later in this Section.

Just for comparison with the GFEM features, it is interesting to remind that the FEM element approximation of a given field $u^{aprox.}$ is given by

$$u^{aprox.} = \sum_{i=1}^{n_{el}} \phi_i u_i , \tag{1}$$

in which ϕ_i are the PU function, and u_i is the interpolated field value in the i^th^ of the n_{el} element's nodes, also named degree of freedom.

Once the Galerkin method is adopted to formulate the boundary value problem, the global equation system is assembled by the contributions of the element stiffness matrices, each one computed by the following expression

$$\boldsymbol{K}_l = \int_{\Omega_e} \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B} \, d\Omega . \tag{2}$$

In the relation above $\boldsymbol{B}$ is the matrix of derivative operators that relates the strain tensor $\boldsymbol{\varepsilon}$ to the displacement field and $\boldsymbol{D}$ is the constitutive stiffness matrix relating the stress tensor $\boldsymbol{\sigma}$ to the $\boldsymbol{\varepsilon}$ tensor. This adopted symbology is classical in the FEM literature (Zienkiewicz and Taylor, 1991). It is worth to note that in the developed parallelized code, a plane linear elastic model was assumed. The mesh elements are four nodes isoparametric quadrilaterals, since its shape functions are in accordance with the definition of a PU.

In the GFEM, the approximation results from the product of the regular PU ϕ_i and a set $\boldsymbol{L}$ of n_{enr} enrichment functions L^{enr}. By definition, the first component of this set is equal to the unity. Thus, the regular GFEM approximation preserves the FEM fields while including additional enriched interpolation terms, resulting in

$$u^{aprox.} = \left(\sum_{i=1}^{n_c} \phi_i u_i \right) \boldsymbol{L} = \sum_{i=1}^{n_c} \phi_i u_i + \sum_{i=1}^{n_c} \sum_{j=2}^{n_{enr}} \phi_i L_j^{enr.} u_{ij} = \underbrace{\sum_{i=1}^{n_{enr}} \phi_i u_i}_{regular\ interpolation} + \underbrace{\sum_{i=1}^{n_c} \sum_{j=2}^{n_{enr}} \phi_{ij}^{enr.} u_{ij}}_{enriched\ interpolation} . \tag{3}$$

In the previous relation, n_c is the number of clouds and u_{ij} is the additional nodal parameters in correspondence to each one of the enrichment function.

In order to explore a similar program framework as presented by FEM, an element by element systematic assembling procedure can be followed. Accordingly, to compute the local stiffness matrix, one applies the same expression (2). However, in the GFEM, the $\boldsymbol{B}$ matrix order is dependent on the enrichment set $\boldsymbol{L}$ adopted. Then, the resulting local matrix contains more terms than the original one for no enrichment status, see Figure 3. Actually, additional lines and columns appear in correspondence to the new nodal parameters introduced by the enrichment fields. The same happens to the nodal force vector of the element.

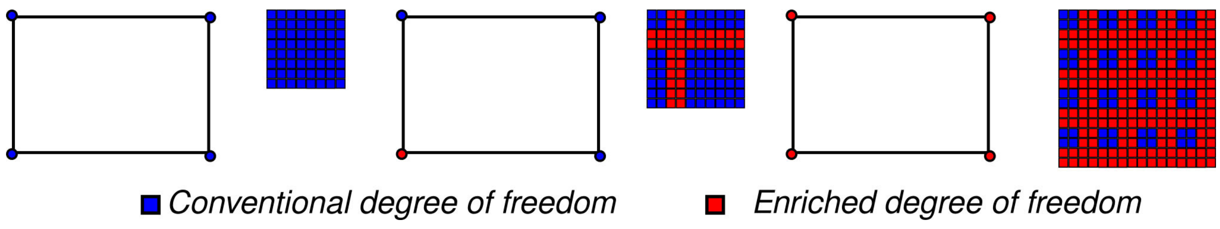


Figure 3: Conventional element matrix and a scheme of the resulting additional lines and columns on its enriched local stiffness matrix.

Even though the required meshes used in the GFEM are in general coarser than the ones used in the FEM, the computing of the enriched element stiffness matrix is usually more time expensive, as was already observed in Figure 1. Furthermore, as the resulting coefficient matrix is less sparse than the one in the FEM, also the solution of the global equation system is more expensive.

The enrichment functions used are polynomial function on the horizontal and vertical coordinates *x* and *y*. These functions are defined exploring the cloud radius already defined, here denoted by *h*, in order to preserve system dimensionality. Assuming that x_0 and y_0 represents the coordinates of the vertex node, the set of enrichment functions ***L*** employed in this work is given as

$$\boldsymbol{L} = \left\{ 1 \quad (x - x_0)/h \quad (y - y_0)/h \right\}. \tag{4}$$

3 PARALLEL PROGRAMMING

Generally the main objective of parallel processing is to reduce the time spent to process an application. An important concept in this context is the speed up of the parallelized code, defined as the relation between the time T_1 spent to process the sequential code, and the one spent to process it in parallel, using n_p processor units. The speed up *S* is then given by

$$S = T_1 / T_{np} . \tag{5}$$

It is important to mention that T_{np} includes the time spent in process intercommunication, so one can not expect the theoretical speed up values when testing the efficiency of a parallelization, i.e, hardly one would observe a $S=2$ for $n_p=2$, for instance.

In general, the parallelization of a code fragment is a challenging task and a good understanding about the hardware being used. One of the main objectives of this article is to show that some difficulties faced in parallel programming are minimized when using Python, especially if one choose to employ the Multiprocessing package. On what follows, a brief description of this package is given, making use of simple examples to illustrate its 'user-friendly' character. Then, the code fragments parallelized are shown and briefly commented, proving that indeed only very small changes are necessary in order to make it work.

3.1 Python's 'Multiprocessing' package

Multiprocessing is a high-level package for parallel programming available in Python Standard Library since its version 2.6. Presenting a similar structure to the one found in Python's threading module, the Multiprocessing package allows the programmer to access the various processors of a machine using sub processes instead of threads (Rossum, 2011). Even though it runs both on Unix and Windows, it presents a better performance on Linux, especially when the code demands to share state between processes. Within this subsection it will be presented some simple examples of the usage of this package, based on the online documentation (Rossum, 2011).

In order to create a simple process, the Multiprocessing package contains the Process class responsible for the behavior of objects of type Process. A simple example creating a process is depicted in Figure 4.

```
from multiprocessing import Process                                    # Importing Process module of the
Multiprocessing package

def process_routine(name):                                             # Defining the routine to be
                                                                       # executed by the process
    print 'Hello %s!!!' % name

if __name__ == '__main__':
    proc = Process(target = process_routine, args = ('World',))       # Creating the process 'proc'
    proc.start()                                                       # Starting the process 'proc'
    proc.join()                                                        # Waiting for the end of
                                                                       # the process 'proc'
```

Figure 4: Example of a simple process call using multiprocessing

As it can be observed, the creation of the process is made in only one code line, in which the name of the target function and its arguments are indicated. The process is then started using the function start() over the process object. The function join() indicates that the interpreter must wait until the end of the task sent to the process to continue to run the remaining code.

