Módulo para la librería PyPANCG de resolución de sistemas no lineales dispersos en paralelo mediante precondicionadores

Héctor Migallón
Dpto. de Física y
Arquitectura de Computadores
Universidad Miguel Hernández
hmigallon@umh.es

Violeta Migallón
Dpto. de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
violeta@dccia.ua.es

José Penadés
Dpto. de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
jpenades@dccia.ua.es

Resumen

En este artículo presentamos el desarrollo de un nuevo módulo de la librería paralela PyPANCG, el módulo PySParNLPCG. Esta librería, que está desarrollada como una interfaz de alto nivel, consta de diferentes módulos para la resolución de sistemas no lineales dispersos. Concretamente, el nuevo módulo implementa el método del gradiente conjugado no lineal precondicionado. Este trabajo sigue las pautas de una interfaz de alto nivel en términos de usabilidad, pero realmente se trata de una librería que ha sido desarrollada siguiendo un modelo mixto a nivel de utilización de lenguajes de programación. Por otra parte, la librería está diseñada para adaptarse a las diferentes etapas de diseño, en las cuales el objetivo principal puede ser el rendimiento o puede ser la velocidad de desarrollo. Expondremos las características de los algoritmos paralelos desarrollados en este trabajo y se relatarán los diferentes modos de trabajo que permite dicho módulo, ilustrando su funcionamiento mediante algunos ejemplos y resultados experimentales.

1. Introducción

Este trabajo está orientado al uso del lenguaje de programación *Python* [13], el cual es un lenguaje tipo *script* que está teniendo una gran aceptación entre la comunidad dedicada a las ciencias computacionales. Este tipo de lenguajes, entre los que también podemos mencionar *Perl*, *Ruby* o *Tcl*, acortan el tiempo de desarrollo y permiten una alta integración de códigos individuales de dimensión considerable. Características comunes de dichos lenguajes son la simplicidad, la potencia y la búsqueda de una alta integración debido a que su rendimiento natural no es muy alto.

Se ha decidido desarrollar PyPANCG y sus diferentes módulos con Python, entre otros motivos por la disponibilidad de diferentes herramientas para automatizar el desarrollo de interfaces de acceso a rutinas desarrolladas en lenguaje Fortran (y también en lenguaje C), y por la posibilidad de trabajar con diferentes herramientas para gestionar el entorno paralelo a través de *MPI* [7]; entre estas utilidades podemos mencionar *PyMPI* [8] o *mpipython* incluido en *Scientific Python* [5]. Además existen múltiples interfaces a librerías de computación científica, por ejemplo *PyACTS*, *Visual Python*, *PyTrilinos* o *PyPLiC*. Pero también son importantes las herramientas que facilitan el desarrollo de dichas librerías científicas, por ejemplo *numarray* y *SciPy* y su paquete fundamental *NumPy* [6], *Scientific Python* [5] o *PyIMSL*.

PyPANCG es una librería Python para la resolución de sistemas no lineales dispersos que se distribuye como un paquete Python estándar. El nuevo módulo PySParNLPCG hace uso del método del gradiente conjugado precondicionado para resolver en paralelo un sistema no lineal disperso del tipo $F(x) = Ax - \phi(x) = 0$. La función ϕ debe cumplir que la componente l-ésima de $\phi(x)$ dependa úni-

camente de la componente l-ésima de x. Los algoritmos desarrollados están basados en la versión del método del gradiente conjugado de Fletcher-Reeves y en precondicionadores polinomiales construidos a partir del método por bloques en dos etapas.

Dadas las características del problema, la librería que ha de generarse no puede ser una librería estática, debe ser una librería dinámica que ha de adaptarse al problema no lineal a resolver. En la sección 2 se introduce el método del gradiente conjugado no lineal precondicionado utilizado en la librería. En la sección 3 se describe la paralelización del método y en las secciones 4, 5 y 6 se detalla el módulo PySParNLPCG (véase http://atc.umh.es/PyPANCG), analizando las herramientas básicas utilizadas, los parámetros y las distintas formas de implementación de la no linealidad. Por último se muestran algunos ejemplos de utilización de la librería y los resultados obtenidos en los experimentos numéricos (véase secciones 7 y 8).

