J Supercomput (2011) 58:438–448
DOI 10.1007/s11227-011-0609-1

A Parallel Python library for nonlinear systems

Héctor Migallón · Violeta Migallón · José Penadés

Published online: 15 April 2011
© Springer Science+Business Media, LLC 2011

Abstract In this paper, we present PyPANCG, a Python library-interface that implements both the conjugate gradient method and the preconditioned conjugate gradient method for solving nonlinear systems. We describe 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. Experimental results report the performance of our approach on different parallel computers.

Keywords Parallel libraries · Nonlinear algorithms · Python high-level interfaces

1 Introduction

The use of high level environments, and particularly the Python programming language [15], is common place in science and engineering (see, e.g., [3, 11, 12]) to enable the development of custom applications, particularly during the early stages

H. Migallón (✉)
Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, 03202 Elche, Alicante, Spain
e-mail: hmigallon@umh.es

V. Migallón · J. Penadés
Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, 03071 Alicante, Spain

V. Migallón
e-mail: violeta@dccia.ua.es

J. Penadés
e-mail: jpenades@dccia.ua.es

of a new product or system modeling, simulation, and optimization. These very high level languages make it easy to manipulate high level objects (e.g., matrices), hiding many of the underlying low-level programming complexities from users. They also support rapid code iteration and refinement by enabling an interactive development and execution environment. Although there are other high level languages (like Matlab, Mathematica, ...) that are more popular than Python, we decided to use this interpreter due to its remarkable power and very clear syntax; also, it is available for free. Python scripts can call out to existing Fortran, C and C++ libraries, Java classes, and many more.

Although Python was not targeted for parallel programming in the original design, nowadays several implementations exist that enable execution in a message-passing parallel environment. Different tools exist to manage the parallel environment through *MPI* [8]. These tools are Python extensions that are implemented by building an alternative startup executable for Python. Examples of these parallel extensions are *pyMPI* [10] and *mpipython* included in *Scientific Python* [6]. In addition, there are several interfaces to scientific computing libraries, for example, *PyACTS* [3], *Visual Python* [14], and *PyTrilinos* [13].

The goal of this paper is to present PyPANCG (atc.umh.es/PyPANCG), a Python based high-level parallel interface-library for solving 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, consists of two modules, PySParNLCG and PySParNLPCG. The PySParNLCG module provides parallel implementations of the nonlinear conjugate gradient method (NLCG) and the PySParNLPCG module implements the nonlinear preconditioned conjugate gradient method (NLPCG). These algorithms are based on both the Fletcher–Reeves version of the conjugate gradient method [4] and polynomial preconditioners based on the block two-stage method [5].

This paper is structured as follows. Section 2 introduces these algorithms and the parallelization we have performed in the PySParNLCG and PySParNLPCG modules of PyPANCG. Sections 3 and 4 explain the main tools used to build PyPANCG, the involved parameters and different ways to implement the nonlinearity. In Sect. 5, some examples of using PyPANCG are reported while in Sect. 6 the behavior of this library is illustrated by means of numerical experiments. Finally, concluding remarks are presented in Sect. 7.

2 Nonlinear conjugate gradient algorithms

Consider that the matrix A in (1) is partitioned into $p \times p$ blocks, with square diagonal blocks of order n_j, $\sum_{j=1}^{p} n_j = n$. Analogously, we consider $x^{(i)}, r^{(i)}, p^{(i)}$, and $\Phi(x^{(i)})$ partitioned according to the block structure of A. With this notation, we construct the following parallel algorithm based on the NLCG method.

Algorithm 1 (*Parallel Nonlinear Conjugate Gradient*)
Given an initial vector $x^{(0)}$

Process 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*
Process j, $j = 1, 2, \ldots, p$
$\alpha_i \Longrightarrow$ *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
Process j, $j = 1, 2, \ldots, p$
$\vartheta_j = \langle r_j^{(i+1)}, r_j^{(i+1)} \rangle$
$\sigma_j = \langle r_j^{(i)}, r_j^{(i)} \rangle$
Process 1 *computes and broadcasts* $\beta_{i+1} = -\sum_{j=1}^{p} \vartheta_j / \sum_{j=1}^{p} \sigma_j$
Process j, $j = 1, 2, \ldots, p$
Compute and perform an allgather $p_j^{(i+1)} = r_j^{(i+1)} - \beta_{i+1} p_j^{(i)}$

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

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 $\text{cond}(M^{-1}A) \ll \text{cond}(A)$, we can iteratively solve $M^{-1}Ax = M^{-1}\Phi(x)$ more quickly than the original problem.