In order to promote communication between processes, the package presents two forms of communication channel: Queues and Pipes. Herein the usage of these features will not be shown since they were not used in the parallelization of the GFEM code. In fact, even though queue seems to be an efficient way to return the matrix and vector computed inside the process, we have found that it failed to manage great amounts of data, which is the characteristic of our parallelized code.

The Multiprocessing package has all the synchronization primitives of the threading module, in spite of the reduced need for synchronization primitives in a multi-process program if compared to a multithreaded program.

When using multiple processes, it is usually best to avoid using shared state as far as possible. However, if the use of shared data is required, the package provides a couple of ways of doing so: Value, Array and Manager. The main and most general way to work with shared data is the manager. Managers provide a way to create data which can be shared between different processes. A manager object controls a server process which manages shared objects.

Figure 5 shows a simple code in which a manager list is used by 6 different processes.

```
from multiprocessing import Process, Manager                 # Importing Process and Manager modules
                                                             # of the Multiprocessing package

def manager_routine(list, i_index, j_index):                 # Defining the function to be executed
    for i in range(i_index, j_index, 1):
        list[i] = list[i] * list[i]

if __name__ == '__main__':
    manager = Manager()                                      # Creating an object of Mananger type
    list = manager.list(range(1000))                         # Creating the list of objects on which
                                                             # to run the routine

    n_cpu = 6                                                # Defining the number of processors
    step = (len(list) / n_cpu) + 1
    procs = []
    for i in range(n_cpu):                                   # Creating and initializing processes
        if (i <> n_cpu-1):
            i_index = i*step
            j_index = (i+1)*step
            procs.append(Process(target = manager_routine, args = (list, i_index, j_index,)))
        else:
            procs.append(Process(target = manager_routine, args = (list, i*step,len(list),)))
        procs[i].start()

    for proc in procs:                                       # Waiting for the end of the processes
        proc.join()

    print list                                               # Printing the results
```

Figure 5: Example of parallelization on computations over items of a list using Multiprocessing's Manager.

Figure 6 indicates the scheme of parallelization adopted for the local stiffness matrices and local force vector using manager to get back the computed data. A better description of the parallelization of the GFEM code using a manager is given in the next sub section.

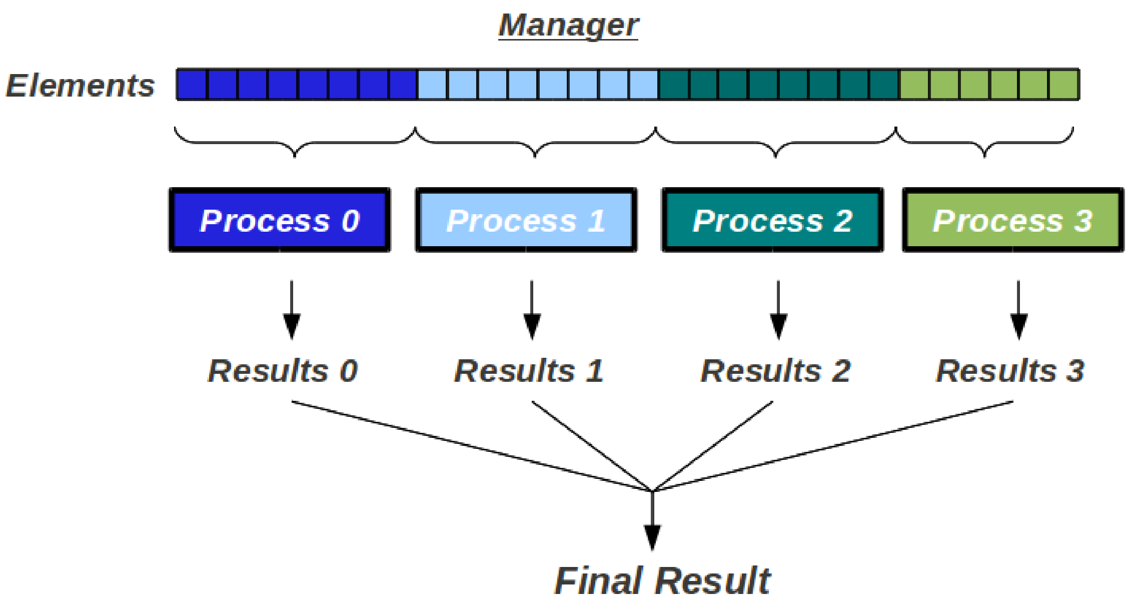
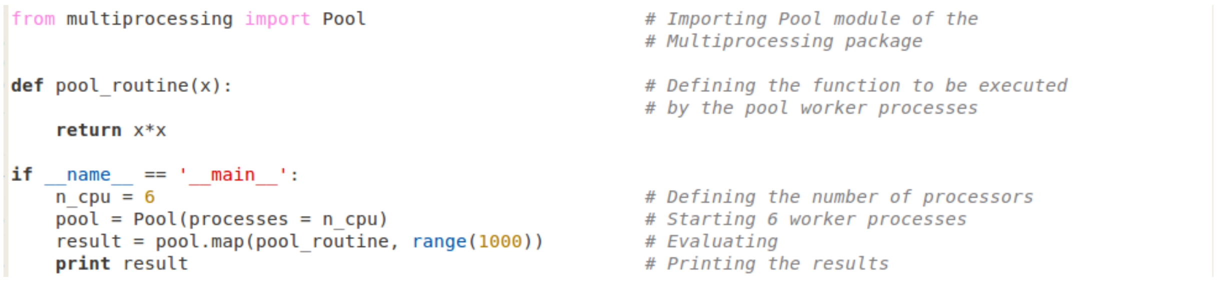


Figure 6: Scheme of the local stiffness matrix and force vector computation using Multiprocessing's Manager.

The simplest way to work with multiple processes using the Multiprocessing package is through the class Pool. A pool is a collection of processes kept ready to be used depending on the demand of their 'service'. Pools offer a good solution in situations in which the cost of initializing a class instance is high, and so, calling the constructor and destructor on demand would be inefficient. Figure 7 indicates a simple code example performing a function over a list of a thousand numbers, making use of pool.map.



```
from multiprocessing import Pool                        # Importing Pool module of the
                                                        # Multiprocessing package

def pool_routine(x):                                    # Defining the function to be executed
                                                        # by the pool worker processes
    return x*x

if __name__ == '__main__':
    n_cpu = 6                                           # Defining the number of processors
    pool = Pool(processes = n_cpu)                      # Starting 6 worker processes
    result = pool.map(pool_routine, range(1000))        # Evaluating
    print result                                        # Printing the results
```

Figure 7: Example of parallelization on computations over items of a list using Multiprocessing's Pool.

