Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2011 26-30 June 2011.

Python Interface-Library using OpenMP and CUDA for solving 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 new features of PyPANCG. PyPANCG is a parallel library treated as a high-level interface for solving nonlinear systems. Using the new features, PyPANCG is able to exploit the parallelism offered by shared memory platforms and graphics processing units (GPUs). The new library still has two modules, PySParNLCG and PySParNLPCG, which include new features, both modules are backward-compatible with the earlier versions of PyPANCG. The PySparNLCG module parallelizes the conjugate gradient method for solving mildly nonlinear systems, 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. Experimental results report the behavior and the parallel performance of our approach on both the shared memory platforms and the GPUs.

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

1 Introduction

In this paper we present new features of 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.

 ISBN: 978-84-614-6167-7

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 earlier versions could work with different tools to manage a distributed memory platform through *MPI* (www-unix.mcs.anl.gov/mpi). The PyPANCG current version allows to work with shared memory platforms through OpenMP and, using CUDA, this library is able to work with GPUs.

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

2 Nonlinear methods

Consider the problem of solving the nonlinear system (1), where $A \in \Re^{n\times n}$ is a symmetric positive definite matrix and $\Phi : \Re^n \to \Re^n$ is a nonlinear function with certain local smoothness properties. Let $\Psi : \Re^n \to \Re$ be a nonlinear mapping and consider $\langle x, y\rangle = x^T y$ the inner product in $\Re^n$. The minimization problem of finding $x \in \Re^n$ such that

$$J(x) = \min_{y\in\Re^n} J(y), \tag{2}$$

where $J(x) = \frac{1}{2}\langle Ax, x\rangle - \Psi(x)$, is equivalent to find $x \in \Re^n$ such that $F(x) = Ax - \Phi(x) = 0$, where $\Phi(x) = \Psi'(x)$.
An effective approach for solving nonlinear system (1), by considering the connection with the minimization problem (2), is the Fletcher-Reeves version [5] of the nonlinear conjugate gradient method (NLCG), which takes the following form:

Algorithm 1 *(Fletcher-Reeves Nonlinear Conjugate Gradient)*
Given an initial vector $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
$p^{(0)} = r^{(0)}$
For $i = 0, 1, \ldots,$ *until convergence*
$\alpha_i =\rightarrow$ *see below*
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
Convergence test
$\beta_{i+1} = -\frac{\langle r^{(i+1)}, r^{(i+1)}\rangle}{\langle r^{(i)}, r^{(i)}\rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

 ISBN: 978-84-614-6167-7

Note that, in Algorithm 1, α_i is chosen to minimize the associated functional J in the direction $p^{(i)}$. This is equivalent to solve the one dimensional zero-point problem $\frac{dJ(x^{(i)}+\alpha_i p^{(i)})}{d\alpha_i} = 0$. From the definition of J it follows that

$$J(x^{(i)} + \alpha p^{(i)}) = \frac{1}{2}\left\langle A(x^{(i)} + \alpha_i p^{(i)}), x^{(i)} + \alpha_i p^{(i)}\right\rangle - \Psi(x^{(i)} + \alpha_i p^{(i)}).$$

Then a simple differentiation with respect to α_i yields

$$\frac{dJ(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i} = \alpha_i \left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle r^{(i)}, p^{(i)}\right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i p^{(i)}), p^{(i)}\right\rangle,$$

where $r^{(i)} = \Phi(x^{(i)}) - Ax^{(i)}$ is the nonlinear residual.

On the other hand, it is easy to see that the second derivative with respect to α_i takes the form

$$\frac{d^2 J(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i^2} = \left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i p^{(i)})p^{(i)}, p^{(i)}\right\rangle.$$

Then, using the Newton method for solving the zero-point problem for α_i, we obtain $\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$, where

$$\delta^{(k)} = \frac{dJ(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i}{d^2 J(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i^2} =$$

$$\frac{\alpha_i^{(k)} \left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle r^{(i)}, p^{(i)}\right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i^{(k)} p^{(i)}), p^{(i)}\right\rangle}{\left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i^{(k)} p^{(i)})p^{(i)}, p^{(i)}\right\rangle}.$$

Note that in order to obtain $\delta^{(k)}$, the inner products $\langle Ap^{(i)}, p^{(i)}\rangle$ and $\langle r^{(i)}, p^{(i)}\rangle$ can be computed once at the first Newton iteration. Moreover $Ap^{(i)}$ is available from the conjugate gradient iteration.

