Mecánica Computacional Vol. XXIII, pp. 3153-3163
G.Buscaglia, E.Dari, O.Zamonsky (Eds.)
Bariloche, Argentina, November 2004

DESARROLLO DE APLICACIONES PARALELAS EN PYTHON

Lisandro Dalcín, Mario Storti y Rodrigo Paz

Centro Internacional de Métodos Computacionales en Ingeniería
CONICET - INTEC - U.N.L.
Parque Tecnológico del Litoral Centro
(3000) Santa Fe, Argentina
email: dalcinl@intec.unl.edu.ar
home: http://www.cimec.org.ar

Palabras clave: Python, MPI, PETSc, ParMETIS, cálculo paralelo.

Resumen. *Python es un lenguaje de programación interpretrado, interactivo y orientado a objetos. Combina el soporte de módulos, clases, excepciones y tipos de datos de muy alto nivel con una sintaxis muy clara. Su librería estándar provee acceso al sistema operativo, librerías gráficas e Internet. Su implementación es portable (UNIX, Mac, Windows) y totalmente libre para uso, modificación y redistribución.*

En este trabajo se describen nuestras experiencias en CIMEC utilizando Python en la implementación de aplicaciones paralelas sobre clusters de PC's. Se comentan algunas de las herramientas disponibles para el cálculo científico. También se presentan interfaces desarrolladas a algunas librerías paralelas populares, tales como MPI, PETSc y ParMETIS, junto a algunos ejemplos básicos de su utilización.

1. INTRODUCCIÓN

Las aplicaciones para cálculo científico orientadas a la simulación de problemas multifísica deben presentar a los usuarios del código una interfaz capaz de proveer un núcleo general de funcionalidades básicas, pero con la flexibilidad suficiente para incorporar las necesidades particulares de cada modelo.

Con el transcurso del tiempo, el uso de aplicaciones de este tipo evoluciona en la generación de archivos de configuración y entrada de datos más o menos complicados, con sentencias que se adicionan a medida que surgen nuevos modelos a simular con nuevos parámetros a ingresar. En definitiva, la aplicación debe interpretar una especie de *"script ad-hoc"*, lo que en general origina dificultades para los usuarios (que prácticamente deben aprender un nuevo lenguaje) como para los encargados del mantenimiento del código (que deben mantener la consistencia y la documentación).

Una solución alternativa es la utilización de algún lenguaje de extensión bien establecido, portable y con abundante soporte, orientado la programación funcional y orientada a objetos. De esta manera se provee a los usuarios una interfaz totalmente programable y fácil de extender.

1.1. Características de Python

- Python[1] es un lenguaje de programación moderno, orientado a objetos, interpretado e interactivo. Su sintaxis es muy clara y de rápido aprendizaje . Su repertorio incluye programación funcional, clases y manejo de errores mediante excepciones.
- Posee un número reducido de tipos de datos de muy alto nivel, tales como listas y diccionarios, y un conjunto completo de operaciones con *strings*, incluyendo expresiones regulares. Soporta la programación con *threads*.
- Python está implementado en ISO C, lo que lo hace sumamente portable. Está disponible para varias variantes de UNIX, Mac y Windows. El código fuente es totalmente abierto; puede distribuirse y/o modificarse libremente incluso en aplicaciones comerciales.
- Existen interfaces a servicios del sistema y varias librerías populares, incluyendo las utilizadas en desarrollo de interfaces gráficas de usuario y aplicaciones de visualización (X11, Motif, Tk, Mac, MFC, GTK, VTK, Qt).
- Puede utilizarse como lenguaje de extensión en aplicaciones que requieran una interface programable.
- Es fácilmente extensible mediante nuevos módulos escritos en C/C++ que se incorporan inmediatamente al lenguaje mediante importación dinámica. Este mecanismo permite la utilización de cualquier librería escrita en Fortran, C o C++dentro de Python.

