Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2010 27–30 June 2010.

PyPANCG: A Parallel Python Interface-Library for solving Mildly Nonlinear Systems

Héctor Migallón[1], Violeta Migallón[2] and José Penadés[2]

[1] *Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, 03202 Elche, Alicante, Spain*

[2] *Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, 03071 Alicante, Spain*

emails: hmigallon@umh.es, violeta@dccia.ua.es, jpenades@dccia.ua.es

Abstract

In this paper we present a parallel library, PyPANCG, treated as a high-level interface for solving nonlinear systems. This library consists of two modules, PySParNLCG and PySParNLPCG. The PySparNLCG module parallelizes the conjugate gradient method for solving mildly nonlinear system, and the PySParNLPCG module implements the preconditioning technique based on block two-stage methods. In order to create the high-level interfaces, we have chosen the Python language. On the other hand, the developed Fortran routines offer all the performance of the low-level language. Experimental results report the numerical accuracy and the parallel performance of our approach on different parallel computers.

Key words: parallel libraries, nonlinear algorithms, Python high-level interfaces

1 Introduction

The goal of this paper is to present PyPANCG (http://atc.umh.es/PyPANCG), a Python based high-level parallel interface-library for solving mildly nonlinear systems of the form

$$Ax = \Phi(x), \tag{1}$$

where $A \in \Re^{n\times n}$ and $\Phi : \Re^n \to \Re^n$ is a nonlinear diagonal mapping, i.e., the ith component ϕ_i of ϕ is a function only of the ith component x_i of x.

This library, distributed as a standard Python package, provides parallel implementations of both the nonlinear conjugate gradient method (NLCG) and the nonlinear preconditioned conjugate gradient method (NLPCG). PyPANCG can work with different tools to manage the parallel environment through *MPI* (www-unix.mcs.anl.gov/mpi), by using *PyMPI* or *mpipython* included in *Scientific Python* [4].

 ISBN 13: 978-84-613-5510-5

This paper is structured as follows. Section 2 introduces the nonlinear conjugate gradient method (NLCG) and the parallelization we have performed in the PySParNLCG module of PyPANCG. The nonlinear preconditioned conjugate gradient method and the parallelization performed in the PySParNLPCG module of PyPANCG are introduced in Section 3. In Sections 4, 5 and 6 we explain the main tools used to build PyPANCG, the involved parameters and different ways to implement the nonlinearity, respectively. In Section 7 some examples of using PyPANCG are reported while in Section 8 the behavior of this library is illustrated by means of numerical experiments. Finally, concluding remarks are presented in Section 9.

2 Nonlinear Conjugate Gradient Method

Consider the problem of solving the nonlinear system (1), where $A \in \Re^{n\times n}$ is a symmetric positive definite matrix. An effective approach to solve this nonlinear system is the Fletcher-Reeves version [3] of the nonlinear conjugate gradient method (NLCG). In order to describe the parallelization performed of this method, we consider that A is partitioned into $p \times p$ blocks, with square diagonal blocks of order n_j, $\sum_{j=1}^{p} n_j = n$, such that system (1) can be written as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \Phi_1(x) \\ \Phi_2(x) \\ \vdots \\ \Phi_p(x) \end{bmatrix}, \tag{2}$$

where x and $\Phi(x)$ are partitioned according to the size of the blocks of A. Analogously, we consider $x^{(i)}$, $r^{(i)}$, $p^{(i)}$ and $\Phi(x^{(i)})$ partitioned according to the block structure of A in (2). With this notation we construct the following parallel algorithm.

