Nuevas funcionalidades de PyPANCG usando OpenMP y CUDA

H. Migallón, [1] V. Migallón[2] y J. Penadés[2]

Resumen— **En este trabajo presentamos nuevas funcionalidades de PyPANCG. PyPANCG es una librería paralela tratada como una interfaz de alto nivel para resolver sistemas no lineales. Mediante el uso de las nuevas funcionalidades, PyPANCG puede explotar el paralelismo ofrecido por arquitecturas de memoria compartida y por GPUs. La librería sigue disponiendo de dos módulos PySParNLCG y PySParNLPCG, los cuales incluyen las nuevas funcionalidades manteniendo la compatibilidad con versiones anteriores. El módulo PySParNLCG resuelve en paralelo sistemas no lineales mediante el método del gradiente conjugado no lineal, mientras que el módulo PySParNLPCG incluye la técnica de precondicionamiento basada en el método de dos etapas por bloques. El lenguaje de trabajo escogido ha sido Python y los experimentos realizados muestran el buen comportamiento tanto en arquitecturas de memoria compartida como en GPUs.**

Palabras clave— **GPGPU, librerías GPU, gradiente conjugado no lineal, precondicionadores paralelos, factorizaciones ILU, métodos por bloques en dos etapas, Python.**

I. Introducción

En este trabajo presentamos nuevas funcionalidades de PyPANCG (`http://atc.umh.es/PyPANCG`), una interfaz librería paralela para la resolución de sistemas suavemente no lineales de la forma

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

donde $A \in \Re^{n\times n}$ y $\Phi : \Re^n \to \Re^n$ es una función diagonal.

Esta librería, que se distribuye como un paquete estándar de Python, incluye tanto el método del gradiente conjugado no lineal (NLCG) como el método del gradiente conjugado precondicionado no lineal (NLPCG). En las versiones anteriores de PyPANCG se disponía de diferentes herramientas para gestionar el entorno paralelo a través de *MPI* (`http://www-unix.mcs.anl.gov/mpi`). En esta nueva versión se incluyen, por un lado, las herramientas necesarias para manejar arquitecturas de memoria compartida, haciendo uso de OpenMP, y por otro, mediante CUDA se puede trabajar con GPUs.

El presente trabajo está estructurado de la siguiente forma: en la sección II se introducen ambos métodos, tanto el método del gradiente conjugado no lineal (NLCG), como su versión precondicionada (NLPCG), pertenecientes a los módulos PySParNLCG y PySParNLPCG respectivamente. En la secciones III, IV y V se explicarán las herramientas básicas de desarrollo, los parámetros fundamentales de la librería y el modo de implementar la no linealidad del problema, respectivamente. En la sección VI se muestran algunos de ejemplos de utilización de las nuevas funcionalidades de PyPANCG, mientras que en la sección VII se ilustrará, mediante experimentos numéricos, el rendimiento de la librería PyPANCG, aportando algunas conclusiones en la sección VIII.

[1]Dpto. de Física y Arquitectura de Computadores, Universidad Miguel Hernández, e-mail: `hmigallon@umh.es`.

[2]Dpto. de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, e-mail: `violeta,jpenades@dccia.ua.es`.

II. Métodos no lineales

Sea el problema de resolver el sistema no lineal (1), donde $A \in \Re^{n\times n}$ es una matriz simétrica y definida positiva y $\Phi : \Re^n \to \Re^n$ es una función no lineal con algunas propiedades locales no significativas. Donde $\langle x, y\rangle = x^T y$ denota el producto interno en $\Re^n$.

El problema de minimización de encontrar $x \in \Re^n$ tal que

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

donde $J(x) = \frac{1}{2}\langle Ax, x\rangle - \Psi(x)$, es equivalente a encontrar $x \in \Re^n$ tal que $F(x) = Ax - \Phi(x) = 0$, donde $\Phi(x) = \Psi'(x)$. Una buena opción para resolver el sistema no lineal (1), considerando la conexión con el problema de minimización (2), es la versión de Fletcher-Reeves [5] del método del gradiente conjugado no lineal (NLCG), el cual puede describirse de la siguiente forma:

Algoritmo 1: (Versión de Fletcher-Reeves del método NLCG)