1.2. Algunos usuarios de Python en la comunidad científica

- En el *Space Telescope Science Institute* se desarrolla el módulo `numarray` para Python, el cual se utilizan para el procesamiento de imágenes del telescopio espacial Hubble.

- *AlphaGene*, dedicada al descubrimiento de genes y proteínas, utiliza Python como núcleo de su sistema bioinformático. Este sistema integra diferentes formados de datos de entrada, bases de datos, análisis genéticos de larga escala, supercomputadoras especializadas e interfaces basadas en HTML.

- SPaSM[2] es un código paralelo de dinámica molecular desarrollado por la división de física teórica del *LANL*. Es utilizable en cualquier plataforma que soporte MPI. Esta formado por un núcleo de funcionalidades escritas en C, que son llamadas desde Python para conducir las simulaciones en forma flexible y proveer facilidades en el postprocesamiento de los enormes volúmenes de datos generados.

1.3. Algunas herramientas disponibles

- **Numeric y Numarray:**[3] son módulos de extensión (el primero discontinuado; el segundo una reimplementación del anterior) que proveen manipulación de arreglos numéricos y capacidades computacionales similares a las encontradas en IDL, Matlab/Octave o Fortran 90. Utilizando estos módulos se pueden escribir aplicaciones eficientes para el procesamiento de datos directamente en Python, sin necesidad de utilizar código en C, C++ o Fortran.

- **Pyfort:**[4] permite conectar rutinas escritas en Fortran con Python y el módulo `Numeric`. Es capaz de traducir rutinas de Fortran a un módulo de extensión en C que puede ser llamado desde Python.

- **SciPy:**[5] es una librería *open source* de herramientas científicas para Python que suplementa a `Numeric` juntando otros módulos de ciencia e ingeniería en un único paquete. Incluye módulos para gráficos, optimización, integración, procesamiento de señales e imágenes, algoritmos genéticos, *ODE solvers* y otros.

- **SWIG:**[6] es una herramienta de desarrollo que permite conectar código escrito en C y C++ con una variedad de lenguajes de programación de alto nivel. Es utilizado fundamentalmente con lenguajes de *scripting* tales como Perl, Tcl/Tk, y Python.

2. PARALELIZACIÓN BÁSICA DEL INTÉRPRETE DE PYTHON

La generación de una versión paralelizada básica del intérprete de Python es una tarea sencilla (ver figura 1). Es suficiente con proveer la inicialización de MPI (con `MPI_Init()`), la llamada al intérprete (con la función `Py_Main()` de la librería de Python), y la finalización de MPI (con `MPI_Finalize()`).

Esta estrategia solamente permite la ejecución en paralelo de *scripts*, no permite la utilización del intérprete en modo interactivo.

Para utilizar el intérprete en forma interactiva y en paralelo, debe modificarse el mecanismo por el cual se obtiene la entrada del usuario. Se debe leer la entrada en el proceso maestro para luego ser distribuida (*broadcast*) a los demás procesos.

```
#include <Python.h>
#include <mpi.h>

int main(int argc, char **argv) {
  int status;
  /* initialize MPI */
  MPI_Init(&argc, &argv);
  /* call Python main */
  status = Py_Main(argc, argv);
  /* finalize MPI */
  MPI_Finalize();
  return status;
}
```

Figura 1: Paralelización básica del intérprete de Python

3. MÓDULO MPI

3.1. Algunos comentarios sobre MPI

MPI[7–9] es un sistema estandarizado y portable de paso de mensajes, diseñado para funcionar en una amplia variedad de computadoras paralelas. El estándar define la sintaxis y semántica de un conjunto de funciones de librería que permiten a los usuarios escribir programas portables en los principales lenguajes utilizados por la comunidad científica (Fortran, C, C++).