In order to generate efficient algorithms to solve the nonlinear system (1), we have designed a parallel version of Algorithm 1 and a parallel nonlinear preconditioned conjugate gradient algorithm, based on both Algorithm 1 and a polynomial preconditioner type based on block two-stage methods [3]; see [4] and [7] for detailed description.

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 (NLPCG).

Algorithm 2 *(Nonlinear Preconditioned Conjugate Gradient)*
Given an initial vector $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
Solve $Ms^{(0)} = r^{(0)}$

 ISBN: 978-84-614-6167-7

$p^{(0)} = s^{(0)}$
For $i = 0, 1, \ldots,$ *until convergence*
$\alpha_i =\rightarrow$ *see Algorithm 1*
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
Solve $Ms^{(i+1)} = r^{(i+1)}$
Convergence test
$\beta_{i+1} = -\frac{\langle s^{(i+1)}, r^{(i+1)} \rangle}{\langle s^{(i)}, r^{(i)} \rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

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, it wants M to be a good approximation of A. One of the general preconditioning techniques for solving linear systems is the use of the truncated series preconditioning [1]. These preconditioners consist 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 $s^{(0)} = 0$. It is well known that the solution of the auxiliary system $Ms = r$ is effected by $s = (I + R + R^2 + \ldots + R^{m-1})P^{-1}r$, where $R = P^{-1}Q$ and the preconditioning matrix is $M_m = P(I+R+R^2+\ldots+R^{m-1})^{-1}$, cf. [1]. In order to obtain the preconditioners, choosing $s^{(0)} = 0$, we use m steps of the block-Jacobi type two-stage methods toward the solution of $As = r$. In order to obtain the inner splittings of these block methods, incomplete LU factorizations are considered; see e.g., [4].

3 Development resources

This section analyzes the basic resources used in the building process of the designed library. The main language used for the development of the basic routines and on which the final library will be based is Fortran. However, C language is also used in order to develop CUDA-based routines. 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 and CUDA.

In order to access the routines developed in Fortran from Python, the F2PY tool (cens.ioc.ee/projects/f2py2e) has been used. These routines were developed using OpenMP extensions to run on shared memory platforms. An enhanced feature is the least influence on the behavior of the method of both the use and handling of arrays or vectors and the communication between Python and Fortran. However, two equivalent options can still be used: the Python modules for vector management *Numeric* and *numarray* (*numarray* is part of *NumPy*). The new features are not able to use

 ISBN: 978-84-614-6167-7

main routines developed in Python. Hence, vector communications between languages remain an important aspect to consider in order to achieve the best performance.

On the other hand, to access the CUDA-based routines developed in C language from Python, the PyCUDA package was used. PyCUDA is a package that offers access to Nvidia's CUDA parallel computation API from Python in such a way that it is not necessary to access to a set of CUDA-based routines included in a library to link to from Python. The PyCUDA package uses CodePy, a C/C++ metaprogramming toolkit for Python. CodePy compiles C source code and dynamically loads it into the Python interpreter, a key aspect in the nonlinearity implementation.

4 Parameters of the methods and platform

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 ($Ax = \phi(x)$), 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 as seen in Section 2. There is also a set of optional parameters to modify the NLCG and NLPCG methods. If values for the optional parameters are not specified, default values are used. The optional parameters are (see [7] for more information):

- The initial vector to start procedure (*initial_vector*).
- The stopping criterion to stop procedure (*global_stopping_error*).
- The stopping criterion to stop the iterative procedure to compute α (*alfa_stopping_error*).
- The maximum number of iterations performed in the iterative procedure to compute α (*iter_alfa*).
- Way to communicate integers from Python to Fortran or C (*trash_int*).
- Way to communicate doubles from Python to Fortran or C (*trash_double*).
- Level of the incomplete LU factorization performed in the NLPCG method (*level*).
- Number of outer iterations in the NLPCG method (*niter_2e*).
- Number of inner iterations in the NLPCG method (*val_q*).