Dado un vector inicial $x^{(0)}$

$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$

$p^{(0)} = r^{(0)}$

Para $i = 0, 1, \ldots,$ hasta convergencia

$\alpha_i =\to$ ver más adelante

$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$

$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$

Test de convergencia

$\beta_{i+1} = -\frac{\langle r^{(i+1)}, r^{(i+1)}\rangle}{\langle r^{(i)}, r^{(i)}\rangle}$

$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

Hay que tener en cuenta que, en el algoritmo 1, α_i debe escogerse para minimizar la función asociada J en la dirección de $p^{(i)}$. Esto es equivalente a resolver el problema unidimensional $\frac{dJ(x^{(i)}+\alpha_i p^{(i)})}{d\alpha_i} = 0$. De la definición de J se puede deducir que

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

$$-\Psi(x^{(i)} + \alpha_i p^{(i)}).$$

Donde una simple diferenciación con respecto a α_i obtiene

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

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

donde $r^{(i)} = \Phi(x^{(i)}) - Ax^{(i)}$ es el residuo no lineal.

Por otro lado, es fácil observar que la segunda derivada respecto a α_i es

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

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

Utilizando el método de Newton para resolver el problema para α_i, se obtiene $\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$, donde

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

$$\frac{\alpha_i^{(k)}pi1 - pi2 + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)}+\alpha_i^{(k)}p^{(i)}), p^{(i)} \right\rangle}{pi1 - \left\langle \Phi'(x^{(i)}+\alpha_i^{(k)}p^{(i)})p^{(i)}, p^{(i)} \right\rangle}.$$

En estas expresiones puede observarse que para el cálculo de $\delta^{(k)}$, los productos internos $pi1 = \langle Ap^{(i)}, p^{(i)} \rangle$ y $pi2 = \langle r^{(i)}, p^{(i)} \rangle$ solo se calculan una única vez en la iteración inicial del método de Newton. Además el término $Ap^{(i)}$ es calculado en la iteración del cálculo del gradiente conjugado.

Se han desarrollado algoritmos eficientes para resolver el sistema (1), diseñando los algoritmos paralelos tanto del algoritmo 1 como del gradiente conjugado precondicionado, basado este último tanto en el algoritmo 1, como en un precondicionador polinomial basado en el método en dos etapas [3] (una descripción detallada puede verse en [4] y [7]).

El precondicionamiento es una técnica para mejorar el número de condición (cond) de una matriz. Si suponemos que M es una matriz simétrica y definida positiva fácilmente invertible que aproxima a A, podemos resolver el sistema $Ax = \Phi(x)$ indirectamente resolviendo el sistema $M^{-1}Ax = M^{-1}\Phi(x)$. De modo que si $\text{cond}(M^{-1}A) << \text{cond}(A)$ puede resolverse iterativamente $M^{-1}Ax = M^{-1}\Phi(x)$ mucho más rápido que el sistema original. Obteniéndose por tanto el siguiente método del gradiente conjugado precondicionado no lineal (NLPCG).

Algoritmo 2: (Gradiente conjugado no lineal precondicionado)
Dado un vector inicial $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
Resolver $Ms^{(0)} = r^{(0)}$
$p^{(0)} = s^{(0)}$
Para $i = 0, 1, \ldots,$ hasta convergencia
$\alpha_i =\rightarrow$ ver algoritmo 1
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i Ap^{(i)}$
Resolver $Ms^{(i+1)} = r^{(i+1)}$
Test de convergencia
$\beta_{i+1} = -\frac{\langle s^{(i+1)}, r^{(i+1)} \rangle}{\langle s^{(i)}, r^{(i)} \rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1}p^{(i)}$

Dado que el sistema auxiliar $Ms = r$ debe resolverse en cada iteración del gradiente conjugado, debe poder resolverse fácilmente. Además, para obtener un precondicionador eficaz es necesario que M sea una buena aproximación de A. Una de las técnicas generales de precondicionamiento es el uso de series truncadas (ver [1]). Para obtener estos precondicionadores se considera una partición de A tal que