Algorithm 1 *(Parallel Nonlinear Conjugate Gradient)*
Given an initial vector $x^{(0)}$
  In processor j, $j = 1, 2, \ldots, p$
    $r_j^{(0)} = \Phi_j(x^{(0)}) - [A_{j1}\ A_{j2} \cdots A_{jp}]x^{(0)}$
    $p_j^{(0)} = r_j^{(0)}$
  For $i = 0, 1, \ldots,$ *until convergence*
    In processor j, $j = 1, 2, \ldots, p$
      $\alpha_i =\rightarrow$ *see Algorithm 2*
      $x_j^{(i+1)} = x_j^{(i)} + \alpha_i p_j^{(i)}$
      $r_j^{(i+1)} = r_j^{(i)} - \Phi_j(x^{(i)}) + \Phi_j(x^{(i+1)}) - \alpha_i[A_{j1}\ A_{j2} \cdots A_{jp}]p^{(i)}$
    Convergence test
    In processor j, $j = 1, 2, \ldots, p$
      $\vartheta_j = \langle r_j^{(i+1)}, r_j^{(i+1)} \rangle$
      $\sigma_j = \langle r_j^{(i)}, r_j^{(i)} \rangle$
    Processor 1 computes and broadcasts $\beta_{i+1} = -\sum_{j=1}^{p} \vartheta_j / \sum_{j=1}^{p} \sigma_j$

 ISBN 13: 978-84-613-5510-5

In processor j, $j = 1, 2, \ldots, p$
Compute and perform an allgather $p_j^{(i+1)} = r_j^{(i+1)} - \beta_{i+1} p_j^{(i)}$

Note that, in Algorithm 1, α_i is obtained as follows:

Algorithm 2 *(Computing* α*)*
$\alpha_i^{(0)} = 0$
For $k = 0, 1, 2, \ldots,$ *until convergence*
$\delta^{(k)} = \frac{\alpha_i^{(k)} \left\langle Ap^{(i)}, p^{(i)} \right\rangle - \left\langle r^{(i)}, p^{(i)} \right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i^{(k)} p^{(i)}), p^{(i)} \right\rangle}{\left\langle Ap^{(i)}, p^{(i)} \right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i^{(k)} p^{(i)}) p^{(i)}, p^{(i)} \right\rangle}$
$\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$
Stopping criterion $\left(\left|\delta^{(k)}\right| < \zeta\right)$

3 Nonlinear Preconditioned Conjugate Gradient Method

Preconditioning is a technique for improving the condition number (cond) of a matrix. Suppose that M is a symmetric positive definite matrix that approximates A, but is easier to invert. We can solve $Ax = \Phi(x)$ indirectly by solving $M^{-1}Ax = M^{-1}\Phi(x)$. If $\text{cond}(M^{-1}A) << \text{cond}(A)$ we can iteratively solve $M^{-1}Ax = M^{-1}\Phi(x)$ more quickly than the original problem. In this case we obtain the following nonlinear preconditioned conjugate gradient algorithm.

Algorithm 3 *(Nonlinear Preconditioned Conjugate Gradient)*
Given an initial vector $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
Solve $Ms^{(0)} = r^{(0)}$
$p^{(0)} = s^{(0)}$
For $i = 0, 1, \ldots,$ *until convergence*
$\alpha_i =\rightarrow$ *see Algorithm 2*
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i Ap^{(i)}$
Solve $Ms^{(i+1)} = r^{(i+1)}$
Convergence test
$\beta_{i+1} = -\frac{\left\langle s^{(i+1)}, r^{(i+1)} \right\rangle}{\left\langle s^{(i)}, r^{(i)} \right\rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

Since the auxiliary system $Ms = r$ must be solved at each conjugate gradient iteration, this system needs to be easily solved. Moreover, in order to obtain an effective preconditioner, a good approximation M to the matrix A is needed. One of the general preconditioning techniques for solving linear systems [1] consists of considering a splitting of the matrix A as

$$A = P - Q \tag{3}$$

and performing m steps of the iterative procedure defined by this splitting toward the solution of $As = r$, choosing $s^{(0)} = 0$. In order to obtain the preconditioners suppose