2. Gradiente conjugado no lineal precondicionado (NLPCG)

En [9] se presentó el desarrollo del módulo de la librería PyPANCG que implementa el método del gradiente conjugado para la resolución en paralelo de sistemas no lineales dispersos de la forma

$$F(x) = Ax - \phi(x) = 0. \tag{1}$$

El precondicionamiento es una técnica para mejorar el número de condición (cond) de una matriz. Concretamente, si suponemos que M es una matriz simétrica y definida positiva que aproxima a A y cuya inversa es fácil de obtener, a través de M puede resolverse indirectamente el sistema $Ax = \Phi(x)$ resolviendo el sistema $M^{-1}Ax = M^{-1}\Phi(x)$. Además si $\text{cond}(M^{-1}A) << \text{cond}(A)$ la resolución del sistema $M^{-1}Ax = M^{-1}\Phi(x)$ es más rápida que la resolución del sistema original. En base a esto obtenemos el siguiente algoritmo del método del gradiente conjugado precondicionado (NLPCG) para sistemas no lineales.

`Dado un vector inicial` $x^{(0)}$
  $r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
  `Resolver` $Ms^{(0)} = r^{(0)}$
  $p^{(0)} = s^{(0)}$
  `Para i=0,1,..., hasta convergencia`
    $\alpha_i =\rightarrow$ `véase [9]`
    $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
    $r^{(i+1)} = r^{(i)} - \Phi(x^{(i)})$
        $+\Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
    `Resolver` $Ms^{(i+1)} = r^{(i+1)}$
    `Test de convergencia`
    $\beta_{i+1} = -\frac{\langle s^{(i+1)}, r^{(i+1)} \rangle}{\langle s^{(i)}, r^{(i)} \rangle}$
    $p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

Algoritmo 1: GC no lineal precondicionado

Resolver el sistema auxiliar $Ms = r$ debe ser computacionalmente ligero, dado que se tiene que resolver en cada iteración. Además, para obtener un buen precondicionador, M debe ser una buena aproximación de A. En este trabajo, para la obtención del precondicionador nos hemos basado en el precondicionamiento mediante series truncadas [1], que consiste en considerar una particion de la matriz A

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

y realizar m pasos del procedimiento iterativo generado por dicha partición para obtener la solución de $As = r$, tomando $s^{(0)} = 0$. La solución del sistema $Ms = r$ se obtiene a través de $s = (I + R + R^2 + \ldots + R^{m-1})P^{-1}r$, donde $R = P^{-1}Q$ y la matriz precondicionadora es $M_m = P(I + R + R^2 + \ldots + R^{m-1})^{-1}$ [1].

En nuestro caso, el precondicionador se ha obtenido realizando m iteraciones de un método por bloques en dos etapas para obtener la solución de $As = r$, tomando como iterado inicial $s^{(0)} = 0$ [4].

Si consideramos que la matriz A está particionada en $p \times p$ bloques, con bloques diagonales de orden n_j, $\sum_{j=1}^{p} n_j = n$, el sistema (1) puede expresarse de la siguiente forma

$$\begin{bmatrix} A_{11} & \cdots & A_{1p} \\ A_{21} & \cdots & A_{2p} \\ \vdots & \vdots & \vdots \\ A_{p1} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \Phi_1(x) \\ \Phi_2(x) \\ \vdots \\ \Phi_p(x) \end{bmatrix} \tag{3}$$

donde x y $\Phi(x)$ están divididos en función de los bloques de A. Consideramos las particiones (2) tal que

$$P = \mathrm{diag}(A_{11}, \ldots, A_{pp}). \qquad (4)$$

En este caso, la realización de m iteraciones del proceso iterativo definido por la partición (2) para aproximar la solución de $As = r$, consiste en la realización de m iteraciones del método de Jacobi por bloques. De este modo, en cada iteración l, $l = 1, 2, \ldots$, del método de Jacobi por bloques, se tienen que resolver p sistemas lineales independientes

$$A_{jj} s_j^{(l)} = (Qs^{(l-1)} + r)_j, \quad 1 \leq j \leq p. \qquad (5)$$

Cada uno de los sistemas lineales (5) puede resolverse en un procesador diferente. Sin embargo, cuando el orden de los bloques diagonales A_{jj}, $1 \leq j \leq p$ es grande, es conveniente utilizar un nuevo proceso iterativo para aproximar la solución en lugar de resolverlo. En esto básicamente consiste el método iterativo en dos etapas, véase por ejemplo [3]. El siguiente algoritmo explica este método, donde se han considerado las particiones $A_{jj} = B_j - C_j, 1 \leq j \leq p$, y en cada iteración l, para cada j, $1 \leq j \leq p$, se realizan $q(j)$ iteraciones, del procedimiento iterativo definido por dichas particiones, para aproximar la solución del sistema (5).