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

$$A = P - Q \tag{2}$$

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, let us consider the splitting (2) such that $P = \text{diag}(A_{11}, \ldots, A_{pp})$. Note that in this case, performing m steps of the iterative procedure defined by the splitting (2) 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

$$A_{jj} s_j^{(l)} = (Qs^{(l-1)} + r)_j, \quad 1 \le j \le p, \tag{3}$$

need to be solved, therefore, each linear system (3) can be solved by a different process. However, when the order of the diagonal blocks A_{jj}, $1 \le j \le p$ 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., [9]. In a formal way, let us consider the inner splittings

$$A_{jj} = B_j - C_j, \quad 1 \le j \le p, \tag{4}$$

and at each lth step perform for each j, $1 \le j \le p$, $q(j)$ inner iterations of the iterative procedure defined by the splittings (4) to approximate the solution of (3). Algorithm 3 summarizes this parallel block two-stage method for solving $As = r$.

Algorithm 3 (*Parallel Block Two-Stage*)
Given an initial vector $s^{(0)} = ((s_1^{(0)})^T, (s_2^{(0)})^T, \dots, (s_p^{(0)})^T)^T$, *and a sequence of numbers of inner iterations* $q(j), 1 \le j \le p$

For $l = 1, 2, \dots,$ *until convergence*
  Process $j, j = 1, 2, \dots, p$
    $y_j^{(0)} = s_j^{(l)}$
    For $k = 1$ *to* $q(j)$
      $B_j y_j^{(k)} = C_j y_j^{(k-1)} + (Qs^{(l-1)} + r)_j$
    $s^{(l)} = ((y_1^{(q(1))})^T, (y_2^{(q(2))})^T, \dots, (y_p^{(q(p))})^T)^T$

Therefore, using similar notation as in Algorithm 1, we construct the following parallel nonlinear preconditioned algorithm.

Algorithm 4 (*Parallel Nonlinear Preconditioned Conjugate Gradient*)
Given an initial vector $x^{(0)}$
  Process $j, j = 1, 2, \dots, p$
    $r_j^{(0)} = \Phi_j(x^{(0)}) - [A_{j1}\ A_{j2} \cdots A_{jp}]x^{(0)}$
  Use m steps of Algorithm 3 to approximate $As^{(0)} = r^{(0)}$
  $p^{(0)} = s^{(0)}$
  For $i = 0, 1, \dots,$ *until convergence*
    Process $j, j = 1, 2, \dots, p$
      $\alpha_i \Longrightarrow$ *see Algorithm 2*
      $x_j^{(i+1)} = x_j^{(i)} + \alpha_i p_j^{(i)}$
      $r_j^{(i+1)} = r_j^{(i)} - \Phi_j(x^{(i)}) + \Phi_j(x^{(i+1)}) - \alpha_i [A_{j1}\ A_{j2} \cdots A_{jp}]p^{(i)}$
    Use m steps of Algorithm 3 to approximate $As^{(i+1)} = r^{(i+1)}$
    Convergence test
    Process $j, j = 1, 2, \dots, p$
      $\vartheta_j = \langle s_j^{(i+1)}, r_j^{(i+1)} \rangle$
      $\sigma_j = \langle s_j^{(i)}, r_j^{(i)} \rangle$
    Process 1 *computes and broadcasts* $\beta_{i+1} = -\sum_{j=1}^p \vartheta_j / \sum_{j=1}^p \sigma_j$
    Process $j, j = 1, 2, \dots, p$
      Compute and perform an allgather $p_j^{(i+1)} = r_j^{(i+1)} - \beta_{i+1} p_j^{(i)}$

We would like to point out that the developed implementations of the above algorithms exploit, if needed, the sparsity of the matrix A (see Sects. 3 and 5). For the sake of simplicity, we have omitted this description from the formulation of the algorithms.

3 PyPANCG basic tools and parameters

The development of the basic routines has been done using Fortran language in such a way that the whole library is based on this language. 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, and *pyMPI*. The use of MPI through these tools enables PyPANCG to be used, among others, on clusters with or without multicore nodes, and on multicore platforms. In addition, the library has been designed such that one MPI process per core is assigned.

Another very important aspect, both for communication between Python and Fortran as well as for performance is the use and handling of arrays or vectors; two equivalent options can be used here, also. 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), basic operations between vectors (based on BLAS), and specific functions for the methods at hand, which are associated with different steps of the NLCG and NLPCG algorithms. We want to point out that PyPANCG does not need to be compiled with either SPARSKIT or with BLAS because the needed functions of these libraries have been rewritten and adapted to our requirements.