Another important parameter, that the library can calculate, is the size of the problem assigned to each process; this is given by the parameter *block_dimensions*. This parameter is an integer vector whose dimension corresponds to the number of processes and which stores the block size assigned to each process. In the examples provided by PyPANCG, the parameter is internally calculated, such that a load balancing is achieved. On the other hand, this parameter has no relevance if CUDA is used, since

 ISBN: 978-84-614-6167-7

in this case the shared memory multiprocessor is not used even if it is available. In this case a single process manages the GPU computing.

The parameter *For_or_Py* selects the set of routines and the platform to be used. In [7] we can see the options to use in a distributed memory platform: *Python_full, Python, Fortran or Fortran_full.* The following options can be chosen with regard to this parameter in order to use a shared memory platform or a GPU:

1. *Fortran_mp*: The routines are codified in Fortran using OpenMP. Moreover ϕ and ϕ' are codified independently.
2. *Fortran_mp_full*: All of the routines are codified in Fortran using OpenMP but ϕ and ϕ' are not codified independently.
3. *GPU*: All of the routines are codified in C as CUDA kernels. Moreover ϕ and ϕ' are codified as CUDA kernels independently.

Using OpenMP or CUDA, the nonlinear functions must be codified in Fortran or C respectively. Using *Fortran_mp* or *Fortran_mp_full* implies codifying the nonlinear functions in Fortran and the compilation of the Fortran library linked to from Python. However, the use of *GPU* avoids the explicit recompilation of the nonlinear functions developed as CUDA kernels, by exploiting the CodePy features.

Finally, there are new parameters to work with the new options of the *For_or_Py* parameter, i.e. to work using OpenMP and CUDA. The first parameter, *nprocs_mp*, is the number of processes used in the shared memory platform when OpenMP is selected. The rest of the new parameters are used by CUDA. In a CUDA kernel calling, in addition to classical function parameters, there are two parameters that define the structure of threads that will be generated to run the CUDA kernel. Both parameters are the number of blocks to be generated (*grid*) and the number of threads in each block (*block*), see for example [8] to obtain detailed description. Moreover there are two global variables in order to tune the inner products, *VECTOR_N* and *ELEMENT_N*, see [4] for detailed description. Note that, both considered algorithms intensively use inner product, which is also a special operation involving a reduction process.

5 Encoding nonlinear functions

In a library for solving nonlinear systems it is important how to implement the nonlinearity of the problem to be solved. In [7] we show that PyPANCG can work either at component level and at vector level. However, when OpenMP or CUDA is used, for usability reasons and for the nature of the GPU computing, the library works at component level. The example below shows the Fortran code for the function $\phi(x)$ used in the examples of PyPANCG.

```
double precision function phi(input,trash_int,trash_double)
    implicit none
```

 ISBN: 978-84-614-6167-7

```
    real*8 input,trash_double(*),sc
    integer trash_int(*)
    sc = trash_double(1)
    phi = -sc*exp(input)
    return
```

The C CUDA kernel code to compute the same function is:

```
__device__ double Fi_x(double x,double sc){
    return (-sc*__expf(x))
}
```

We would like to note that both functions require a parameter transfer (sc) for the computation of ϕ. Using Fortran and OpenMP, in order 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 them to the problem to be solved. On the other hand, using CUDA the memory allocation and the GPU-CPU communication processes can be expensive, therefore the memory used is the strictly necessary memory. In the previous example we only communicate the necessary double parameter.

6 Python examples using OpenMP and CUDA

The use of the modules PySparNLCG and PySParNLPCG using OpenMP or CUDA is closely similar to earlier version presented in [7]. In order to use the library the size of the system (nrow), the matrix A in CSR format (tcol, trow, tval), and the nonlinear functions (ϕ and ϕ') must be passed at the very least, and optionally, the block size assigned to each process (block_dimensions). Moreover, as we have mentioned, additional parameters, if needed, can be passed by using the variables *trash_int* and *trash_double*. The following code shows the most simple NLCG function call using OpenMP.