As it can be observed, the pool object is constructed using n_cpu processes, which controls a pool of worker processes to which jobs can be submitted. Then a function and a list of arguments are given to this pool of workers, which returns the results in the list 'result'. It is important to mention that Multiprocessing Pool supports asynchronous results with timeouts and callbacks and has a parallel map implementation. Figure 8 indicates a simple scheme of the parallelization of the GFEM code using a pool. A more detailed description of this approach is given on the next subsection.

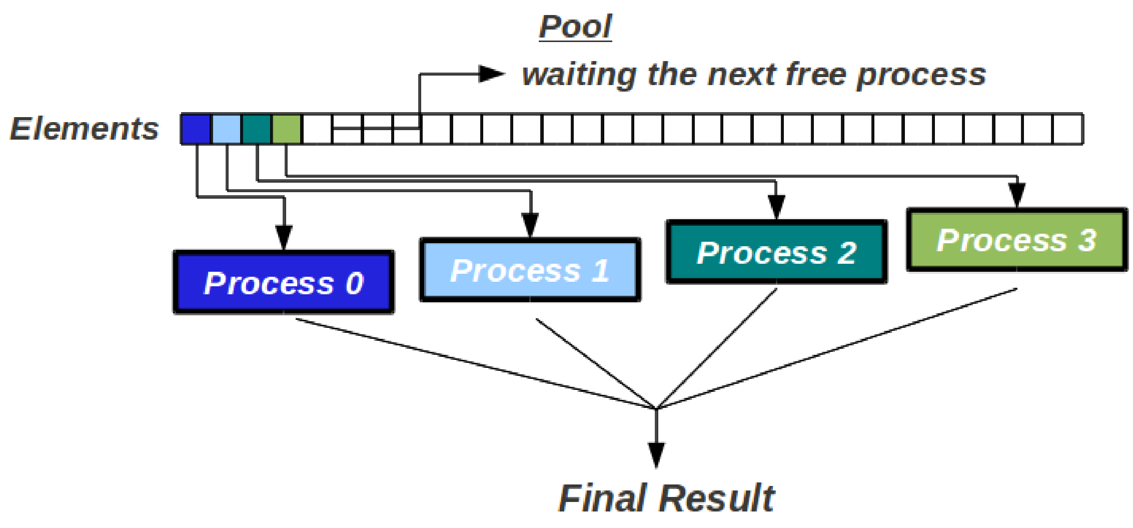
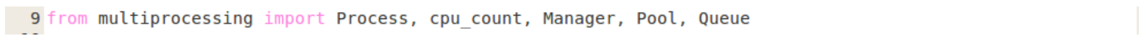


Figure 8: Scheme of the local stiffness matrix and force vector computation using Multiprocessing's Pool.

3.2 Code parallelization

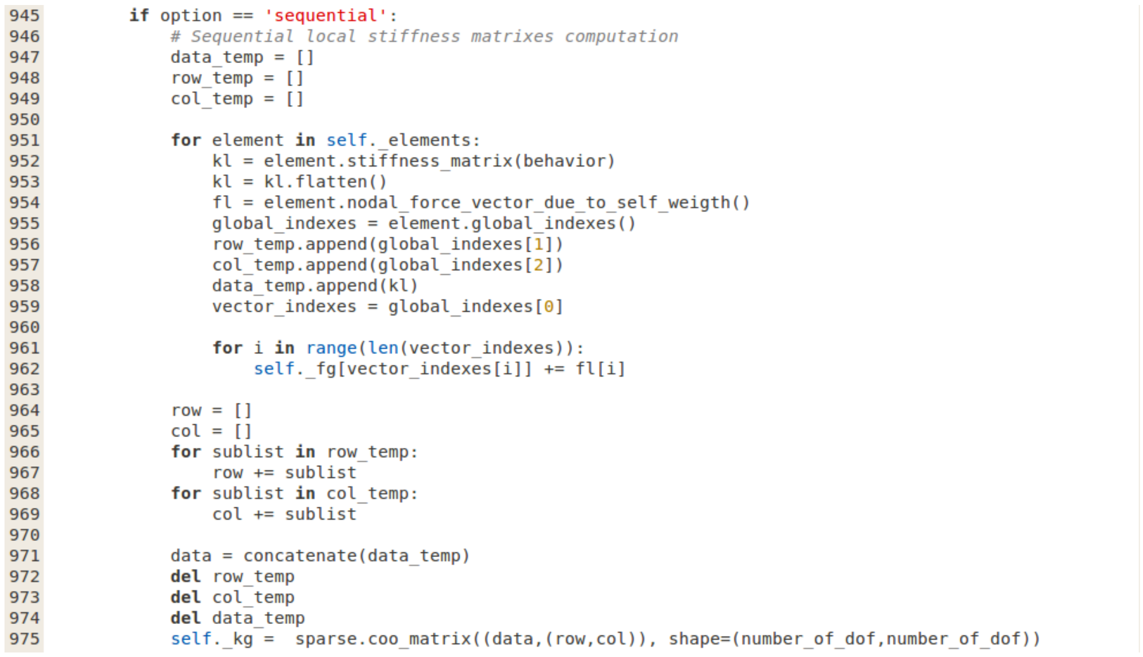
The parallelized program is an object-oriented code aimed to develop a full non-linear GFEM analysis framework. In order to introduce parallelism, all the code modifications are made in the module defining the main application class, which is the Structural_problem class. This class manages all other classes (like node, element and so on) in order to generate a GFEM model, assemble the global equation system and solve it. The first change in the refereed module is to insert a line to import the multiprocessing package in the parallelized code fragments. This import statement is depicted in Figure 9.



```
from multiprocessing import Process, cpu_count, Manager, Pool, Queue
```

Figure 9: Code line importing the used issues of multiprocessing.

The original sequential code fragment to be parallelized is then put into a conditional construction. If the user choice is the option 'sequential', the original code is the executed. It is presented in Figure 10.



```
945         if option == 'sequential':
946             # Sequential local stiffness matrixes computation
947             data_temp = []
948             row_temp = []
949             col_temp = []
950 
951             for element in self._elements:
952                 kl = element.stiffness_matrix(behavior)
953                 kl = kl.flatten()
954                 fl = element.nodal_force_vector_due_to_self_weigth()
955                 global_indexes = element.global_indexes()
956                 row_temp.append(global_indexes[1])
957                 col_temp.append(global_indexes[2])
958                 data_temp.append(kl)
959                 vector_indexes = global_indexes[0]
960 
961                 for i in range(len(vector_indexes)):
962                     self._fg[vector_indexes[i]] += fl[i]
963 
964             row = []
965             col = []
966             for sublist in row_temp:
967                 row += sublist
968             for sublist in col_temp:
969                 col += sublist
970 
971             data = concatenate(data_temp)
972             del row_temp
973             del col_temp
974             del data_temp
975             self._kg =  sparse.coo_matrix((data,(row,col)), shape=(number_of_dof,number_of_dof))
```

Figure 10: The original sequential sparse equation system assemblage.