Dados $s^{(0)} = \left((s_1^{(0)})^T, \ldots, (s_p^{(0)})^T\right)^T$
y $q(j)$, $1 \leq j \leq p$
  Para $l = 1, 2, \ldots$, hasta convergencia
    En el procesador j, $j = 1, 2, \ldots, p$
      $y_j^{(0)} = s_j^{(l)}$
      De $k = 1$ hasta $q(j)$
        $B_j y_j^{(k)} = C_j y_j^{(k-1)} + (Qs^{(l-1)} + r)_j$
      $s^{(l)} = \left((y_1^{(q(1))})^T, \ldots, (y_p^{(q(p))})^T\right)^T$

Algoritmo 2: Dos etapas por bloques

Hacemos notar que para resolver el sistema auxiliar $Ms = r$ del algoritmo 1, se realizan m iteraciones $s^{(l)} = Ts^{(l-1)} + W^{-1}r$, $l = 1, 2, \ldots, m$, tomando $s^{(0)} = 0$ y

$$T = H + (I - H)P^{-1}Q, \quad W = P(I - H)^{-1},$$

con P definida en (4) y $H = \mathrm{diag}((B_1^{-1}C_1)^{q(1)}, \ldots, (B_p^{-1}C_p)^{q(p)})$; véase por ejemplo [11].

3. Algoritmo Paralelo

Consideramos que los vectores $x^{(i)}$, $r^{(i)}$, $s^{(i)}$, $p^{(i)}$ y $\Phi(x^{(i)})$ están particionados de acuerdo con la estructura por bloques de A vista en (3). Por ejemplo, el vector $x^{(i)}$ está particionado en subvectores, denotados por $x_j^{(i)}$, $j = 1, 2, \ldots p$, cada uno en $\Re^{n_j}$. Con esta notación el algoritmo paralelo del gradiente conjugado no lineal precondicionado es el siguiente.

Dado un vector inicial $x^{(0)}$
  En el procesador j, $j = 1, 2, \ldots, p$
    $r_j^{(0)} = \Phi_j(x^{(0)}) - [A_{j1} \ A_{j2} \cdots A_{jp}]x^{(0)}$
  Realizar m pasos del Alg. 2
    para aproximar $As^{(0)} = r^{(0)}$
  $p^{(0)} = s^{(0)}$
  Para i=0,1,..., hasta convergencia,
    En el procesador j, $j = 1, 2, \ldots, p$
      $\alpha_i =\rightarrow$ véase [9]
      $x_j^{(i+1)} = x_j^{(i)} + \alpha_i p_j^{(i)}$
      $r_j^{(i+1)} = r_j^{(i)} - \Phi_j(x^{(i)}) + \Phi_j(x^{(i+1)})$
        $-\alpha_i[A_{j1} \ A_{j2} \cdots A_{jp}]p^{(i)}$
    Realizar m pasos del Alg. 2
      para aproximar $As^{(i+1)} = r^{(i+1)}$
    Test de convergencia
    En el procesador j, $j = 1, 2, \ldots, p$
      $\vartheta_j = \langle s_j^{(i+1)}, r_j^{(i+1)} \rangle$
      $\sigma_j = \langle s_j^{(i)}, r_j^{(i)} \rangle$
    Proc. 1 calcula y distribuye
      $\beta_{i+1} = -\sum_{j=1}^{p} \vartheta_j / \sum_{j=1}^{p} \sigma_j$
    En el procesador j, $j = 1, 2, \ldots, p$
      Calcula y sincroniza
        $p_j^{(i+1)} = r_j^{(i+1)} - \beta_{i+1} p_j^{(i)}$

Algoritmo 3: Método NLPCG paralelo

En [9] puede verse una descripción detallada del cálculo de α, y la paralelización de dicho cálculo.