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

5  nprocs = 4
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=nrow,d=nrow)
```

 ISBN: 978-84-614-6167-7

```
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, _
              For_or_Py='Fortran_mp',trash_double = trash_double,nprocs_mp=nprocs)
```

The number of processes to use is established in line 5. Note that the number of processes must be fixed before the block structure be defined in line 9. The matrix A is obtained in line 10 following the block structure above defined. This matrix is included in PyPANCG as an example or test. It is important to point out that in line 10 root process computes the full matrix A, we maintain the *PartialMatrixA* as routine name in order to accomplish the compatibility. 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 developed in Fortran and the vector *trash_double* is passed, in this case of a single component. Note that OpenMP is not used in the development of *Fi_x* (ϕ) and *Fi_prime_x* (ϕ') because they are implemented at component level.

The most simple NLPCG using an OpenMP function call is similar to the NLCG example above showed. In this case, the PySParNLPCG module must be imported instead of the PySParNLCG module in line 4, and line 11 must be modified by the main function of the PySParNLPCG module.

```
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, _
              For_or_Py='Fortran_mp',trash_double = trash_double,procs_mp=nprocs)
```

The following example shows the simplest example to call NLCG method using CUDA.

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

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

10 nnz,tcol,trow,tval = PyPANCG.PartialMatrixA _
       (Mx=Mx,s=nrow,d=nrow)

11 x,error,time,iter = PySParNLCG.nlcg(nrow=nrow,tcol=tcol,trow=trow,tval=tval, _
              Fi_x=0,Fi_prime_x=0,For_or_Py='GPU', _
              trash_double = trash_double,nprocs_mp=nprocs)
```

In line 5 the *PyPANCG.PySparNLCG_ModGPU* module is imported. This module contains all CUDA kernels needed by the NLCG method, including *Fi_x* (ϕ) and

 ISBN: 978-84-614-6167-7

Fi_prime_x (ϕ'). Note that the encoding of the nonlinear functions must be done in that module without any compiling process. In the NLCG method, the CPU only performs the management of the GPU, therefore only one process is used (see line 6). The full matrix A is computed in line 10 by the CPU. In line 11, the call to the NLCG method to be computed in the GPU takes place. Note that the nonlinear functions *Fi_x* (ϕ) and *Fi_prime_x* (ϕ') are not defined in this call because, as we have mentioned, they are included inside CUDA kernels from *PyPANCG.PySparNLCG_ModGPU* module.

Finally, the most simple NLPCG function call, using CUDA, is similar to the previous example. In this case the PySPArNLPCG module must be imported instead of the PySParNLCG module in line 4. The *PyPANCG.PySparNLPCG_ModGPU* module containing the CUDA kernels used by the NLPCG method, is also imported in line 5.

```
4  import PyPANCG.PySPArNLPCG as PySparNLPCG
5  import PyPANCG.PySparNLPCG_ModGPU as PySPArNLPCG_ModGPU
```

The main function for the NLPCG method takes, in this case, the following form:

```
11 x,error,time,iter = PySPArNLPCG.nlpcg(nrow=nrow,tcol=tcol,trow=trow,tval=tval, _
              Fi_x=0,Fi_prime_x=0,For_or_Py='GPU', _
              trash_double = trash_double,procs_mp=nprocs)
```

In [4] some aspects of GPU computing are pointed out in order to tune the performance of the NLCG and NLPCG methods. Essentially this improvements are related to the inner products computation and the number of threads by block in the CUDA kernels. The following lines show an improvement of the NLPCG method with the use of some parameters (grid and block) as described in [4].