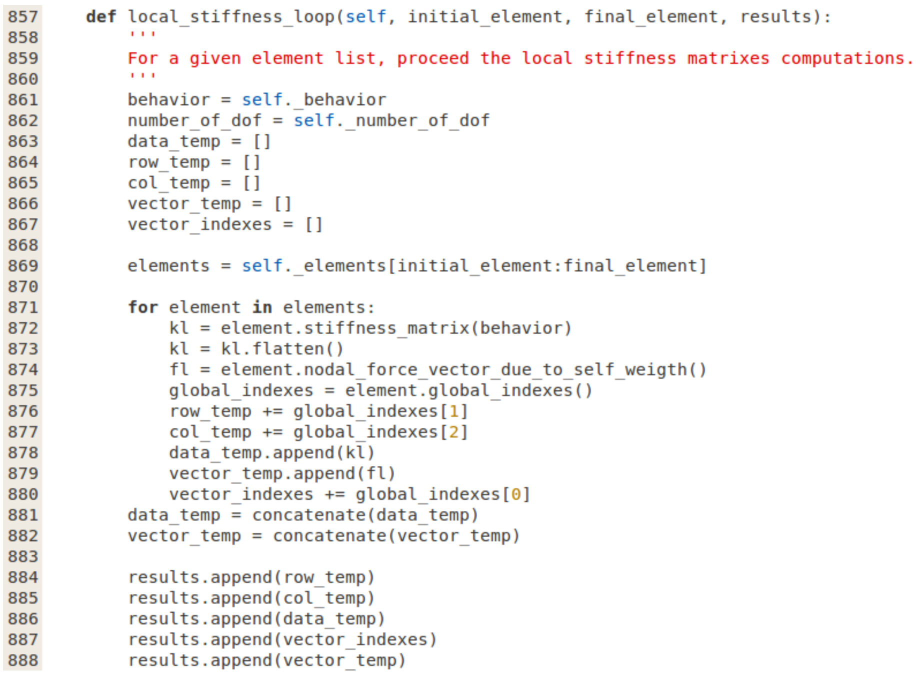
The fragment code to be parallelized is the one relative to the loop that begins in line 951 (Figure 10). As this code is within a method of the Structural_problem class, 'self' refers to the Structural_problem instance, and self._elements is the list containing all its elements. The variable 'element' is assigned for each one of these elements and then the methods to return the stiffness matrix, the force vector due to self weight and the global indices are invoked from the Element class. The global indices correspond to the positions of the global equation system in which the computed terms must be added. The 'for' loop in line 961 sums the local force vector terms in the global system vector self._fg. The code from lines 964 to 974 are used to concatenate the rows and columns indexes in to single lists, and the computed stiffness matrix terms into a single NumPy array. This is the more efficient input form we have found to build the SciPy sparse matrix self._kg, indicated in the line 975 of Figure 10.

The second option is to perform a parallel computation of the local stiffness matrix and local force vector using Multiprocessing's Manager. In the line 979 of Figure 11 a Manager object is instantiated in the 'manager' variable.

```
977          elif option == 'manager':
978              # Parallel local stiffness matrixes computation using multiprocessing module's manager
979              manager = Manager()
980              procs = []
981              procs_return = []
982              for proc in range(self._number_of_cpus):
983                  initial_element = proc * (len(self._elements) / self._number_of_cpus )
984                  if (proc == (self._number_of_cpus - 1)):
985                      final_element = len(self._elements)
986                  else:
987                      final_element = (proc + 1) * (len(self._elements) / self._number_of_cpus )
988                  procs_return.append(manager.list([]))
989                  procs.append(Process(target=self.local_stiffness_loop, args = (initial_element,
990                                                              final_element,procs_return[proc])))
991              for proc in procs:
992                  proc.start()
993
994              for proc in procs:
995                  proc.join()
996
997              #Matrixes lists
998              row = []
999              col = []
1000             data = []
1001             vector_index = []
1002             vector_data = []
1003
1004             data_temp = []
1005             vector_temp = []
1006             for proc_return_item in procs_return:
1007                 row += proc_return_item[0]
1008                 col += proc_return_item[1]
1009                 data_temp.append(proc_return_item[2])
1010                 vector_index += proc_return_item[3]
1011                 vector_temp.append(proc_return_item[4])
1012             data = concatenate(data_temp)
1013             vector_data = concatenate(vector_temp)
1014
1015             for i in range(len(vector_index)):
1016                 self._fg[vector_index[i]] += vector_data[i]
1017
1018             self._kg =  sparse.coo_matrix((data,(row,col)), shape=(number_of_dof,number_of_dof))
1019             del vector_index
1020             del vector_data
1021             del data_temp
1022             del vector_temp
```

Figure 11: Equation system assemblage paralellization using Multiprocessing’s Manager.