Desde su aparición, la especificación MPI se ha transformado en el estándar dominante en librerías de paso de mensajes para computadoras paralelas. En la actualidad se dispone de diversas implementaciones, algunas provistas por los fabricantes de computadoras de alta performance, hasta otras de reconocidos proyectos *open source*, tales como MPICH[10] y LAM/MPI,[11] muy utilizadas en los *clusters* de PC's tipo *Beowulf*.[12]

3.2. Diseño

Este módulo provee una aproximación orientada a objetos para paso de mensajes. Está basado en la sintaxis de la especificación MPI-2 para C++. Por lo tanto, cualquier usuario que conozca la sintaxis estándar de MPI para C++ puede utilizar este módulo sin necesidad de conocimientos adicionales.

El diseño es simple y efectivo. El módulo MPI consiste de código escrito en Python que define constantes, funciones y una jerarquía de clases. Este código llama a un módulo de soporte escrito en C, el cual provee acceso a las constantes y funciones de la especificación MPI-1.

Los objetos a comunicar se serializan utilizando el módulo estándar `cPickle` de Python. Luego, la representación serializada del objeto (en realidad, una cadena de caracteres) es transmitida apropiadamente (utilizando el tipo *MPI_CHAR*). Finalmente, el objeto original se recupera a partir del mensaje recibido.

Si bien la serialización de objetos con `cPickle` impone algunos costos adicionales en tiempo y memoria, la estrategia es completamente general, y permite la comunicación los diversos tipos de objetos de Python en forma totalmente trasparente para el usuario.

3.3. Ejemplo de uso

A modo de ejemplo, en la figura 2 se muestra el uso intérprete paralelizado de Python en una sesión interactiva con 3 procesos en la que se efectúan diversas operaciones colectivas con diversos tipos de objetos.

```
$ lamboot nodes.dat
$ mpirun -np 3 ppython
>>> import mpi
```

(a) *Startup* (LAM/MPI)

```
$ mpirun -machinefile nodes.dat -np 3 ppython
>>> import mpi
```

(b) *Startup* (MPICH)

```
>>> if mpi.rank == 0:
...   sendbuf = { 'op1': True, \
...               'op2': 2.52, \
...               'op3': 'yes' }
... else:
...   sendbuf = None
...
>>> print '[%d]' % mpi.rank, sendbuf
[0] {'op1': True, 'op3': 'yes', 'op2': 2.52}
[2] None
[1] None
>>>
>>> recvbuf = mpi.WORLD[0].Bcast(sendbuf)
>>> print '[%d]' % mpi.rank, recvbuf
[0] {'op1': True, 'op3': 'yes', 'op2': 2.52}
[2] {'op1': True, 'op3': 'yes', 'op2': 2.52}
[1] {'op1': True, 'op3': 'yes', 'op2': 2.52}
```

(c) MPI_BCAST

```
>>> root = mpi.size/2
>>>
>>> sendbuf = None
>>> if mpi.rank = root:
...     sendbuf = [ (i,i**2,i**3) \
...                 for i in [2,3,4] ]
>>> print '[%d] %s' % (mpi.rank, sendbuf)
[0] None
[1] [(2, 4, 8), (3, 9, 27), (4, 16, 64)]
[2] None
>>>
>>> recvbuf = mpi.WORLD[root].Scatter(sendbuf)
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] (2, 4, 8)
[1] (3, 9, 27)
[2] (4, 16, 64)
```

(d) MPI_SCATTER

```
>>> sendbuf = [mpi.rank**2 , mpi.rank%2!=0]
>>> print '[%d] %s' % (mpi.rank, sendbuf)
[0] [0, False]
[1] [1, True]
[2] [4, False]
>>>
>>> root = mpi.size/2
>>> recvbuf = mpi.WORLD[root].Gather(sendbuf)
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] None
[1] [[0, False], [1, True], [4, False]]
[2] None
```

(e) MPI_GATHER

```
>>> sendbuf = []
>>> for i in xrange(mpi.size):
...    sendbuf += [(mpi.size+mpi.rank)*100]
...
>>> print "[%d] %s" % (mpi.rank, sendbuf)
[0] [300, 300, 300]
[1] [400, 400, 400]
[2] [500, 500, 500]
>>>
>>> recvbuf = mpi.WORLD.Alltoall(sendbuf)
>>> print "[%d] %s" % (mpi.rank, recvbuf)
[0] [300, 400, 500]
[1] [300, 400, 500]
[2] [300, 400, 500]
```

(f) MPI_ALLTOALL

Figura 2: MPI en Python

4. MÓDULO PETSC

4.1. Algunos comentarios sobre PETSc

PETSc[13] es desarrollado en la división de matemática y ciencias de la computación en *ANL*, y utilizado por decenas de paquetes y aplicaciones en variadas áreas.