The only indispensable parameters in a PyPANCG call are the parameters related to the system to be solved ($Ax = \phi(x)$), 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. 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.
- *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.

– *trash_int*: Integer vector (see Sect. 4).
– *trash_double*: Double precision vector (see Sect. 4).

The PySParNLPCG module uses Algorithm 3 for preconditioning the NLCG method. For this algorithm, the outer splitting $A = P - Q$ is determined by $P = \text{diag}(A_{11}, \ldots, A_{pp})$ and the inner splittings are of the form $A_{jj} = L_j U_j - R_j$, where each $L_j U_j$ is an incomplete LU factorization of the matrix A_{jj}, $j = 1, 2, \ldots, p$. 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 of the factorization, many strategies for altering the pattern have been proposed. In the "level of fill-in" factorizations [7], ILU(κ), $\kappa \geq 0$, a level of fill-in is recursively attributed to each fill-in position from the levels of its parents. Then the positions of the level lower than κ are removed from S. The PySParNLPCG module implements these "level of fill-in" factorizations, ILU(κ), $\kappa \geq 0$. In this way, the NLPCG specific parameters and their default values are:

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

Another important parameter that the system can calculate if the matrix is available in only one of the processes is the size of the problem assigned to each process; this is given by the parameter *block_ dimensions*. In the examples provided by PyPANCG, the parameter is internally calculated, such that a load balancing is achieved. If the matrix is distributed among processes, this parameter must specify the portion available at each process. 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.
2. *Python*: The routines used are codified in Python but the functions based on 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.

4 Nonlinearity implementation

One of the major obstacles to developing 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 the vector level or at the vector component level. For performance reasons, development will take place at vector level if Python is used and, for usability reasons, it will take place at the component level when Fortran 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 while 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 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 being solved. If they are implemented in Python, the *Python* or *Python_full* options must be used. If they are implemented in Fortran, the *Fortran* or *Fortran_full* options must be used. Moreover, in the latter case, the module must be installed and compiled again following the development of the functions. The *Python* and *Fortran* options are very similar; both use basic functions in Fortran but differ in their implementation of the functions ϕ and ϕ'.

The *Fortran_full* option 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.

5 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 process (block_dimensions), and the nonlinear functions (ϕ and ϕ') must be passed at the very least. However, if we wish to pass other 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 process 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 aforementioned NLCG example. In this case it must import the PySParNLPCG module instead of the PySParNLCG module in line 4, and it must call the NLPCG method in line 11.

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

```
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)
```

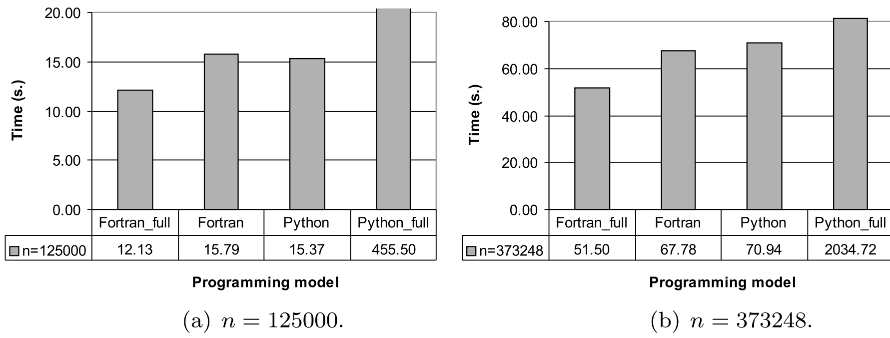
Note that there are parameters that can significantly change the behavior of the NLPCG method. These parameters are the level of fill-in (*level*) used in the ILU factorization, the number of steps m performed by Algorithm 3 (*niter_2e*), and the number of inner iterations (*val_q*) performed in the block two-stage method. The following example changes the default values of these parameters to other ones.

```
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, _
              level = 2, niter_2e = 4, val_q = 5, trash_double = trash_double)
```