 ISBN 13: 978-84-613-5510-5

that system (1) is partitioned as in (2). Let us consider the splitting (3) consists of the diagonal blocks of A in (2), that is $P = \mathrm{diag}(A_{11}, \ldots, A_{pp})$. Note that in this case, performing m steps of the iterative procedure defined by the splitting (3) to approximate the solution of $As = r$, corresponds to perform m steps of the Block Jacobi method. Thus, at each step l, $l = 1, 2, \ldots$, of a Block Jacobi method, p independent linear systems of the form

$$A_{jj} s_j^{(l)} = (Q s^{(l-1)} + r)_j, \quad 1 \leq j \leq p, \tag{4}$$

need to be solved; therefore each linear system (4) can be solved by a different processor. However, when the order of the diagonal blocks A_{jj}, $1 \leq j \leq p$, is large, it is natural to approximate their solutions by using an iterative method, and thus we are in the presence of a two-stage iterative method; see e.g., [6]. In a formal way, let us consider the splittings

$$A_{jj} = B_j - C_j, \quad 1 \leq j \leq p, \tag{5}$$

and at each lth step perform, for each j, $1 \leq j \leq p$, $q(j)$ iterations of the iterative procedure defined by the splittings (5) to approximate the solution of (4). That is, to solve the auxiliary system $Ms = r$ of Algorithm 3, we use m steps of the following algorithm, choosing $s^{(0)} = 0$.

Algorithm 4 *(Parallel Block Two-Stage)*
Given an initial vector $s^{(0)} = \left((s_1^{(0)})^T, (s_2^{(0)})^T, \ldots, (s_p^{(0)})^T\right)^T$, *and a sequence of numbers of inner iterations* $q(j)$, $1 \leq j \leq p$

For $l = 1, 2, \ldots,$ *until convergence*
  In processor j, $j = 1, 2, \ldots, p$
    $y_j^{(0)} = s_j^{(l)}$
    For $k = 1$ *to* $q(j)$
      $B_j y_j^{(k)} = C_j y_j^{(k-1)} + (Q s^{(l-1)} + r)_j$
    $s^{(l)} = \left((y_1^{(q(1))})^T, (y_2^{(q(2))})^T, \ldots, (y_p^{(q(p))})^T\right)^T$

Therefore, using similar notation as in Section 2, we construct the following parallel nonlinear algorithm.

Algorithm 5 *(Parallel Nonlinear Preconditioned Conjugate Gradient)*
Given an initial vector $x^{(0)}$

In processor j, $j = 1, 2, \ldots, p$
  $r_j^{(0)} = \Phi_j(x^{(0)}) - [A_{j1}\ A_{j2} \cdots A_{jp}] x^{(0)}$
Use m *steps of Alg. 4 to approximate* $As^{(0)} = r^{(0)}$
$p^{(0)} = s^{(0)}$
For $i = 0, 1, \ldots,$ *until convergence*
  In processor j, $j = 1, 2, \ldots, p$
    $\alpha_i =\rightarrow$ *see Algorithm 2*
    $x_j^{(i+1)} = x_j^{(i)} + \alpha_i p_j^{(i)}$
    $r_j^{(i+1)} = r_j^{(i)} - \Phi_j(x^{(i)}) + \Phi_j(x^{(i+1)}) - \alpha_i [A_{j1}\ A_{j2} \cdots A_{jp}] p^{(i)}$

 ISBN 13: 978-84-613-5510-5

Use m steps of Alg. 4 to approximate $As^{(i+1)} = r^{(i+1)}$
Convergence test
In processor $j,\ j = 1, 2, \ldots, p$
$\vartheta_j = \langle s_j^{(i+1)}, r_j^{(i+1)} \rangle$
$\sigma_j = \langle s_j^{(i)}, r_j^{(i)} \rangle$
Processor 1 computes and broadcasts $\beta_{i+1} = -\sum_{j=1}^{p} \vartheta_j / \sum_{j=1}^{p} \sigma_j$
In processor $j,\ j = 1, 2, \ldots, p$
Compute and perform an allgather $p_j^{(i+1)} = r_j^{(i+1)} - \beta_{i+1} p_j^{(i)}$

4 PyPANCG basic tools

This section analyzes the basic tools used in the developed library. The language used for the development of the basic routines and on which the final library will be based was Fortran. The desired objective is to unite the development features offered by Python in a single platform and to approach the execution features offered by, in this case, Fortran. To do this, equivalent routines were developed in both languages. In addition, mixed routines which work with both languages at different levels were developed.