Esta librería provee un conjunto de estructuras de datos y rutinas para la solución escalable (paralela) de aplicaciones científicas modeladas por ecuaciones diferenciales en derivadas parciales. Emplea el estándar MPI para todas sus comunicaciones de paso de mensajes.

4.2. Diseño

PETSc es implementado en C con técnicas de orientación a objetos, y provee mecanismos para el chequeo consistente de errores en tiempo de ejecución mediante el valor de retorno de las funciones librería.

A fin de facilitar el acceso a las diversas estructuras de datos y algoritmos, se implementó previamente una jerarquía de clases en C++. Dichas clases son *wrappers* de los objetos nativos de PETSc (`Vec`, `Mat`, `KSP`, etc.). Adicionalmente, los errores son mapeados a excepciones.

La interfaz para Python se generó posteriormente utilizando SWIG. Algunas de las características avanzadas de esta herramienta permiten la conectar PETSc y Numarray con relativa facilidad. De esta manera se puede, por ejemplo, llamar a `MatSetValues()` con datos de un `array` de Numarray.

4.3. Ejemplo de uso

A modo de ejemplo, en la figura 3 se muestra una porción de código de Python que implementa el método de gradientes conjugados para la solución de sistemas de ecuaciones lineales. En la figura 4 se muestra la solución de la ecuación de Laplace en el cuadrado unitario uilizando diferencias finitas.

5. MÓDULO PARMETIS

5.1. Algunos comentarios sobre METIS/ParMETIS

METIS[14,15] es una familia de programas para el particionamiento de grafos no estructurados e hipergrafos, y para el cómputo de reordenamientos de matrices ralas.

Los algoritmos en los que se basan las librerías METIS constituyen el estado del arte en métodos multinivel, y producen resultados de alta calidad escalables a problemas muy grandes. Dentro de esta familia de herramientas, ParMETIS provee rutinas para el particionamiento paralelo de grafos y mallas de elementos finitos.

5.2. Diseño

La librería de ParMETIS consta de un número reducido de funciones que proveen acceso a sus algoritmos. Lamentablemente, la interfaz no provee facilidades para el chequeo de errores. Por este motivo, su utilización en un lenguaje interactivo como Python obliga a implementar

$i \Leftarrow 0$

$r \Leftarrow b - Ax$

$d \Leftarrow r$

$\delta_0 \Leftarrow r^T r$

$\delta_{new} \Leftarrow \delta_0$

While $i < i_{max}$ and $\delta_{new} > \delta_0 \epsilon^2$ do

$q \Leftarrow Ad$

$\alpha \Leftarrow \dfrac{\delta_{new}}{d^T q}$

$x \Leftarrow x + \alpha d$

$r \Leftarrow r - \alpha q$

$\delta_{old} \Leftarrow \delta_{new}$

$\delta_{new} \Leftarrow r^T r$

$\beta \Leftarrow \dfrac{\delta_{new}}{\delta_{old}}$

$d \Leftarrow r + \beta d$

$i \Leftarrow i + i$

```
def cg_solve(A,b,x,imax=50,eps=1e-6):
    """
    A, b, x   : matrix, rhs, solution
    imax, eps : max iters, tolerance
    """
    r = b.Duplicate()
    d = b.Duplicate()
    q = b.Duplicate()
    i=0
    A.Mult(x,r); r.AYPX(-1,b)
    d.Copy(r)
    delta_0   = r.Norm()
    delta_new = delta_0
    while i<imax and \
              delta_new>delta_0*eps**2:
        A.Mult(d,q)
        alfa = delta_new/d.Dot(q)
        x.AXPY(alpha,d)
        r.AXPY(alpha,q)
        delta_old = delta_new
        delta_new = r.Norm()
        beta = delta_new/delta_old
        d.AYPX(beta,r)
        i= i+1
```