6 Numerical experiments

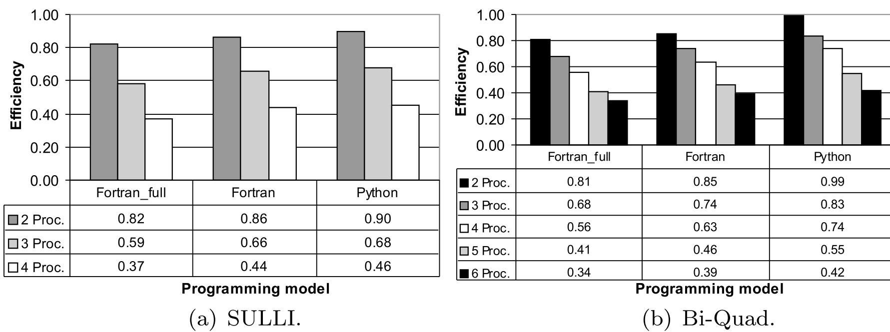
In order to illustrate the behavior of PyPANCG, we have tested the algorithms provided by this library on two multicore computers. The first platform, Bi-Quad, is a DELL PowerEdge 2900 with two Quad-Core Intel Xeon 5320 sequence processors at up to 1.86 GHz with 8 GB of RAM. The second platform, SULLI, is an Intel Core 2 Quad Q6600, 2.4 GHz with 4 GB of RAM and 8 MB of L2 Cache Memory. The operating system in Bi-Quad is Ubuntu 8.04 (Hardy Heron), and the operating system in SULLI is Ubuntu 9.04 (Jaunty Jackalope), in both cases for 64 bit systems. We have used the GNU Fortran compiler *gfortran* included in *gcc 4.3.3*. Furthermore, the versions of the tools used in PyPANCG are the following: *Python 2.6.2*, *numpy 1.2.1*, *Numeric 24.2*, *mpich 1.2.7-p1*, *pyMPI 2.5*, and *mpipython 1.0*.

As our illustrative example, we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem [2]. We present here the results obtained with 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 $|\delta| < 10^{-7}$. We first analyze the behavior of PySParNLCG, Fig. 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 *For_or_Py* = *'Python_full'* option. On the other hand, the options that combine Fortran and Python code achieve similar performance. Figure 2 analyzes the influence



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

Fig. 1 PySParNLCG using 2 processes, pyMPI, SULLI



(a) SULLI. (b) Bi-Quad.

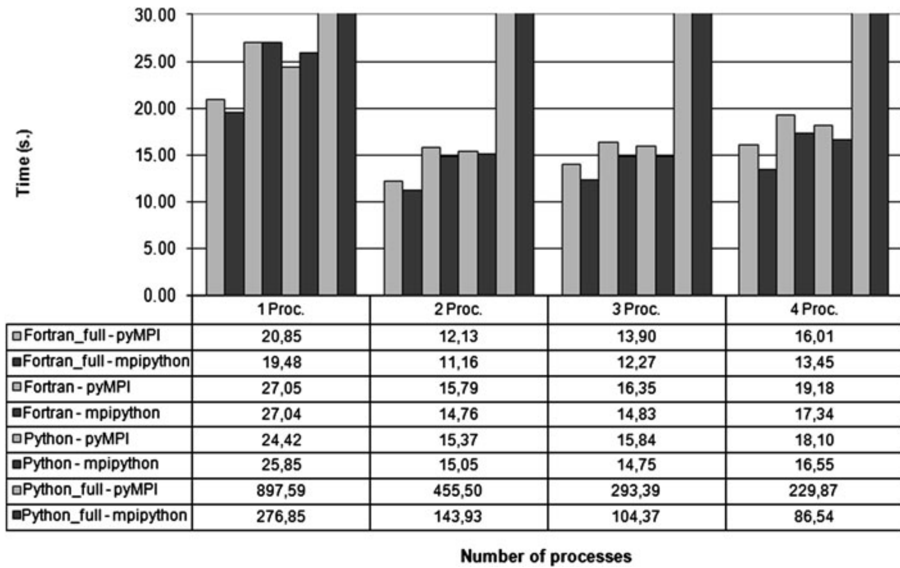
Fig. 2 Efficiency of PySParNLCG, $n = 373248$, pyMPI

of the number of processes, on the two multicore platforms mentioned previously. The best efficiencies are obtained using 2 or 3 processes in SULLI, or a maximum of 5 processes 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 can be seen, *Numeric* offers better performance than *numpy*, more remarkable when the *For_or_Py* = *'Python_full'* option is used.

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.