In order to access the routines developed in Fortran from Python, the F2PY tool (cens.ioc.ee/projects/f2py2e) was used. To increase the possible parallel environments, the library has been developed to enable work with two of the most common tools, *mpipython*, which forms part of Scientific Python [4], and *pyMPI*.

Another very important aspect, both for communication between Python and Fortran and for performance, is the use and handling of arrays or vectors; here too, two equivalent options can be used. This is important with regard to the performance of the Python codes and is indispensable when it comes to communication between languages. For the manipulation of vectors, we can use *Numeric* or the *numarray* module included in *NumPy*. The use of one tool or the other is directly related to the tool used to manage the parallel environment. If *mpipython* is used, *Numeric* must be used; if *pyMPI* is chosen, *numarray* must be used instead.

We have developed four specific routines for each functionality. These routines were developed in pure Fortran, or in pure Python, or using two different mixed models. The basic routines have been grouped into operations for sparse matrices (based on SPARSKIT, www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html), basic operations between vectors (based on BLAS, www.netlib.org/blas), and specific functions for the methods at hand, which are associated with different steps of the NLCG and NLPCG algorithms.

5 PyPANCG parameters

This section deals with the parameters which have to be passed to the Python functions which solve a sparse nonlinear system using the NLCG or NLPCG method. The only indispensable parameters are the parameters of the system to be solved ($Ax = \phi(x)$),

 ISBN 13: 978-84-613-5510-5

which are the size of the system, the matrix A stored in CSR (Compressed Sparse Row) format, and the nonlinear mapping $\phi(x)$. In addition the derivative of $\phi(x)$ ($\phi'(x)$) is required for computing δ according to Algorithm 2. However, there is a series of parameters that permits the modification of these algorithms. If values are not specified, default values are used. The optional parameters used in both algorithms and their default values are as follows:

- *initial_vector*: Initial iterate equal to zero.
- *global_stopping_error* $\xi = 10^{-7}$: Global stopping criterion evaluated using the euclidean norm of the residual vector ($\|r\|_2$).
- *alfa_stopping_error* $\zeta = 10^{-7}$: Stopping criterion for computing α evaluated using the absolute value of δ.
- *iter_alfa* $= 0$: By setting this parameter to a value higher than 1, we can limit the number of iterations performed to calculate α.
- *For_or_Py* = "Python_full": It selects one of the four different sets of routines to be used during the algorithm execution. As it has been mentioned above, these routines differ in the coding language.
- *trash_int*: Integer vector (see Section 6).
- *trash_double*: Double precision vector (see Section 6).

The NLPCG specific parameters and their default values are:

- *level* $= 1$: Level of fill-in of the incomplete LU factorization used in Algorithm 4 in order to obtain the inner splittings (5) (see Section 8).
- *niter_2e* $= 3$: Number of steps m performed by Algorithm 4 to approximate the corresponding linear system in Algorithm 5.
- *val_q* $= 3$: Number of inner iterations $q(j), 1 \leq j \leq p$ performed in Algorithm 4.

Another important parameter that the system can calculate -if the matrix is available in the *root* processor- is the size of the problem assigned to each processor; this is given by the parameter *block_dimensions*. This parameter is an integer vector whose dimension corresponds to the number of processors and which stores the block size assigned to each processor. In the examples provided by PyPANCG, the parameter is internally calculated, such that a load balancing is achieved. If the matrix is distributed among processors, this parameter must specify the portion available at each processor.

The parameter *For_or_Py* selects the set of routines to be used. The following options can be chosen with regard to this parameter:

1. *Python_full*: All of the routines used are codified in Python.

 ISBN 13: 978-84-613-5510-5

2. *Python*: The routines used are codified in Python but the functions that come from SPARSKIT and BLAS are in Fortran.