Figura 3: Gradientes Conjugados en Python con PETSc

```
import petsc
petsc.Initialize()

# problem size
# ------------
m = 10         # number of points
h = 1.0/(m-1)  # grid spacing
ndof = m**2    # number of DOF's

# matrix & assembly
# -----------------
A = petsc.MatMPIAIJ(petsc.DECIDE,petsc.DECIDE,
                    ndof,ndof,5,1)
Istart, Iend = A.GetOwnershipRange();
INS_VAL = petsc.Matrix.INSERT # insert values
for I in xrange(Istart,Iend) :
    v = -1.0; i=I/n; j = I - i*n;
    if i>0  : J=I-n; A.SetValue(I,J,v,)
    if i<m-1: J=I+n; A.SetValue(I,J,v,INS_VALV)
    if j>0  : J=I-1; A.SetValue(I,J,v,INS_VALV)
    if j<m-1: J=I+1; A.SetValue(I,J,v,INS_VALV)
    v = 4.0;         A.SetValue(I,I,v,INSERT)
A.Assemble()
A.Scale(1.0/h**2)

# rigth hand side
# ---------------
b = petsc.VecMPI(petsc.DECIDE,ndof)
b.Set(1.0)

# solution vector
# ---------------
x = b.Duplicate(); x.Set(0)

# Krylov solver & precontitioner
# ------------------------------
ksp  = petsc.KSP(petsc.GetCommWorld());
ksp.SetType(petsc.KSP.CG) # conjugate gradients
pc = petsc.PC (ksp.GetPC());
pc.SetType(petsc.PC.BJACOBI) # block jacobi
P = A                        # with same matrix

# solve
# -----
ksp.SetOperators(A,P); ksp.SetRhs(b);
ksp.SetSolution(x)
ksp.Solve()

# save solution
# -------------
vw = petsc.ViewerASCII('solution.m');
vw.SetFormat(petsc.ViewerASCII.MATLAB)
x.SetName('u'); x.View(vw)
```

Figura 4: Diferencias Finitas en Python con PETSc

algunos mecanismos mínimos que informen sobre inconsistencia en los datos de entrada. Para cumplir con este requisito, se implementaron llamadas auxiliares en C que chequean los datos de entrada provistos por el usuario y retornan códigos de error.

Nuevamente, la interfaz para Python se generó utilizando SWIG, conectando ParMETIS con Numarray. Adicionalmente, se implementó una jerarquía de clases en Python (soportando diversas abstracciones como `Graph`, `Mesh`, `Weight`, `Partition`, etc.) que proveen un acceso simplificado a los algoritmos mediante una aproximación orientada a objetos.

5.3. Ejemplo de uso

A modo de ejemplo, en la figura 5 se muestra una porción de código de Python con la que particiona un malla estructurada de cuadrángulos.

6. DISPONIBILIDAD

Las herramientas presentadas en este trabajo, junto con los requisitos previos e instrucciones para la instalación están disponibles en `http://www.cimec.org.ar/python/`.

7. CONCLUSIONES

En este trabajo se presentó al lenguaje Python, se comentaron sus características fundamentales y algunas de las herramientas disponibles para el desarrollo de aplicaciones científicas.

Python resulta un lenguaje muy eficaz para el desarrollo rápido de prototipos y scripts. Como es totalmente orientado a objetos y refuerza el concepto de modularidad, es también apto para el desarrollo de aplicaciones de tamaño considerable.