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

y se realizan m pasos del método iterativo definido por esta partición en busca de la solución de $As = r$, siendo $s^{(0)} = 0$. Es bien conocido que la solución del sistema auxiliar $Ms = r$ es $s = (I + R + R^2 + \ldots + R^{m-1})P^{-1}r$, donde $R = P^{-1}Q$ y la matriz precondicionadora es $M_m = P(I + R + R^2 + \ldots + R^{m-1})^{-1}$ (ver por ejemplo [1]). Para obtener los precondicionadores siendo $s^{(0)} = 0$, se realizan m pasos del método en dos etapas por bloques de Jacobi hacia la solución de $As = r$. Para obtener las particiones internas de estos bloques hacemos uso de factorizaciones incompletas LU (ver por ejemplo [4]).

III. Herramientas de desarrollo

En esta sección se analizan las diferentes herramientas básicas en el proceso de desarrollo de la librería PyPANCG. El lenguaje de programación utilizado para el desarrollo es Fortran, de modo que las rutinas básicas de la librería para arquitecturas de memoria compartida han sido desarrolladas en este lenguaje. El lenguaje C ha sido utilizado para el desarrollo en CUDA de las rutinas básicas. El objetivo es aunar la potencia de desarrollo de un lenguaje como Python con la potencia de ejecución que ofrecen tanto el lenguaje Fortran como CUDA.

Para acceder a las rutinas desarrolladas en Fortran desde Python se ha usado la herramienta F2PY (`http://cens.ioc.ee/projects/f2py2e`). Estas rutinas han sido desarrolladas haciendo uso de las extensiones OpenMP para el manejo del entorno paralelo de memoria compartida. Una característica añadida en esta versión es la menor influencia de la herramienta utilizada para el manejo de arrays o vectores y la comunicación de éstos entre Python y Fortran. No obstante, siguen estando disponibles las dos opciones para el manejo de vectores, tanto *Numeric* como *numarray* (*numarray* es parte de *NumPy*). Las nuevas funcionalidades no permiten el cómputo de ninguna rutina principal en Python, sin embargo la comunicación de vectores entre lenguajes sigue siendo un aspecto importante para mejorar el rendimiento.

Por otra parte, para acceder desde Python a las rutinas de CUDA desarrolladas en C, hemos hecho uso de PyCUDA. PyCUDA es un paquete que ofrece acceso al API de CUDA desde Python, de tal modo que no es necesario desarrollar una librería que incluya las rutinas CUDA, a la que se acceda desde Python. El paquete PyCUDA hace uso de CodePy,

este paquete compila código C dinámicamente y lo carga en el intérprete Python, éste es un aspecto muy importante para el desarrollo posterior de la no linealidad de los sistemas a resolver.

IV. Parámetros de PyPANCG

En esta sección se verán los parámetros que han de pasarse a las funciones Python para resolver un sistema no lineal mediante los métodos incluidos, NLCG y NLPCG. Los únicos parámetros indispensables son aquellos que determinan el sistema ($Ax = \phi(x)$) a resolver, es decir la matriz A almacenada en formato CSR (Compressed Sparse Row), la función no lineal $\phi(x)$ y el tamaño del sistema. Además se necesita la derivada de la función $\phi(x)$ ($\phi'(x)$) para el cálculo de δ, tal y como se vio en la sección II. También hay un conjunto de parámetros opcionales que permiten modificar los métodos implementados NLCG y NLPCG. En el caso de que los parámetros opcionales no sean especificados toman un valor dado por defecto. Los parámetros opcionales son (ver [7] para más información):

- Vector inicial del proceso iterativo (*initial_vector*).
- Criterio de parada global (*global_stopping_error*).
- Criterio de parada en el cálculo de α (*alfa_stopping_error*).
- Número máximo de iteraciones en el cálculo de α (*iter_alfa*).
- Mecanismo para comunicar variables enteras entre Python y Fortran o C (*trash_int*).
- Mecanismo para comunicar variables en doble precisión entre Python y Fortran o C (*trash_double*).
- Nivel de la factorización incompleta LU del método NLPCG (*level*).
- Número de iteraciones externas del método NLPCG (*niter_2e*).
- Número de iteraciones internas del método NLPCG (*val_q*).