4. Herramientas básicas de PySParNLPCG

En esta sección analizaremos las herramientas básicas utilizadas para el desarrollo del módulo PySParNLPCG de la librería PyPANCG. La librería desarrollada debe permitir el desarrollo de aplicaciones con buenos rendimientos computacionales, por ello es necesario disponer de herramientas que permitan acceder desde Python a librerías realizadas en otros lenguajes que obtengan mejor rendimiento. En este caso, el lenguaje utilizado para el desarrollo de las rutinas básicas, en las que se apoyará la librería final, ha sido el lenguaje Fortran. Con este objetivo se han desarrollado rutinas equivalentes en ambos lenguajes; en la sección 5 se describe cómo seleccionar unas u otras rutinas. Además, se han desarrollado rutinas mixtas que trabajan con ambos lenguajes a diferentes niveles. Para poder acceder desde Python a las rutinas desarrolladas en Fortran se ha hecho uso de la herramienta F2PY [12], que automatiza el proceso de realización de las interfaces necesarias para la conexión entre Fortran y Python.

El manejo del entorno paralelo debe ser transparente al usuario para que la utilización de la librería desde el nivel superior de Python no presente complicaciones. Para ello haremos uso de herramientas que permitan trabajar con MPI [7], desde Python. Para aumentar los posibles entornos de trabajo se ha desarrollado la librería para que pueda trabajar con dos de las herramientas más comunes, éstas son *mpipython*, que forma parte de *Scientific Python* [5], y *pyMPI* [8].

Otro aspecto muy importante, tanto para la comunicación entre Python y Fortran como para el rendimiento, es el uso y manejo de los *arrays* o vectores, para ello podremos utilizar dos opciones igualmente. El uso de una herramienta u otra está directamente relacionado con la herramienta utilizada para la gestión del entorno paralelo. En caso de utilizar *mpipython* será necesario trabajar con *Numeric*, y en el caso de utilizar *pyMPI* se trabajará con *numarray*.

5. Parámetros de PySParNLPCG

En esta sección se verán los parámetros que deben pasarse a la función Python que hace uso del método paralelo NLPCG para resolver un sistema no lineal. Los únicos parámetros indispensables son los propios del sistema a resolver ($Ax = \phi(x)$), que son el tamaño del sistema, la matriz A en formato CSR (Compressed Sparse Row) y la función $\phi(x)$. Además, para el cálculo de α en el algoritmo 3 (véase [9]), se necesita la derivada de la función $\phi(x)$ ($\phi'(x)$).

Con el fin de optimizar el uso de la memoria, cada procesador almacenará únicamente la porción de matriz que necesita para calcular las componentes de los vectores asociados a dicho procesador. En el caso de disponer de toda la matriz en un sólo procesador será necesario distribuir a cada procesador la porción de matriz asociada.

En caso de no ser especificados se utilizan valores por defecto. Concretamente, los parámetros y su valor por defecto son:

- *initial_vector*: iterado inicial igual a cero.
- *global_stopping_error* $\xi = 10^{-7}$: criterio de parada evaluado sobre la norma euclídea del vector residuo ($\|r\|_2$).
- *alfa_stopping_error* $\zeta = 10^{-7}$: criterio de parada del cálculo de α, véase [9].
- *iter_alfa* = 0: fijando este parámetro a un valor superior a 1 podemos limitar el número de iteraciones realizadas para calcular α o podemos utilizar un número de iteraciones fijo.
- *For_or_Py* = "Python_full": todas las rutinas utilizadas están desarrolladas en Python.
- *trash_int*, array de enteros (véase sección 6).
- *trash_double*, array de reales en doble precisión (véase sección 6).
- *level* = 1: nivel de llenado de la factorizacion ILU. En el algoritmo 2 se realiza

una particion mediante el uso de factorizaciones ILU(S), donde S es el nivel de llenado (ver [10]).

- *niter_ 2e* = 3: en el algoritmo 3 se realizan m (i.e. niter_2e) iteraciones del algoritmo 2 para aproximar el sistema lineal.
- *val_ q* = 3: el parámetro $q(j)$ (i.e val_q) en el algoritmo 2 es el número de iteraciones realizado en el proceso iterativo.

El parámetro *block_ dimensions* es un array cuyo tamaño coincide con el número de procesadores, que almacena los tamaños n_j del problema asignado a cada procesador. En el problema ejemplo de PySParNLPCG se calcula internamente en función del problema particular a resolver.