3. *Fortran*: All of the routines used are codified in Fortran. Moreover ϕ and ϕ' are codified independently.

4. *Fortran_full*: All of the routines used are codified in Fortran but ϕ and ϕ' are not codified independently.

The options are listed in performance order from poorest to best and in usability and development speed order from best to poorest. It is worth pointing out that the *Python* option is a mixed option whereas the rest of the options use either Python or Fortran for the basic routines. The difference between *Fortran* and *Fortran_full* is that in the first option the user must only codify the functions ϕ and ϕ' in Fortran, whereas in the latter option all routines implementing these functions must be codified in Fortran, and thus the user must understand the internal development of the method in great depth.

6 Nonlinearity implementation

One of the major obstacles to develop libraries for solving nonlinear systems is the implementation of the nonlinearity of the problem to be solved. One important aspect is that the ith component ϕ_i of ϕ only depends on the ith component of x. Thus, ϕ and ϕ' can be developed at vector level or at vector component level. For performance reasons, development will take place at vector level if the development is realized in Python and, for usability reasons, it will take place at component level when Fortran language is used. The example below shows the Python code for the function $\phi(x)$ used in the examples of PyPANCG.

```
def Fi_x(vector,trash_int,trash_double):
    sc = trash_double[0]
    x = -sc*numpy.exp(vector)
    return x
```

The same function developed in Fortran is:

```
double precision function phi(input,trash_int,trash_double)
    implicit none
    real*8 input,trash_double(*),sc
    integer trash_int(*)
    sc = trash_double(1)
    phi = -sc*exp(input)
    return
```

In addition to observing that the Python code works using vectors whilst Fortran works using a single component, it is important to note that both functions require a parameter transfer (sc) for the computation of ϕ. To realize this transfer -both real values and integer values if needed- we use two vectors, one integer vector *trash_int* and

 ISBN 13: 978-84-613-5510-5

one double precision real vector *trash_double*. These vectors are dynamic and thus all parameters required for the computation can be passed to functions ϕ and ϕ'. Naturally, these functions must always be implemented in order to adapt to the problem to be solved. If they are implemented in Python, the option *Python* or *Python_full* must be used. If they are implemented in Fortran, the option *Fortran* or *Fortran_full* must be used. Moreover, in the latter case, the module must be installed and compiled again following the development of the functions.

The options *Python* and *Fortran* are very similar; both use basic functions in Fortran but differ in their implementation of the functions ϕ and ϕ'. The option *Fortran_full* does not use these functions except for integrating them in the routines that use these functions. Thus, its adaptation is more complicated and laborious. However, it is the option that provides the best performance. On the other hand, *Python_full* option does not use any Fortran code, which enables much faster development but an excessively poor performance.