Si bien es cierto que los lenguajes de *scripting* no son eficientes, la posibilidad de extensión con lenguajes compilados, tales como C/C++ o Fortran, permiten salvar este problema en las partes sensibles del código. Inclusive, todas las rutinas y librerías desarrolladas previamente son fácilmente acomodables en el nuevo entorno ganando en facilidad de uso y sin sacrificar eficiencia.

Los módulos para MPI, PETSc y ParMETIS desarrollados son buenos ejemplos del excelente soporte de Python para la extensión del lenguaje, incluso en ambientes paralelos. Estas herramientas son la base necesaria para el desarrollo de aplicaciones paralelas más complejas.

```
import mpi
import numarray as na
import parmetis

# root processor and communicator
# -------------------------------
root = 0
comm = parmetis.COMM_WORLD

# grid size
# ---------
n0, n1 = 5, 4        # nodes
e0, e1 = n0-1, n1-1  # elements

# quads generation
# ----------------
if mpi.rank == root:
    nodes = na.arange(n0*n1, shape=(n0,n1))
    icone = na.zeros(shape=(e0,e1,4))
    icone[:,:,0] = nodes[ 0:n0-1 , 0:n1-1 ]
    icone[:,:,1] = nodes[ 1:n0   , 0:n1-1 ]
    icone[:,:,2] = nodes[ 1:n0   , 1:n1   ]
    icone[:,:,3] = nodes[ 0:n0-1 , 1:n1   ]
    icone.shape = (e0*e1,4)
    del nodes

# distribute quads
# as a graph in CSR format
# ------------------------
if mpi.rank == root:
    edist = []
    eptr  = na.arange(0,e0*e1,4)
    eind  = icone.flat; del icone
else:
    edist, eptr, eind = [], [], []
edist, eptr, eind = parmetis.ScatterAdj(edist,eptr,eind,
                                        root,comm)

# dual graph construction
# -----------------------
ncnod   = 2      # number of common nodes
vtxdist = edist  # vertex distribution
xadj, adjncy = parmetis.Mesh2Dual(edist,eptr,eind,
                                  ncnod,comm)

# dual graph partition
# --------------------
part   = zeros(len(xadj)-1)  # partition array
nparts = mpi.size            # number o partitions
vwgt   = [[]]  # vertex weigts
adjwgt = []    # adjacency weigts
tpwgts = [[]]  # partition weigts
ubvec  = []    # unbalance tolerance
opts   = []    # options for ParMETIS
parmetis.PartKway(vtxdist, xadj, adjncy, vwgt, adjwgt,
                  part, nparts, tpwgts, ubvec,
                  opts, comm)
```

Figura 5: Particionamiento de mallas en Python con ParMETIS

REFERENCIAS

[1] Guido van Rossum. Python home page. http://www.python.org/, (2003).

[2] SPaSM. SPaSM parallel molecular dynamics code home page, (2001). http://bifrost.lanl.gov/MD/MD.html.

[3] Perry Greenfield, Todd Miller, Richard L. White, and J. C. Hsu. Numarray home page. http://www.stsci.edu/resources/software_hardware/numarray, (2004).

[4] Paul F. Dubois. Pyfort home page. http://pyfortran.sourceforge.net/, (2004).

[5] SciPy Home Page. Scientific tools for Python. http://www.scipy.org/, (2004).

[6] David M. Beazley. SWIG home page. http://www.swig.org/, (2004).

[7] Message Passing Interface Forum. MPI home page. http://www.mpi-forum.org/, (1994).

[8] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, (1994).

[9] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. MIT Press, 2nd. edition, (1998).

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, **22**(6), 789–828 (September 1996).

[11] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, (1994).

[12] Beowulf.org. Beowulf cluster computing home page, (2004). http://www.beowulf.org/.

[13] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, (2001). http://www.mcs.anl.gov/petsc.

[14] Kirk Schloegel, George Karypis, and Vipin Kumar. ParMETIS - parallel graph partitioning, (2001). http://www-users.cs.umn.edu/~karypis/metis/parmetis/.

[15] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). *Lecture Notes in Computer Science*, **1900**, 296–310 (2001).