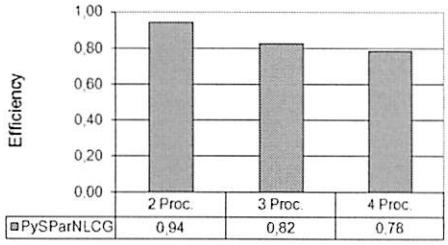
```
11 if (nrow == 125000)
              VECTOR_N = 128
              ELEMENT_N = 2916
              grid = (1458,1,1)
              block = (256,1)
12 x,error,time,iter = PySPArNLPCG.nlpcg(nrow=nrow,tcol=tcol,trow=trow,tval=tval, _
              Fi_x=0,Fi_prime_x=0,For_or_Py='GPU', _
              trash_double = trash_double,procs_mp=nprocs, _
              block=block,grid=grid)
```

7 Numerical experiments

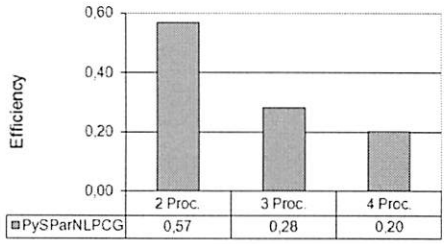
In order to illustrate the behavior of PyPANCG, we have tested the algorithms provided by this library on an Intel Core 2 Quad Q6600, 2.4 GHz, with 4 GB of RAM, called SULLI. The GPU available in SULLI is a GeForce GTX 280. The performed analysis is based on the run-times measured on the GeForce GTX 280, and on the parallel run-times measured on SULLI using OpenMP, when pure Fortran code (using OpenMP) or pure C code (using CUDA) are used, compared with the times obtained by PyPANCG.

As our illustrative example we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem. In this problem, heat generation from

 ISBN: 978-84-614-6167-7



(a) PySParNLCG



(b) PySParNLPCG

Figure 1: Efficiency using OpenMP, $n = 373248$.

a combustion process is balanced by heat transfer due to conduction. The three-dimensional model problem is given as

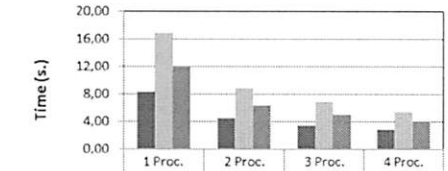
$$\nabla^2 u - \lambda e^u = 0, \tag{4}$$

where u is the temperature and λ is a constant known as the Frank-Kamenetskii parameter; see e.g., [2]. There are two possible steady-state solutions to this problem for a given value of λ. One solution is close to $u = 0$ and it is easy to obtain. A starting point near to the other solution is needed to converge to it. For our model case, we consider a 3D cube domain Ω of unit length and $\lambda = 6$. To solve equation (4) using the finite difference method, we consider a grid in Ω of d^3 nodes. This discretization yields a nonlinear system of the form $Ax = \Phi(x)$, where $\Phi : \Re^n \to \Re^n$ is a nonlinear diagonal mapping, i.e., the ith component Φ_i of Φ is a function only of the ith component of x. The matrix A is a sparse matrix of order $n = d^3$ and the typical number of nonzero elements per row of this matrix is seven, with fewer in rows corresponding to boundary points of the physical domain.

First, we analyze the efficiency for both methods using OpenMP. The NLCG method is performed by PySParNLCG module and the NLPCG method is performed by PySParNLPCG module. Optimal values of the parameters are used when the NLPCG method is computed, these parameters are the level of fill-in of the incomplete LU factorization (*level*), the number of outer iterations of the block two-stage method (*niter_2e*), and the number of inner iterations of the block two-stage method (*val_q*). On the other hand, in all experiments reported here the values of *global_stopping_error* and *alfa_stopping_error* are 10^-7, and we set *iter_alfa* parameter equal to 2. Figure 1 shows the efficiency of both methods using OpenMP and up to 4 cores available in SULLI. The efficiency behavior of the methods is not influenced by the use of the Python library; a pure Fortran code obtains similar efficiencies. For the NLCG method we obtain a good efficiency with a slight decrease when the number of processes is increased. However, as we showed in [6], the NLPCG method is a very good algorithm but with poor scalability, even for very large systems.