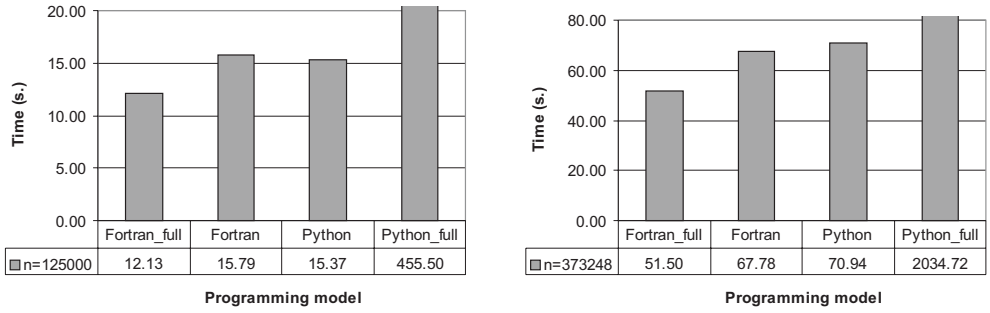
7 Using PySParNLCG and PySParNLPCG

As already mentioned, in order to use the library the size of the system (nrow), the matrix A in CSR format (tcol, trow, tval), the block size assigned to each processor (block_dimensions), and the nonlinear functions (ϕ and ϕ') must be passed at the very least. However, if we wish to pass additional parameters we will use the variables *trash_int* and *trash_double*. The following code shows the most simple NLCG function call, in which we assume that the functions ϕ and ϕ' were implemented in Python beforehand.

```
1  from math import exp
2  import numpy
3  import PyPANCG
4  import PyPANCG.PySParNLCG as PySparNLCG

5  iam = PySParNLCG.iam
6  trash_double = numpy.zeros(((1),),float)
7  trash_double[0] = 6/(float(49)**3)
8  nrow = 125000

9  nrow,block_dimensions,bls = _
       PyPANCG.MakeBlockStructure(nrow=nrow)
10 nnz,tcol,trow,tval = PyPANCG.PartialMatrixA _
       (Mx=Mx,s=bls[iam],d=block_dimensions[iam])

11 x,error,time,iter = PySParNLCG.nlcg(nrow=nrow, _
                tcol=tcol,trow=trow,tval=tval, _
                block_dimensions = block_dimensions, _
                Fi_x=Fi_x,Fi_prime_x=Fi_prime_x, _
                trash_double = trash_double)
```

The matrix A is obtained in lines 9 and 10; this code is enclosed with the library but can only be used as an example or test. It is important to point out that each

 ISBN 13: 978-84-613-5510-5



(a) $n = 125000$ (b) $n = 373248$

Figure 1: PySParNLCG using 2 processors, pyMPI, SULLI.

processor only contains the portion of the matrix that it requires. In line 11, the actual call to the NLCG method takes place, whereby we assume that *Fi_x* (ϕ) and *Fi_prime_x* (ϕ') were declared in Python and the vector *trash_double* is passed, in this case of a single component.

The most simple NLPCG function call is similar to the NLCG example above showed. In this case it must import PySParNLPCG module instead of PySParNLCG module in line 4,

```
4  import PyPANCG.PySParNLPCG as PySparNLPCG
```

and it must call NLPCG method in line 11,

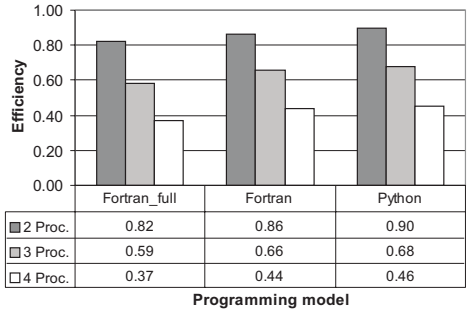
```
11 x,error,time,iter = PySParNLPCG.nlpcg(nrow=nrow, _
              tcol=tcol,trow=trow,tval=tval, _
              block_dimensions = block_dimensions, _
              Fi_x=Fi_x,Fi_prime_x=Fi_prime_x, _
              trash_double = trash_double)
```

8 Numerical experiments

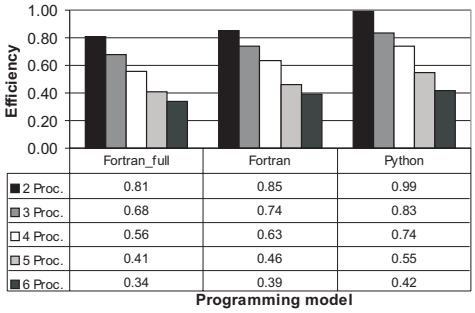
In order to illustrate the behavior of PyPANCG, we have tested the algorithms provided by this library on two multicore computers. The first platform, Bi-Quad, is a DELL PowerEdge 2900 with two Quad-Core Intel Xeon 5320 sequence processors at up to 1.86 GHz, with 8 GB of RAM. The second platform, SULLI, is an Intel Core 2 Quad Q6600, 2.4 GHz, with 4 GB of RAM.