Otro parámetro importante, que la librería puede calcular, es el tamaño de problema asignado a cada proceso, dado por el parámetro *block_dimensions*. Este parámetro es un array de enteros cuya dimensión coincide con el número de procesos y que almacena el tamaño de bloque asignado a cada proceso. En los ejemplos incluidos en el paquete de PyPANCG este parámetro es calculado internamente con el fin de balancear la carga correctamente. Por otro lado este parámetro no es utilizado si se hace uso de CUDA, ya que en este caso no se hace uso del sistema multicore aunque estuviera disponible. En este caso un core maneja la computación en la GPU.

El parámetro *For_or_Py* selecciona el conjunto de rutinas que van a utilizarse y por tanto también selecciona la plataforma paralela a utilizar. En [7] pueden verse las opciones para el uso de una plataforma de memoria distribuida. Las siguientes opciones son las adecuadas para el uso bien de una plataforma de memoria compartida o bien de una GPU:

1. *Fortran_mp*: Las rutinas están codificadas en Fortran usando OpenMP. Además ϕ y ϕ' están codificadas independientemente.
2. *Fortran_mp_full*: Todas las rutinas están codificadas en Fortran, pero ϕ y ϕ' no están codificadas independientemente.
3. *GPU*: Todas las rutinas están codificadas en C implementadas como kernels de CUDA. Además ϕ y ϕ' están codificadas como kernels de CUDA independientes.

El uso de OpenMP o CUDA implica que las funciones no lineales deben ser codificadas en Fortran o C respectivamente. El uso de *Fortran_mp* o *Fortran_mp_full* obliga a codificar las funciones no lineales en Fortran y tras un proceso de compilación linkarlas desde Python. Sin embargo, el uso de *GPU* evita el proceso explícito de recompilación de las funciones desarrolladas como kernels de CUDA, explotando las características que ofrece CodePy.

Por último, existen nuevos parámetros para trabajar con las nuevas opciones del parámetro *For_or_Py*, es decir para trabajar con OpenMP o CUDA. El primer parámetro es el número de procesos de la arquitectura de memoria compartida, en el caso de utilizar OpenMP. El resto de parámetros nuevos son usados por CUDA. En una llamada a un kernel de CUDA, además de los clásicos parámetros de entrada de una función, existen dos parámetros que definen la estructura de hilos que se generará para el cómputo del kernel. Estos parámetros son el número de bloques que serán generados (*grid*), y el número de hilos en cada bloque (*block*), para ver su significado específico puede consultarse por ejemplo [8]. Además existen dos variables globales para la optimización del cálculo de los productos internos, *VECTOR_N* y *ELEMENT_N*, ver [4] para una descripción detallada de dichos parámetros. Hay que remarcar, que tanto en el método NLCG como el método NLPCG el producto interno es una operación importante, siendo de relevancia especial en CUDA por conllevar un proceso de reducción.

V. Codificación de las funciones no lineales

En una librería para resolver sistemas no lineales es muy importante el modo de implementar las funciones no lineales del problema a resolver. En [7] puede verse que PyPANCG puede trabajar a nivel de vector o a nivel de componente. Sin embargo, si se hace uso de OpenMP o de CUDA es necesario trabajar a nivel de componente y no de vector. El siguiente ejemplo muestra el código Fortran para el cálculo de la función $\phi(x)$ del problema resuelto en los ejemplos aportados por PyPANCG.

```
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
```

El código del kernel de CUDA para calcular la misma función es:

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

Se puede observar que ambas funciones requieren que les sea comunicado un parámetro (sc) para el cálculo de la función ϕ. Si se usa Fortran y OpenMP, para comunicar parámetros tanto si son reales como si son enteros, se usan dos vectores, uno para enteros *trash_int* y otro para valores reales de doble precisión *trash_double*. Estos vectores son dinámicos y todos aquellos parámetros necesarios para el cómputo de las funciones ϕ y ϕ' pueden ser comunicados a través de ellos. Lógicamente las funciones ϕ y ϕ' siempre han de ser implementadas para adaptarse al problema a resolver. Por otra parte, si se hace uso de CUDA, la reserva de memoria y las comunicaciones GPU-CPU pueden ser costosas, por lo tanto la memoria utilizada debe ser la estrictamente necesaria. Por ello en el ejemplo anterior no pasamos un vector sino únicamente el elemento necesario.