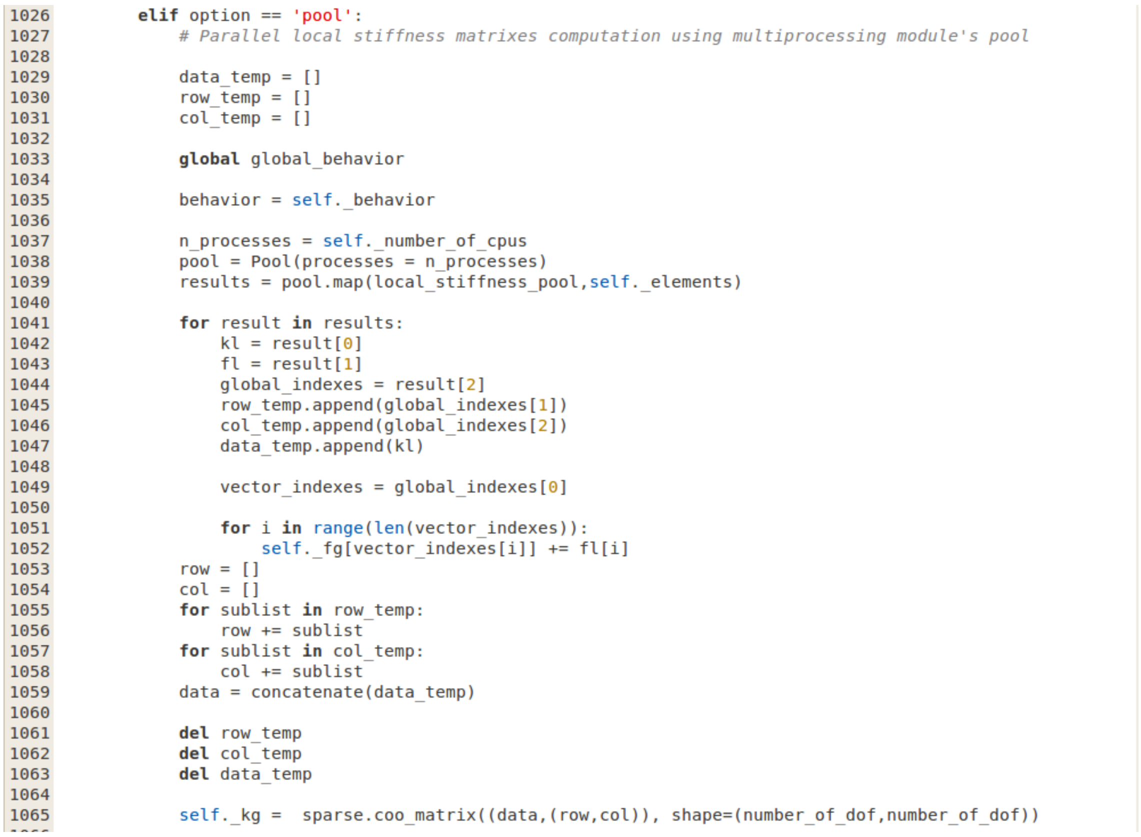
In Figure 11 a list ‘procs’ is filled with Process objects (see lines 989-990). The ‘target’ argument specifies the function to be called, in this case the ‘local_stiffness_loop’, indicated in Figure 12. Its arguments are the number of the initial and the final element to compute in the given process, and the list in which the results of these elements must be returned. These lists are manager.lists stored in the vector ‘procs_return’. The processes in the ‘procs’ list are started in line 992. They are run in parallel until all of them finish (line 995). The following lines in Figure 11 (997 to 1022) are a straightforward adaptation of the tasks necessary to build the equation system sparse matrix.



```python
    def local_stiffness_loop(self, initial_element, final_element, results):
        '''
        For a given element list, proceed the local stiffness matrixes computations.
        '''
        behavior = self._behavior
        number_of_dof = self._number_of_dof
        data_temp = []
        row_temp = []
        col_temp = []
        vector_temp = []
        vector_indexes = []

        elements = self._elements[initial_element:final_element]

        for element in elements:
            kl = element.stiffness_matrix(behavior)
            kl = kl.flatten()
            fl = element.nodal_force_vector_due_to_self_weigth()
            global_indexes = element.global_indexes()
            row_temp += global_indexes[1]
            col_temp += global_indexes[2]
            data_temp.append(kl)
            vector_temp.append(fl)
            vector_indexes += global_indexes[0]
        data_temp = concatenate(data_temp)
        vector_temp = concatenate(vector_temp)

        results.append(row_temp)
        results.append(col_temp)
        results.append(data_temp)
        results.append(vector_indexes)
        results.append(vector_temp)
```

Figure 12: Function to perform the computations on a given set of elements within a process.

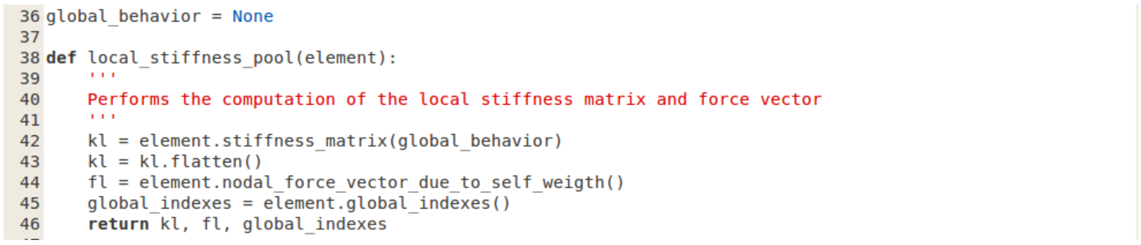
Finally, the last option that the user can choose is to perform the computation in parallel using a Multiprocessing Pool. The Pool object, containing n processes, is instantiated in line 1038 of Figure 13.



```python
1026        elif option == 'pool':
1027            # Parallel local stiffness matrixes computation using multiprocessing module's pool
1028
1029            data_temp = []
1030            row_temp = []
1031            col_temp = []
1032
1033            global global_behavior
1034
1035            behavior = self._behavior
1036
1037            n_processes = self._number_of_cpus
1038            pool = Pool(processes = n_processes)
1039            results = pool.map(local_stiffness_pool,self._elements)
1040
1041            for result in results:
1042                kl = result[0]
1043                fl = result[1]
1044                global_indexes = result[2]
1045                row_temp.append(global_indexes[1])
1046                col_temp.append(global_indexes[2])
1047                data_temp.append(kl)
1048
1049                vector_indexes = global_indexes[0]
1050
1051                for i in range(len(vector_indexes)):
1052                    self._fg[vector_indexes[i]] += fl[i]
1053            row = []
1054            col = []
1055            for sublist in row_temp:
1056                row += sublist
1057            for sublist in col_temp:
1058                col += sublist
1059            data = concatenate(data_temp)
1060
1061            del row_temp
1062            del col_temp
1063            del data_temp
1064
1065            self._kg =  sparse.coo_matrix((data,(row,col)), shape=(number_of_dof,number_of_dof))
```

Figure 13: Equation system assembly parallelization using Multiprocessing's Pool.

The use of a pool is easier than the use of a manager, since it only takes one line of code (line 1039 of Figure 13) to call the pool of workers to invoke the ‘local_stiffness_pool’ function, which is shown in Figure 14. The only requirement of this tool is that this function must be defined out of the class, so that’s the reason why it was stated in the first lines of the structural_problem module (from line 38 to 46) and that was required the global variable global_behavior (see lines 1033 of Figure 13 and 36 of Figure 14).



```
36 global_behavior = None
37
38 def local_stiffness_pool(element):
39     '''
40     Performs the computation of the local stiffness matrix and force vector
41     '''
42     kl = element.stiffness_matrix(global_behavior)
43     kl = kl.flatten()
44     fl = element.nodal_force_vector_due_to_self_weigth()
45     global_indexes = element.global_indexes()
46     return kl, fl, global_indexes
```

Figure 14: Function to perform the computations on a single element sent to a pool working process.

The rest of the code of the pool implementation (from lines 1041 to 1065 of Figure 13) is identical to the sequential code.