As our illustrative example we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem [2]. To solve this problem using the finite difference method, we consider a grid in Ω of d^3 nodes, where Ω is a 3D cube domain of unit length. The discretization of this problem yields a nonlinear system of the form

 ISBN 13: 978-84-613-5510-5



(a) SULLI



(b) Bi-Quad

Figure 2: Efficiency of PySParNLCG, $n = 373248$, pyMPI.

$Ax = \Phi(x)$, where $\Phi : \Re^n \to \Re^n$ is a nonlinear diagonal mapping. We present here results obtained with $d = 50$, $d = 72$ and $d = 84$, that lead to nonlinear systems of size 125000, 373248 and 592704, respectively. The convergence test used was $\|r\|_2 < 10^{-7}$ and the stopping criterion for α was $|\delta| < 10^{-7}$. Concretely, these are the default values for *global_stopping_error* and *alfa_stopping_error* in PyPANCG.

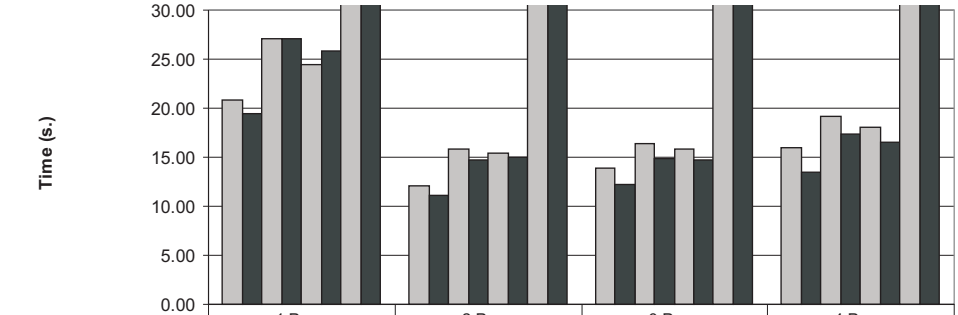
First, we analyze the behavior of PyPANCG.PySParNLCG depending on different values of the parameter *For_or_Py*. Figure 1 shows that the best results are obtained using routines fully developed in Fortran such that the computation of ϕ and ϕ' is performed inside these routines. The worst results are obtained with the option *For_or_Py='Python_full'*. Note that this option uses pure Python routines and it should only be used in the development process. On the other hand, the options that combine Fortran and Python code get similar performance.

Figure 2 analyzes the influence of the number of processors, on the two multicore platforms above mentioned. As it can be seen, the best efficiencies are obtained using 2 or 3 processors in SULLI, or a maximum of 5 processors in Bi-Quad.

Figure 3 compares the use of *mpipython* and *Numeric* with the use of *pyMPI* and *numpy* in the behavior of PyPANCG.PySParNLCG. As it can be seen, *Numeric* offers better performance than *numpy*, specially when the option *For_or_Py='Python_full'* is used. Therefore, a calling to a module with a single processor always uses *Numeric*.

In order to analyze the PyPANCG.PySParNLPCG module we consider, in our experiments, the outer splitting $A = P - Q$ determined by $P = \text{diag}(A_{11}, \ldots, A_{pp})$. Let us further consider an incomplete LU factorization of each matrix A_{jj}, $j = 1, 2, \ldots, p$, that is $A_{jj} = L_j U_j - R_j$, and at each *lth* step perform, for each j, $q(j)$ inner iterations of the iterative procedure defined by this splitting. Let us denote by ILU(S) the incomplete LU factorization associated with the zero pattern subset S of $S_n = \{(i, j) : i \neq j,\ 1 \leq i, j \leq n\}$. In particular, when $S = \{(i, j) :\ a_{ij} = 0\}$, the incomplete factorization with zero fill-in, known as ILU(0), is obtained. To improve the quality