VI. Ejemplos con Python usando OpenMP y CUDA

El uso de las nuevas funcionalidades de los módulos PySparNLCG y PySParNLPCG de PyPANCG que hacen uso de OpenMP y CUDA es muy similar a las versiones anteriores presentadas en [7]. Para usar la librería al menos debe pasarse el tamaño del sistema (nrow), la matriz A en formato CSR (tcol, trow, tval), las funciones no lineales (ϕ y ϕ') y opcionalmente el tamaño de bloque asignado a cada proceso (block_dimensions). Además, si son necesarios otros parámetros se hará a través de *trash_int* y *trash_double*. El siguiente ejemplo muestra el código más simple para implementar una llamada a PYPANCG usando OpenMP y el método NLCG.

```
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)

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)
```

El número de procesos se fija en la línea 5. El número de procesos ha de fijarse con anterioridad a definir la estructura de bloques, que se define en la línea 9. La matriz A se obtiene en la línea 10, en función de la estructura de bloques definida anteriormente. Hay que remarcar que la generación de la matriz A realizada en la línea 10 es realizada por completo por el proceso root, mantenemos el nombre de la rutina como *PartialMatrixA* para mantener la compatibilidad con versiones anteriores. En la línea 11 se hace la llamada al método NLCG, dando por supuesto que las funciones *Fi_x* (ϕ) y *Fi_prime_x* (ϕ') han sido desarrolladas en Fortran. En el desarrollo o implementación de *Fi_x* (ϕ) y *Fi_prime_x* (ϕ') no se hace uso de OpenMP dado que se implementan a nivel de componente y no de vector.

El ejemplo más sencillo para la utilización del método NLPCG con OpenMP es muy similar al ejemplo anterior que llamaba al método NLCG. En este caso, el módulo a importar debe ser el módulo PySParNLPCG (línea 4), y en la línea 11 debe llamarse a la función principal del módulo PySParNLPCG.

```
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)
```

En el caso de usar CUDA el ejemplo más sencillo es el siguiente.

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

En la línea 5 se importa el módulo *PyPANCG.PySparNLCG_ModGPU*. Este módulo contiene todos los kernels de CUDA que son necesarios para el cómputo del método NLCG, incluyendo las funciones *Fi_x* (ϕ) y *Fi_prime_x* (ϕ'). Es importante remarcar que la codificación de las funciones no lineales se realiza sin la necesidad de ningún proceso de compilación. Si se usa CUDA y el método NLCG la CPU únicamente realiza tareas de gestión de la GPU, por lo tanto un único proceso es necesario (ver línea 6). La matriz se calcula en la línea 10 por la CPU. En la línea 11 se realiza la llamada al método NLCG para ser computado en la GPU. Como se puede observar las funciones *Fi_x* (ϕ) y *Fi_prime_x* (ϕ') no son definidas en esta llamada, esto se debe a que, como se ha comentado, estas funciones deben estar incluidas en el módulo que contiene los kernels de CUDA, es decir el módulo *PyPANCG.PySparNLCG_ModGPU*.

Por último, la llamada más simple al método NLPCG usando CUDA puede verse a continuación.

Esta llamada es muy similar a la anterior importando el módulo PySParNLPCG en lugar del módulo PySParNLCG, en la línea 4. E importando el módulo *PyPANCG.PySparNLPCG_ModGPU* que contiene los kernels de CUDA del método NLPCG, en la línea 5.

```
4  import PyPANCG.PySParNLPCG as PySparNLPCG
5  import PyPANCG.PySparNLPCG_ModGPU as PNLPCG_ModGPU
```

La llamada a la función principal del módulo NLPCG es la siguiente:

```
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)
```