4 PERFORMANCE EVALUATION

In order to evaluate the computational performance of the resulting parallelized code, a two-dimensional linear elastic problem was stated. It consists of a 5.0 side and unit thickness square solid (dimensionless). The elastic material parameters are: Young modulus E = 1,000.0 and Poisson ratio υ = 0.3. The upper boundary surface is submitted to a distributed load q =1.0. Dirichlet boundary conditions constraining the horizontal and vertical displacements are prescribed as null at the lower boundary surface of the solid. Finally, a plane stress behavior was assumed.

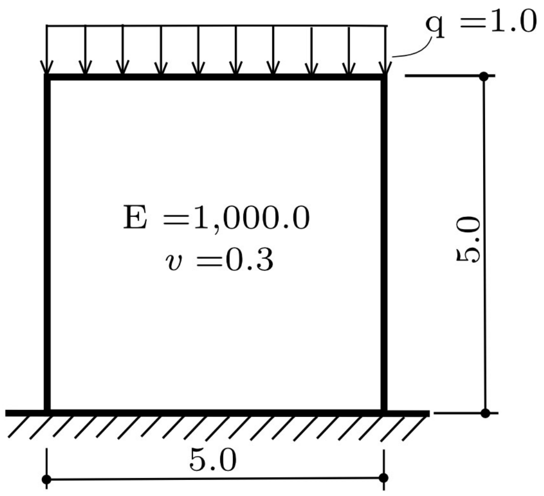


Figure 15: Two-dimensional problem stated to evaluate the computational performance.

Even though it represents a very simple structural problem, the time spent to compute and solve the equation system is similar to any other linear problem. So, once the main aim of this example is to evaluate the efficiency of Python’s Multiprocessing Package, its simple geometry and boundary conditions is very attractive to perform these tests.

The first test was performed in an Intel Core i7 x980 running at 3.33 GHz and 24 GB

RAM, under a GNU/Linux Ubuntu 11.04 64 bits. The attained speed ups for the parallelized part of the code, for not enriched meshes containing around 10,000, 50,000 and 120,000 degrees of freedom, are shown in Figure 16.

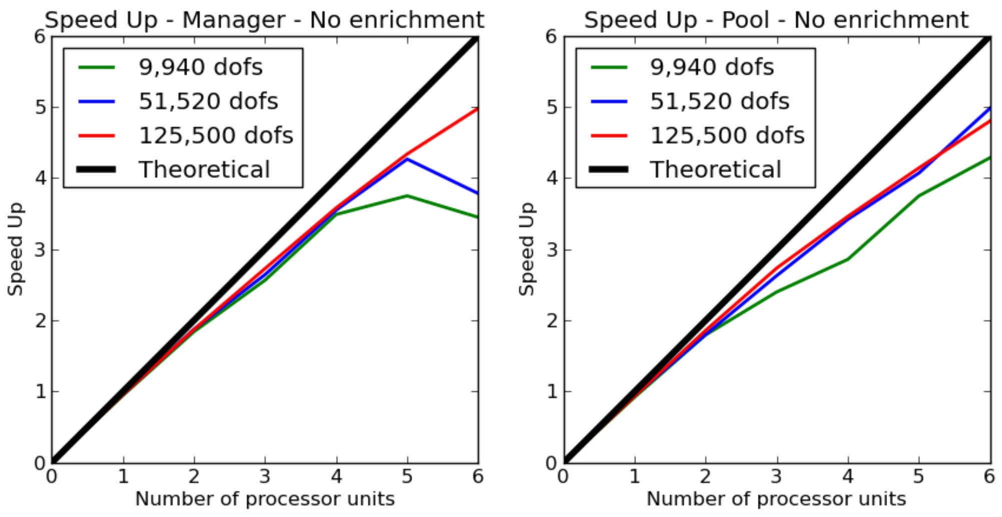


Figure 16: Speed up achieved - Not enriched - 1 to 6 processors.

The performance results for the GFEM using the enrichments on all nodes (except the ones in which Dirichlet Boundary Condition was applied), for meshes containing around the same number of degrees of freedom tested for the not enriched case, are depicted in Figure 17.

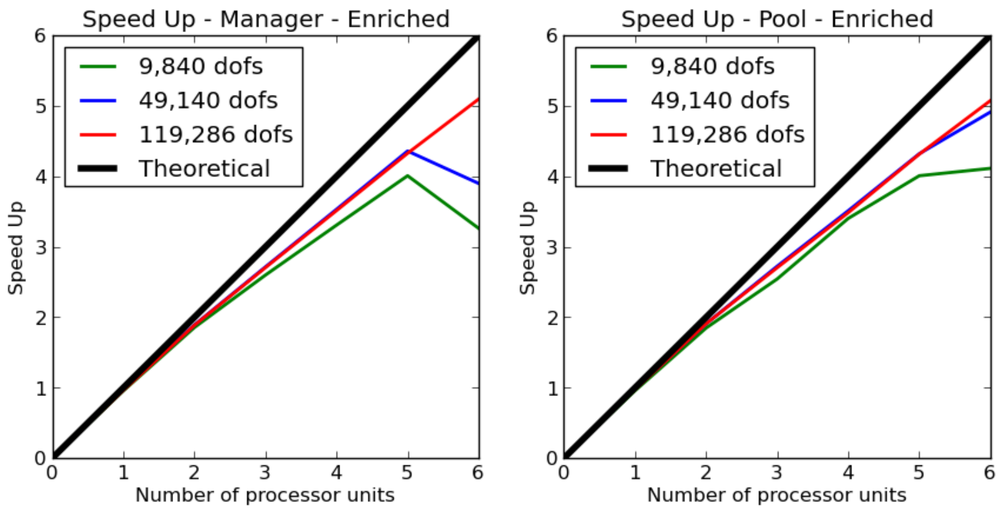


Figure 17: Speed up achieved - Enriched - 1 to 6 processors

The same meshes were also tested in a Windows 7 64bits Operating System, but the two parallelization approaches (using manager and pool) failed, resulting in a total processing time higher than the sequential one. These results are not shown here.

The same meshes were also tested in a cluster node containing 12 processing cores Intel Xeon X5660 running at 2.80 GHz, and 48 GB RAM. The Operating System is also a GNU/Linux Ubuntu 11.04 64 bits. The achieved results, considering not enriched and enriched cases, are shown, respectively, in Figure 18 and Figure 19.