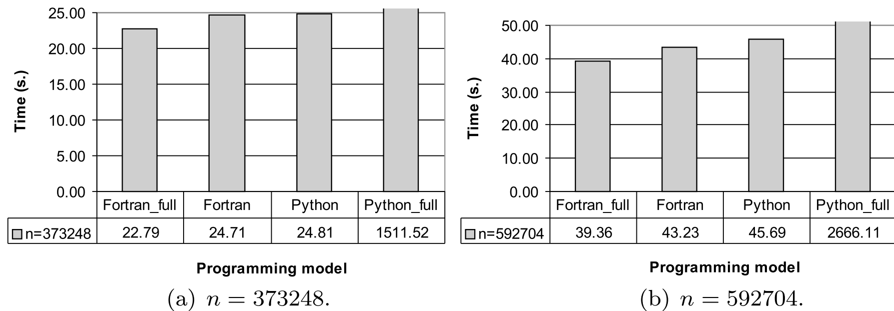
7 Conclusions

In this paper, we have presented PyPANCG, a parallel Python library-interface for solving nonlinear systems. PyPANCG is distributed as a standard Python package, and it consists of two modules that parallelize the conjugate gradient method for solving nonlinear systems, and the preconditioning technique based on block two-stage methods using incomplete LU factorizations. This work follows the guidelines of a high-level interface providing an easy-to-use parallel environment that hides the challenges of parallel programming, but in fact it is a library developed using a mixed



	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

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



	Fortran_full	Fortran	Python	Python_full
n=373248	22.79	24.71	24.81	1511.52

(a) $n = 373248$.

	Fortran_full	Fortran	Python	Python_full
n=592704	39.36	43.23	45.69	2666.11

(b) $n = 592704$.

Fig. 4 PySParNLPCG using 2 processes, $\kappa = 1$, mpipython, SULLI

model that uses different programming languages. In order to create the high-level interfaces, we have chosen the Python language for several reasons, for example, its remarkable power is combined with a very clear syntax. The library combines the development features offered by Python and the execution performance offered by, in this case, Fortran. It has been designed to adapt to different stages of the design process, depending on whether the purpose is computational performance or rapid code iteration and refinement. Moreover, in order to illustrate the behavior of PyPANCG, we have tested the algorithms provided by this library on two multicore computers. Numerical experiments show good behavior but are obviously not comparable with lower level developments. The best results are obtained using routines fully developed in Fortran where the computation of the nonlinearity is performed inside these routines.

Acknowledgements This research was partially supported by the Spanish Ministry of Science and Innovation under Grant No. TIN2008-06570-C04-04.

References

1. Adams L (1985) M-step preconditioned conjugate gradient methods. SIAM J Sci Stat Comput 6:452–462
2. Averick BM, Carter RG, More JJ, Xue G (1991) The MINPACK-2 test problem collection. Technical Report MCS-P153-0692, Mathematics and Computer Science Division, Argonne
3. Drummond LA, Galiano V, Marques O, Migallón V, Penadés J (2007) PyACTS: a high-level framework for fast development of high performance applications. In: Lect notes comput sci, vol 4395. Springer, Berlin, pp 417–425
4. Fletcher R, Reeves C (1964) Function minimization by conjugate gradients. Comput J 7:49–154
5. Galiano V, Migallón H, Migallón V, Penadés J (2009) Parallel nonlinear preconditioners on multicore architectures. J Supercomput. doi:10.1007/s11227-009-0351-0
6. Hinsen K (2002) Scientific Python user's guide. Centre de Biophysique Moleculaire CNRS, Grenoble
7. Langtangen HP (1989) Conjugate gradient methods and ILU preconditioning of nonsymmetric matrix systems with arbitrary sparsity patterns. Int J Numer Methods Fluids 9:213–233
8. Message Passing Interface (2011) http://www-unix.mcs.anl.gov/mpi/
9. Migallón V, Penadés J (1997) Convergence of two-stage iterative methods for Hermitian positive definite matrices. Appl Math Lett 10(3):79–83
10. Miller P (2002) PyMPI—an introduction to parallel Python using MPI. http://www.llnl.gov/computing/develop/python/pyMPI.pdf
11. Painter J, Merritt EA (2004) mmLib Python toolkit for manipulating annotated structural models of biological macromolecules. J Appl Crystallogr 37(1):174–178
12. Sáenz J, Zubillaga J, Fernández J (2002) Geophysical data analysis using Python. Comput Geosci 24(4):457–465
13. Sala M, Spotz W, Heroux M (2008) PyTrilinos: high-performance distributed-memory solvers for Python. ACM Trans Math Softw 34(2):1–33
14. Scherer D, Dubois P, Sherwood B (2000) VPython: 3D interactive scientific graphics for students. Comput Sci Eng 2(5):56–62
15. van Rossum G, Drake FL Jr (2003) An introduction to Python. Network Theory Ltd., Bristol