 ISBN 13: 978-84-613-5510-5



	1 Proc.	2 Proc.	3 Proc.	4 Proc.
Fortran_full - pyMPI	20.85	12.13	13.90	16.01
Fortran_full - mpipython	19.48	11.16	12.27	13.45
Fortran - pyMPI	27.05	15.79	16.35	19.18
Fortran - mpipython	27.04	14.76	14.83	17.34
Python - pyMPI	24.42	15.37	15.84	18.10
Python - mpipython	25.85	15.05	14.75	16.55
Python_full - pyMPI	897.59	455.50	293.39	229.87
Python_full - mpipython	276.85	143.93	104.37	86.54

Number of processors

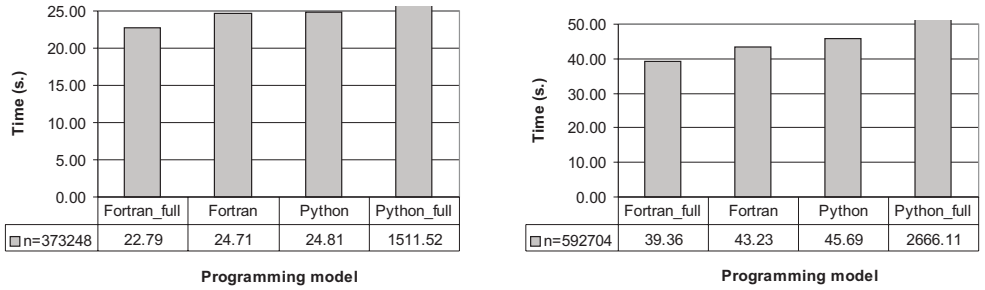
Figure 3: PySParNLCG: *mpipython* versus *pyMPI*, $n = 125000$, SULLI.

of the factorization, many strategies for altering the pattern have been proposed. In the experiments reported here, we have used the "level of fill-in" factorizations [5], ILU(κ), $\kappa \geq 0$. Figure 4 illustrates the behavior of PySParNLPCG depending on the different options of *For_or_Py*. Similar performances to those for PySParNLCG module are obtained. That is, the best results are obtained setting *For_or_Py='Fortran_full'* and the worst results using *For_or_Py='Python_full'*. The other two options present similar performance.

9 Conclusion

In this paper we have presented PyPANCG, a Python library-interface that implements both the conjugate gradient method and the preconditioned conjugate gradient method for solving nonlinear systems. We have described the use of the library and its advantages in order to get fast development. The aim of this library is to develop high performance scientific codes for high-end computers hiding many of the underlying low-level programming complexities from users with the use of a high-level Python interface. The library has been designed for adapting to different stages of the design process, depending on whether the purpose is computational performance or fast development.

 ISBN 13: 978-84-613-5510-5



(a) $n = 373248$ (b) $n = 592704$

Figure 4: PySParNLPCG using 2 processors, $\kappa = 1$, mpipython, SULLI.

Acknowledgements

This research was supported by the Spanish Ministry of Science and Innovation under grant number TIN2008-06570-C04-04.

References

[1] L. Adams, *M-step preconditioned conjugate gradient methods*, SIAM Journal on Scientific and Statistical Computing **6** (1985) 452–462.

[2] B. M. Averick, R. G. Carter, J. J. More and G. Xue, *The MINPACK-2 Test Problem Collection*, Technical Report MCS-P153-0692, Mathematics and Computer Science Division, Argonne, 1992.

[3] R. Fletcher and C. Reeves, *Function Minimization by Conjugate Gradients*, The Computer Journal **7** (1964) 149–154.

[4] K. Hinsen, *Scientific Python User's Guide*, Centre de Biophysique Moleculaire CNRS, Grenoble, France, 2002.

[5] H. P. Langtangen, *Conjugate gradient methods and ILU preconditioning of non-symmetric matrix systems with arbitrary sparsity patterns*, International Journal for Numerical Methods in Fluids **9** (1089) 213–233.

[6] V. Migallón and J. Penadés, *Convergence of two-stage iterative methods for hermitian positive definite matrices*, Applied Mathematics Letters **10(3)** (1997) 79–83.

 ISBN 13: 978-84-613-5510-5