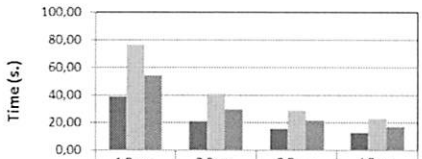
In order to select OpenMP, we have two options for parameter *For_or_Py* (see Section 4). In Figure 2 we can observe the behavior for both options. Setting *For-*

 ISBN: 978-84-614-6167-7



	1 Proc.	2 Proc.	3 Proc.	4 Proc.
Pure Fortran	8,33	4,43	3,45	2,87
Fortran_mp	16,83	8,92	6,81	5,36
Fortran_mp_full	11,96	6,4	4,94	4,03

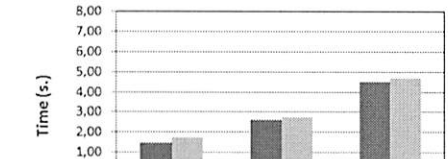
(a) 125000



	1 Proc.	2 Proc.	3 Proc.	4 Proc.
Pure Fortran	39,20	21,27	15,59	12,91
Fortran_mp	76,60	40,76	28,97	22,99
Fortran_mp_full	54,18	29,41	21,38	16,93

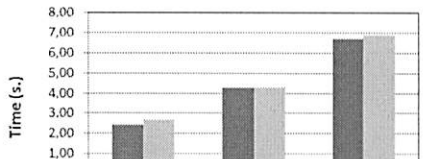
(b) 373248

Figure 2: PySparNLCG using OpenMP.



	n=373248	n=592704	n=884736
Pure CUDA	1,48	2,64	4,51
Python + CUDA	1,76	2,78	4,71

(a) PySParNLCG



	n=373248	n=592704	n=884736
Pure CUDA	2,46	4,26	6,70
Python + CUDA	2,66	4,30	6,88

(b) PySParNLPCG

Figure 3: PyPANCG using CUDA.

tran_mp_full option we obtain results closer to those obtained with pure Fortran than using *Fortran_mp* option. Note that the development effort using *Fortran_mp_full* is higher than using *Fortran_mp*.

Finally, we compare the results obtained by both modules when CUDA is used. Concretely, Figure 3 shows the results of these PyPANCG modules compared with a pure CUDA code for several problem sizes. As it can be appreciated both implementations report similar execution times.

8 Conclusion

In this paper we have presented new features of PyPANCG, a Python library-interface that implements both the conjugate gradient method and the preconditioned conjugate gradient method for solving nonlinear systems. 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 new features are designed to allow PyPANCG to be able to work on both shared memory platforms and GPUs. We have described the use of the library and its advantages in order to get fast development. The library has been designed for

 ISBN: 978-84-614-6167-7

adapting to different stages of the design process, depending on whether the purpose is computational performance or fast development. We have achieved both objectives at once using the GPU as a computation platform, which is also the platform on which the proposed algorithms have better performance.

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. BRU, V. MIGALLÓN, J. PENADÉS AND D.B. SZYLD, *Parallel, Synchronous and Asynchronous Two-Stage Multisplitting Methods*, Electronic Transactions on Numerical Analysis **3** (1995) 24–38.

[4] V. GALIANO, H. MIGALLÓN, V. MIGALLÓN AND J. PENADÉS, *GPU-Based Parallel Nonlinear Conjugate Gradient Algorithms*, Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (2011) Paper 24.

[5] R. FLETCHER AND C. REEVES, *Function Minimization by Conjugate Gradients*, The Computer Journal **7** (1964) 149–154.

[6] H. MIGALLÓN, V. MIGALLÓN, J. PENADÉS, *Parallel Nonlinear Conjugate Gradient Algorithms on Multicore Architectures*, Proceedings of the 9th International Conference on Computational and Mathematical Methods in Science and Engineering (2009) 689–700.

[7] H. MIGALLÓN, V. MIGALLÓN AND J. PENADÉS, *PyPANCG: A Parallel Python Interface-Library for solving Mildly Nonlinear Systems*, Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering (2010) 646–657.

[8] NVIDIA CORPORATION, *NVIDIA CUDA C Programming Guide, Version 3.2, 2010*, `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`

 ISBN: 978-84-614-6167-7