En [4] se comentan algunos aspectos importantes para mejorar el rendimiento de estos métodos (NLCG y NLPCG) en CUDA. Esencialmente, estas mejoras se obtienen por la mejora en el cálculo de los productos internos y en el número de hilos por bloque en la llamada a los kernels de CUDA. El siguiente ejemplo muestra una mejora en el método NLPCG a través del uso de los parámetros grid y block, descritos en [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)
```

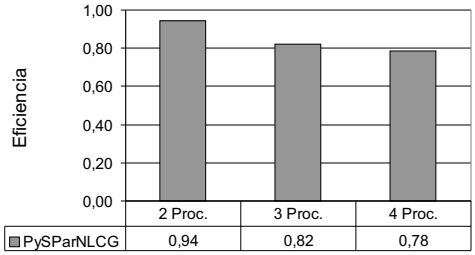
VII. Experimentos numéricos

Para analizar el comportamiento de la librería PyPANCG, en particular los algoritmos correspondientes a las nuevas funcionalidades, hemos utilizado un multiprocesador Intel Core 2 Quad Q6600, 2.4 GHz, denominado SULLI. La GPU disponible en SULLI es una GeForce GTX 280. Los análisis realizados comparan los tiempos de cómputo tanto en la GTX 280 como en el multiprocesador SULLI cuando se usa código Fortran puro (usando OpenMP) o código C puro (usando CUDA), respecto a los tiempos obtenidos por la librería PyPANCG.

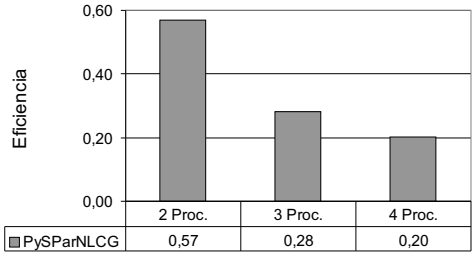
El ejemplo a resolver es una ecuación elíptica en derivadas parciales, denominado problema de Bratu. En este problema la generación de calor por combustión debe equilibrarse con el calor transmitido por conducción. El modelo tridimensional viene dado por

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

donde u es la temperatura y λ es una constante conocida como el parámetro de Frank-Kamenetski (ver por ejemplo [2]). Este problema tiene dos posibles soluciones para un valor de λ dado. Una de las soluciones está cercana a $u = 0$ y es fácil de obtener. La otra solución se obtiene partiendo de un punto cercano a dicha solución para que el sistema converja. En nuestro modelo consideramos un dominio cúbico 3D Ω de longitud unidad y $\lambda = 6$. Para resolver la ecuación (4) mediante el método de diferencias finitas se considera un grid de Ω con d^3 nodos. Esta discretización da lugar a un sistema no lineal de la forma $Ax = \Phi(x)$, donde $\Phi : \Re^n \to \Re^n$ es una función diagonal no lineal, es decir la componente i-ésima de Φ_i de Φ es función únicamente de la i-ésima componente de x. La matriz A es una matriz dispersa de orden $n = d^3$ con un número típico de elementos no nulos por fila igual a siete, y con menor número de elementos no nulos en los puntos que corresponden a la frontera del dominio físico.



(a) PySParNLCG

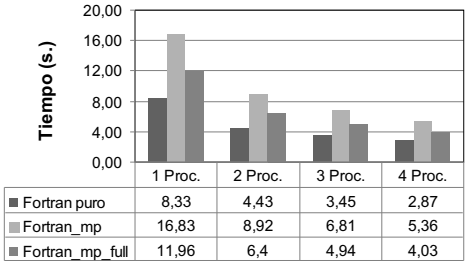


(b) PySParNLPCG

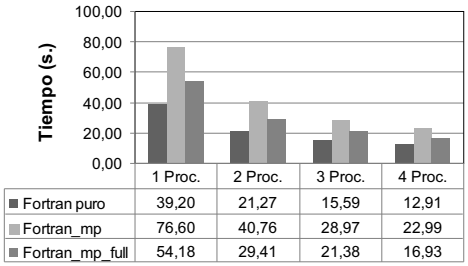
Fig. 1. Eficiencia OpenMP, $n = 373248$.

En primer lugar analizamos la eficiencia de ambos métodos haciendo uso de OpenMP. Hay que recordar que el método NLCG está incluido en el módulo PySParNLCG, mientras que el método NLPCG está incluido en el módulo PySParNLPCG. En los experimentos realizados se han fijado los parámetros a su valor óptimo. Los parámetros fijados han sido: el nivel de llenado (*level*) de la factorización incompleta LU, el número de iteraciones internas (*val_q*) y externas (*niter_2e*) del método iterativo en dos etapas por bloques. En todos los experimentos los