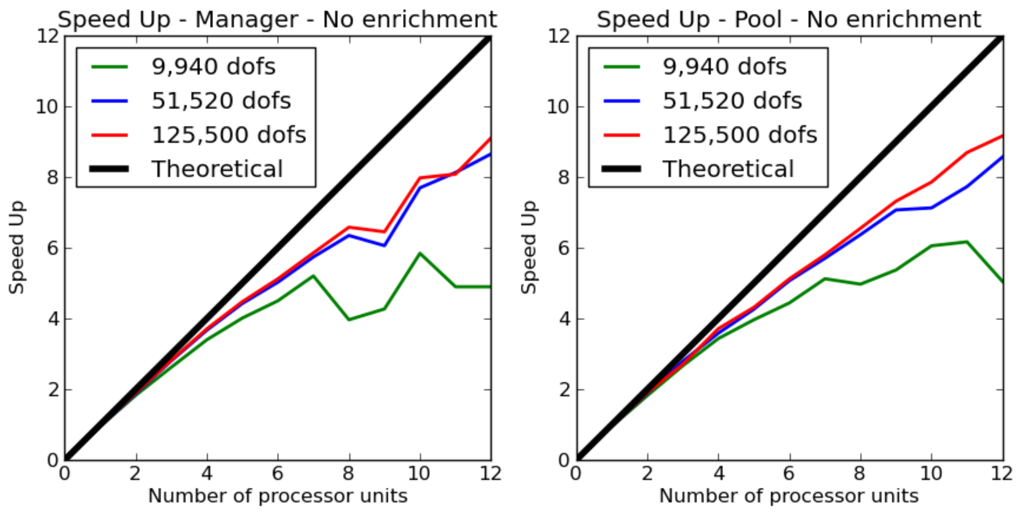


Figure 18: Speed up achieved - Not enriched - 1 to 12 processors.

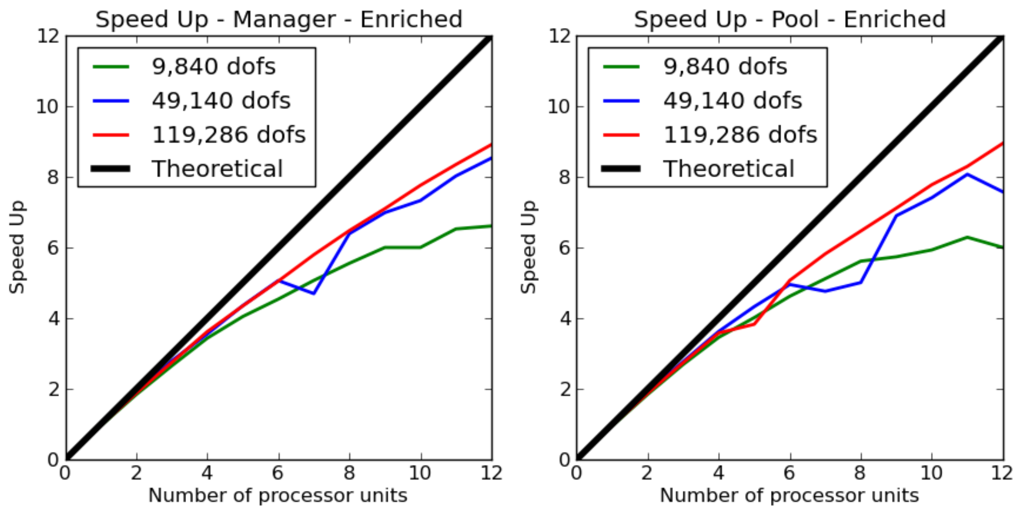


Figure 19: Speed up achieved - Enriched - 1 to 12 processors.

As it can be observed, the speed up results seems to be not so stable for the coarser meshes, when more than 6 processing cores were used. For these cases, the time spent by the parallelized computation is very short. Taking into account that the calling of each process demands a considerable amount of time, these non monotonic behavior are then explained, also indicating that the parallelization benefits depends upon the amount of data in computation.

The total processing time, including also the sequential parts of the computations, are depicted in Figure 20, Figure 21 and Figure 22, respectively for the meshes containing around 10,000, 50,000 and 120,000 degrees of freedom.

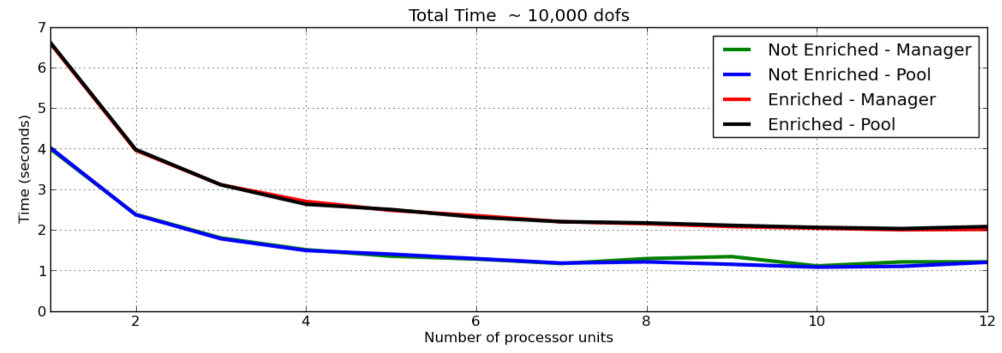


Figure 20: Total solution time for meshes containing around 10,000 degrees of freedom

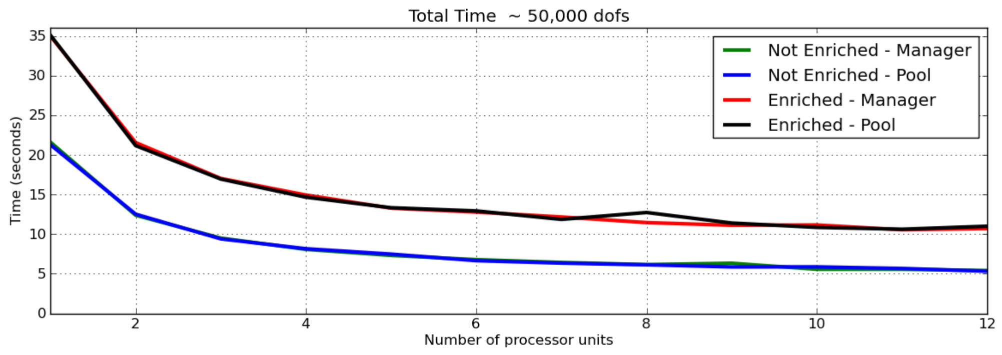


Figure 21: Total solution time for meshes containing around 50,000 degrees of freedom

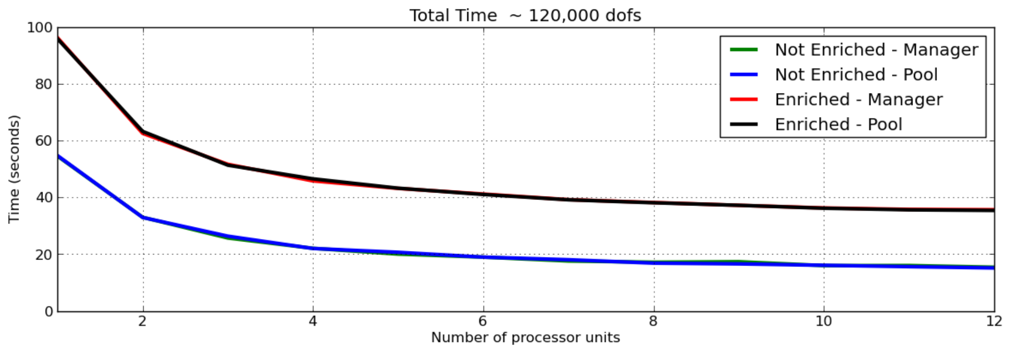


Figure 22: Total solution time for meshes containing around 120,000 degrees of freedom

For all the cases, the gains achieved are more notable up to 6 processing cores; for 7 to 12 processing cores the total time reduction is small. These correspond to the situations in which the speed up results depicted in Figure 18 and Figure 19 shown an instable improvement.