El parámetro *For_ or_ Py* selecciona las rutinas que se utilizarán durante el desarrollo del método. Respecto a este parámetro podemos seleccionar cuatro opciones:

1. *Python_ full*: todas las rutinas utilizadas están codificadas en Python.
2. *Python*: las rutinas utilizadas están codificadas en Python pero las funciones provenientes de SPARSKIT y BLAS lo están en Fortran.
3. *Fortran*: todas las rutinas utilizadas están codificadas en Fortran, y se utilizan funciones en Fortran para calcular tanto ϕ como ϕ'.
4. *Fortran_ full*: todas las rutinas utilizadas están codificadas en Fortran, y además las funciones ϕ y ϕ' no están codificadas independientemente.

Las opciones listadas están en orden de menor a mayor en términos de rendimiento, y de mayor a menor en términos de usabilidad y velocidad de desarrollo. Cabe destacar que la opción *Python* es mixta, mientras que el resto de opciones o utilizan Python o utilizan Fortran para las rutinas básicas. La diferencia entre las opciones *Fortran* y *Fortran_ full* es que en la primera opción el usuario deberá codificar en Fortran únicamente las funciones ϕ y ϕ', mientras que en la segunda deberá hacer la codificación en Fortran de todas las rutinas que implementan dichas funciones y por tanto conocer en profundidad el desarrollo interno del método.

Los parámetros *alfa_ stopping_ error* e *iter_ alfa* pueden trabajar independiente o conjuntamente. Si alguno de ellos se fija a 0 será utilizado el otro parámetro, pero se utilizarán ambos si ambos son fijados. En este caso se realizarán iteraciones hasta cumplir el criterio de parada dado por *alfa_ stopping_ error*. No obstante si se realizan *iter_ alfa* iteraciones sin alcanzar el criterio de parada se finalizará el proceso iterativo para el cálculo de α. Por último, los dos últimos parámetros, o variables en este caso, *trash_ int* y *trash_ double*, pueden ser necesarios para implementar la no linealidad del sistema (véase la sección 6).

6. Implementación de la no linealidad

Uno de los mayores obstáculos para la realización de librerías para la resolución de sistemas no lineales es la implementación que debe realizar el usuario de la no linealidad del problema a resolver. Si dicha implementación implica el desarrollo casi completo del algoritmo la librería no actúa como tal. Tal y como se ve en el algoritmo 3 del método NLPCG, la no linealidad, es decir ϕ y ϕ', son utilizadas en el cálculo de los vectores residuo $r^{(i)}$, y como se ha comentado, también son utilizadas en el cálculo de α.

Un aspecto importante es que la componente l-ésima de $\phi(x)$, únicamente depende de la componente l-ésima de x. Por tanto, las funciones ϕ y ϕ' pueden desarrollarse a nivel vector pero también a nivel de componente de vector. Atendiendo a razones de prestaciones se desarrollará a nivel de vector si dicho desarrollo se realiza en Python, y atendiendo a razones de usabilidad se realizará a nivel de componente en Fortran.

Podemos ver en el ejemplo siguiente el código Python de la función $\phi(x)$ utilizada en el ejemplo de PySParNLPCG.

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

Esa misma función desarrollada en Fortran es:

```
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
```

Además de observar que el código Python opera sobre vectores, mientras que el código Fortran opera sobre un único elemento, se debe observar que a ambas funciones se pasa un parámetro (sc) necesario para el cómputo de ϕ. Para realizar esta transferencia, tanto de valores reales como de valores enteros, utilizamos dos arrays, uno de enteros (trash_int) y otro de reales en doble precisión (trash_double). Estos arrays son dinámicos y por tanto pueden pasarse a las funciones ϕ y ϕ' todos los parámetros necesarios para su cómputo. Lógicamente, estas funciones deben implementarse siempre para adecuarse al problema a resolver. Si se implementan en Python se deberá utilizar la opción *Python* o *Python_full*. Si se implementan en Fortran se deberá utilizar la opción *Fortran* o *Fortran_full*. Además, en este último caso, tras el desarrollo de las funciones se deberá instalar y compilar el módulo de nuevo.

Las opciones *Python* y *Fortran* son muy similares, ambas utilizan funciones básicas en Fortran y difieren en la implementación de las funciones ϕ y ϕ'. La opción *Fortran_full* no utiliza las funciones ϕ y ϕ', sino que las integra en las rutinas que utilizan dichas funciones, por tanto su adecuación es más complicada y laboriosa pero es la opción que mejor rendimiento presenta. Por último, la opción *Python_full* no utiliza ningún código Fortran, lo que permite un desarrollo mucho más rápido pero un rendimiento excesivamente pobre.