valores asignados a los criterios de parada de los procesos iterativos ha sido 10^-7, tanto para *global_stopping_error* como para *alfa_stopping_error*, el valor de *iter_alfa* ha sido fijado a 2. La figura 1 muestra la eficiencia de ambos métodos usando OpenMP con hasta 4 procesos. Puede observarse que el comportamiento de la eficiencia no se ve modificado por el uso de PyPANCG, de hecho las eficiencias son similares a las obtenidas con código Fortran puro. En el caso del método NLCG se obtiene buenas eficiencias que decrecen ligeramente al aumentar el número de procesos. Sin embargo, como puede verse en [6], el método NLPCG es un muy buen algoritmo pero con una escalabilidad limitada.

Puede seleccionarse la opción OpenMP mediante dos valores distintos del parámetro *For_or_Py* (ver sección IV), en la figura 2 puede observarse el comportamiento de ambas opciones. En el caso de uti-

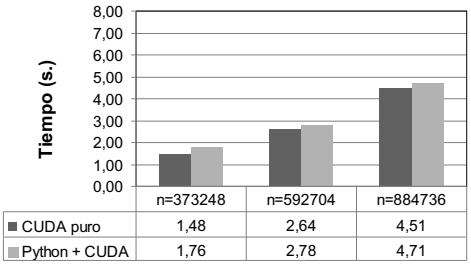


(a) 125000

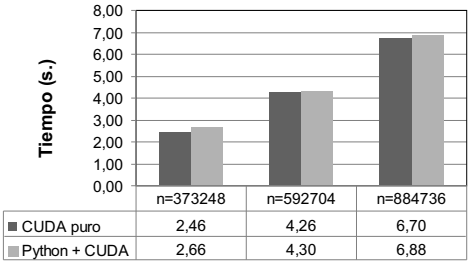


(b) 373248

Fig. 2. PySparNLCG usando OpenMP.



(a) PySParNLCG



(b) PySParNLPCG

Fig. 3. PyPANCG usando CUDA.

lizar la opción *Fortran_mp_full* se obtienen resultados más similares a los obtenidos con Fortran puro que usando la opción *Fortran_mp*. Hay que remarcar que el proceso de desarrollo es más sencillo haciendo uso de *Fortran_mp_full* que si se hace uso de la opción *Fortran_mp*.

Por último comparamos los resultados obtenidos por ambos módulos si seleccionamos trabajar con CUDA. En particular, la figura 3 muestra los resultados obtenidos con PyPANCG comparados con los tiempos computacionales obtenidos usando código C puro, variando el tamaño del problema a resolver. Tal y como puede observarse los tiempos de cómputo en ambos casos son muy similares.

VIII. Conclusiones

En este trabajo hemos presentado nuevas funcionalidades de la librería PyPANCG, la cual es una librería interfaz que implementa tanto el método del gradiente conjugado como el método precondicionado del gradiente conjugado para sistemas no lineales. El objetivo de esta librería es ofrecer, mediante un lenguaje de alto nivel como Python, el desarrollo de aplicaciones científicas escondiendo todo lo posible la complejidad de la programación a bajo (o medio) nivel. Las nuevas funcionalidades has sido desarrolladas para trabajar en arquitecturas de memoria compartida, mediante OpenMP, y en GPUs. Se ha descrito la librería mostrando sus ventajas para obtener tiempos de desarrollo cortos. La librería ha sido diseñada para adaptarse a las diferentes etapas del proceso de diseño de nuevas aplicaciones, para el caso de arquitecturas de memoria compartida. En el caso de OpenMP debe trabajarse con opciones diferentes si el objetivo es bien desarrollar rápidamente o bien el rendimiento computacional. En el caso de usar GPU ambos objetivos pueden conseguirse simultáneamente, que además es la plataforma en la que se obtienen los mejores prestaciones.

Acknowledgements

El presente trabajo ha sido financiado por el Ministerio de Educación y Ciencia mediante el proyecto TIN2011-26254.

Referencias

[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`