Finally, in order to show the good results achieved by means of the GFEM program, it is shown in Figure 23 the displacements field u and v, respectively in the horizontal and vertical direction, as the stress fields Sxx and Syy in the horizontal and vertical direction, and the shear stress Sxy.

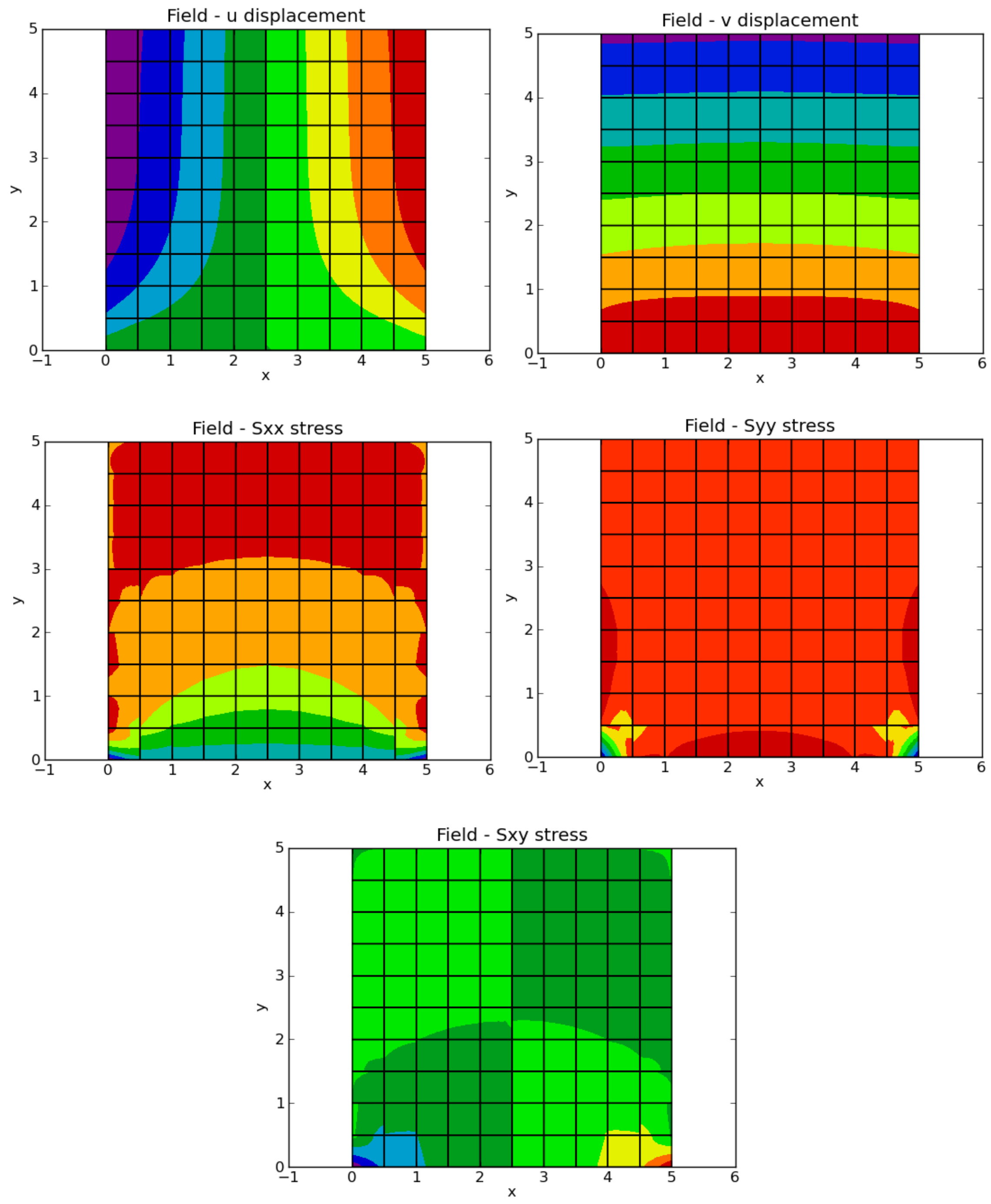


Figure 23: Displacement and stress fields.

5 CONCLUSIONS

Multiprocessing is a process-based ‘threading’-like interface available in Python’s Standard Library that provides an convenient package for shared memory style parallel programming. Its high-level features provides an easy-to-use tool that has demanded only few changes in the original, sequential code, in order to parallelize the local stiffness matrix computation of a Generalized Finite Element code. The speed up observed is attractive in the GNU/Linux operating system and resulted in good time reduction of the total processing time. The same efficiency was not observed in Windows.

REFERENCES

Duarte, C. A. and Oden, J. T., An hp adaptative method using clouds. *Computer Methods in Applied Mechanics and Engineering*, 139:237-262, 1996.

Babuška, I. and Melenk, J. M., The Partition of Unity Method. *International Journal for Numerical Methods in Engineering*, 140:4:727–758, 1997.

Melenk, J.M. and Babuška, I., The Partition of Unity Finite Element Method: Basic Theory and Applications. *Seminars fur Angewandte Mathematik, Eidgenossische Technishe Hochschule*, Research Report No. 96-01, January, CH-8092 Zurich, Switzerland, 1996.

Rossum, G.v., The Python Library Reference - Release 2.7.2, 2011. Available in <http://docs.python.org/download.html>. Acessed in July 17, 2011.

The Scipy Community, NumPy Reference – NumPy v1.5 Manual (DRAFT), 2010. Available in <http://docs.scipy.org/doc/numpy-1.5.x/reference/>. Acessed in April 08, 2011.

The Scipy Community, SciPy - SciPy v0.9 Reference Guide (DRAFT), 2011. Available in <http://numpy.scipy.org/>. Acessed in April 08, 2011.

Zienkiewicz, O.C., and Taylor, R.L., *The finite element method*, volume I. McGraw Hill, 1991.