7. Ejemplos de PySParNLPCG

Como se ha comentado, para utilizar la librería es necesario pasar al menos el tamaño del sistema (nrow), la matriz en formato CSR (tcol, trow, tval), el tamaño asignado a cada procesador (block_dimensions) y las funciones que implementan la no linealidad (ϕ, ϕ'), pero además si deseamos pasar parámetros utilizaremos las variables trash_int y trash_double. En el siguiente código podemos observar la llamada más sencilla, en la cual suponemos que las funciones ϕ y ϕ' han sido definidas en Python previamente,

```
1  from math import exp
2  import numpy
3  import PyPANCG
4  import PyPANCG.PySParNLPCG as PySparNLPCG

5  iam = PySParNLPCG.iam
6  trash_double = numpy.zeros(((1),),float)
7  trash_double[0] = 6/(float(71)**3)
8  Mx = 72
9  nrow = 72*72*72

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

12 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)
```

En las líneas 10 y 11 se obtiene la matriz del sistema a resolver, dicho código se adjunta en la librería pero son utilizables como ejemplo y test únicamente. Hay que remarcar que cada procesador contiene únicamente la porción de matriz que necesita. En la línea 12 se hace la llamada propia al método, en el cual se supone que se han declarado las funciones *Fi_x* (ϕ) y *Fi_prime_x* (ϕ') en Python, y se pasa el array, en este caso de un sólo elemento, *trash_double*.

Si no se desea utilizar los valores por defecto se podría sustituir, por ejemplo, la línea 12 por

```
12 x,error,time,iter=PySParNLCG.nlpcg(nrow=nrow,_
        tcol=tcol,trow=trow,tval=tval, _
        block_dimensions = block_dimensions _
        Fi_x=Fi_x,Fi_prime_x=Fi_prime_x, _
        alfa_stopping_error=1e-8, _
        iter_alfa=5,For_or_Py='Fortran', _
        trash_double = trash_double)
```

en la cual el criterio de parada en el cálculo de α es 10^{-8}, pero realizará 5 iteraciones como máximo, y serán utilizadas las rutinas Fortran que hacen uso de ϕ y ϕ' desarrolladas en Fortran.

8. Experimentos numéricos

Los experimentos numéricos han sido realizados sobre dos multiprocesadores de memoria compartida, el primero es SULLI compuesto por un procesador con 4 núcleos Intel Core 2 Quad Q6600, el segundo, Bi-Quad, es un DELL PowerEdge 2900 con dos Quad Core Intel Xeon 5320. Se ha hecho uso de la librería MPI para el manejo del entorno paralelo. El sistema operativo es Ubuntu 8.04 (Hardy Heron) para sistemas de 64 bits.

Para el estudio del comportamiento de la librería tratada, hemos utilizado un problema que corresponde con una ecuación diferencial elíptica no lineal, conocida como el problema de Bratu. Esta ecuación modela un proceso de reacción térmica en un material rígido, en el que el proceso depende de un equilibrio entre el calor generado químicamente y la transferencia de calor por conducción.

Concretamente este problema viene dado por

$$\nabla^2 u - \lambda e^u = 0, \tag{6}$$

donde u es la temperatura y λ una constante conocida como el parámetro de Frank–Kamenetskii (véase por ejemplo [2]). En los experimentos hemos considerado un dominio cúbico 3D de longitud uno y $\lambda = 6$. El sistema $Ax - \phi(x) = 0$ así obtenido cumple que A es una matriz dispersa y que la l-ésima componente de $\phi(x)$ depende únicamente de la componente l-ésima de x. Presentamos resultados correspondientes a sistemas no lineales de tamaños 373248 y 592704.

En primer lugar analizamos el comportamiento de las diferentes opciones que ofrece la librería en función del parámetro *For_or_Py*. En la figura 1 se puede observar que la opción en la cual está desarrollado todo en Fortran presenta los mejores resultados, mientras que la opción que utiliza únicamente código Python presenta unos resultados peores

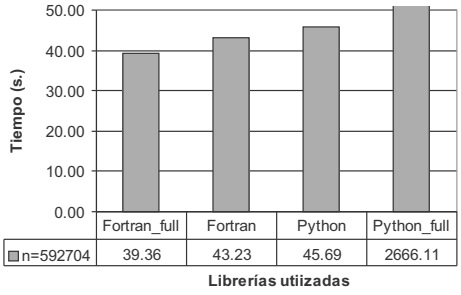

Figura 1: Tiempos PySParNLPCG con 2 procesadores. Parámetros: level=1, niter_2e=2, val_q=3, mpipython, orden 592704. SULLI

pero esperados. Esta opción sólo es válida en un fase de desarrollo. Éste es el comportamiento habitual y lógico de la librería PyPANCG. Por otra parte, podemos ver que las opciones que mezclan código Fortran y Python y desarrollan la no linealidad en Python y en Fortran presentan resultados muy similares, siendo algo mejor los resultados con la opción en la que se desarrolla la no linealidad en Fortran. Esto puede cambiar al variar el orden del sistema, teniendo en cuenta que en Python se trabaja a nivel vector en lugar de a nivel componente en el cómputo de ϕ y ϕ'.

En la figura 2 podemos observar el comportamiento atendiendo al número de procesadores utilizado. En dicha figura se observa que se obtiene una buena eficiencia tanto para 2 como para 3 procesadores, pero que decrece para 4 procesadores, por tanto debe utilizarse el número de procesadores adecuado.

Por último, mencionar que hay una lógica pérdida de rendimiento debido a la diferencia existente entre un lenguaje compilado como Fortran y un lenguaje interpretado como Python. No obstante el beneficio que se obtiene en términos de velocidad de desarrollo y sobre todo de integración puede justificar sin duda dicha pérdida.

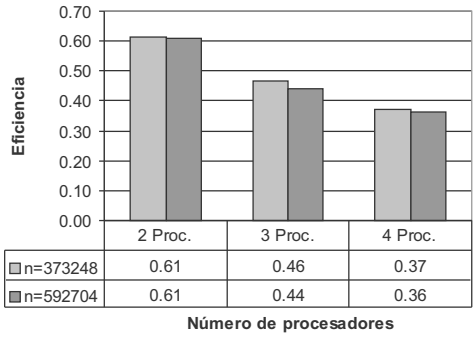

Figura 2: Eficiencia. Parámetros: level=1, niter_2e=1, val_q=1, pyMPI. Bi-Quad

Agradecimientos

El presente trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación mediante el proyecto TIN2008-06570-C04-04, y por la Universidad de Alicante mediante el proyecto VIGROB-020.

Referencias

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

[2] Averick, B.M., Carter, R.G., More, J.J., Xue, G., *The MINPACK-2 test problem collection*, Tech. Rep. MCS-P153- 0692, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[3] Bru, R., Migallón, V., Penadés, J., Szyld, D.B., *Parallel, Synchronous and Asynchronous Two-Stage Multisplitting Methods*, Electronic Transactions on Numerical Analysis, 3:24–38, 1995.

[4] Castel, M.J., Migallón, V., Penadés, J., *Block two-stage preconditioners*, Applied Mathematics Letters, 14(5):625–630, 2001.

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

[6] Jones, E., Oliphant, T., Peterson, P., et al., *SciPy: Open source scientific tools for Python*, `http://www.scipy.org/`.

[7] *Message Passing Interface*, `http://www-unix.mcs.anl.gov/mpi/`.

[8] Miller, P., *PyMPI - An introduction to parallel Python using MPI*, `http://www.llnl.gov/computing/develop/python/pyMPI.pdf`.

[9] Migallón, H., Migallón, V., Penadés, J., *PySParNLCG: Interfaz-librería del Método del Gradiente Conjugado no Lineal para Sistemas Dispersos*, Actas de las XX Jornadas de Paralelismo, 123–128, A Coruña, 2009.

[10] Migallón, H., Migallón, V., Penadés, J., *Parallel Nonlinear Conjugate Gradient Algorithms on Multicore Architectures*, Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE), 689–700, Gijón, 2009.

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

[12] Peterson, P., *F2PY: Fortran to Python interface generator*, `http://cens.ioc.ee/projects/f2py2e`.

[13] van Rossum, G., Drake Jr., F.L., *An Introduction to Python*, Network Theory Ltd, 2003.