A first prototype of PyACTS

Ning Kang L. A. Drummond

September 30, 2003

Abstract

The ACTS Collection is a set of software tools that help developers or programmers write high performance parallel codes for their scientific applications. PyACTS is a Python-based interface to some of the tools in the ACTS Collection. The main purpose of developing PyACTS is to provide a uniform easy-to-use external interface to existing ACTS tools, and support ACTS users to rapidly prototype their codes with the tools. In particular, for users who are new to ACTS, they will find PyACTS helpful to test and try the functionality available in the collection. Further, this training will allow users to acquire the necessary experience to develop their own applications. In the current development phase of PyACTS, part of the ScaLAPACK subroutines are being made available. This report illustrates how we develop the idea of wrapping the ACTS Collection with a high level scripting language, like Python, and a status of the development of the Python front-end interface and future plans.

1 Why Python?

Python is a very high-level, object-oriented, open source programming language, designed to optimize development speed [2]. It is often considered as an object-oriented scripting language since it is commonly used as glue to put software components together in an application, and its easy of use is another big advantage over peer languages.

The reasons for choosing Python as the front-end interface for ACTS are summarized as follows.

- The core Python language and libraries are platform-independent. It implies that most Python codes, like those that use ACTS tools, run without change on almost every computer system in use today. The only requirement is to copy the script over different systems, where Python and ACTS tools are installed.

- Python is designed to make straightforward the integration with other software components in a system. Programs written in Python can be easily blended with other languages. For instance, Python scripts can call out to existing C and C++ libraries, Java classes, and much more. Actually, it is this feature of Python that is employed to develop PyACTS. On the other hand, programs written in other languages can just run Python scripts easily by calling C and Java API (Application Programming Interface) functions.

- Some features of Python which are appealing to PyACTS:

  - No compile or link steps;
  - No type declarations;
  - High-level data-types and operations;
  - Extending and embedding in C as system glue;
  - Dynamic loading of C modules, leading to simplified extensions and smaller binary files;
  - Dynamic reloading of Python modules, letting programs be modified without stopping, which makes possible the dynamic configuration of ACTS functionality;
  - Either in batch mode or interactive mode for running scripts, making possible incremental development and testing under interactive mode;
  - Standard portable system calls;
  - Built-in and third-party libraries.

Since we expect to run ACTS tools under Python language interface, its integration support makes it very useful and convenient for the implementation of PyACTS. More specifically, a well developed C API is included in the Python framework for building extensions. It is relatively straightforward to produce the interface registration tables and module functions required to add the PyACTS extension module. In fact, because Python was built with integration in mind, it has naturally given rise to a growing library of extensions and tools, available as off-the-shelf components to Python users [2].

Because it is interpreted, Python is not as fast or efficient as static, compiled languages like C. Thus, Python alone usually is not the best tool for delivery of performance-critical components. Instead, computationally

intensive operations can be implemented as compiled extensions to Python, and coded in a low-level language in C or in Fortran. Although Python can not be used as the sole implementation language for such components, it can work well as a front-end scripting interface to them. This is exactly how PyACTS interface is implemented using Python. In PyACTS, only the Python front-end combined with the extensions is visible to users and all the programming occurs at the higher-level Python scripting level. This leads to a programming tool that is both efficient and relatively trivial to use.

2 Building PyACTS

Before we start developing PyACTS extension module, there are two fundamental problems needed to be addressed. The first and foremost is that parallelism is deeply embedded in the ACTS tools, while the current Python is not dedicated to parallel environment. Secondly, manipulating large amounts of data in Python is not efficient with its built-in standard data structures such as lists or tuples. For the first problem, we have tried to handle it with the master-slave model in a way such that Python is started on the master node and then it broadcasts modules and data to the slave nodes. It turns out to be a lot of endeavors to achieve this though. Fortunately, there is a Python extension, pyMPI, available to take care of the parallel operations for Python on distributed, parallel machines using MPI. The pyMPI extension makes implementing PyACTS relatively easy since it relieves us the burden of having to run Python on parallel architectures. Another Python extension, NumPy, which has the advantage of handling large data sets efficiently, helps to facilitate the second problem. Actually PyACTS is built upon the shoulders of Python and ACTS tools with the exploit of the two Python extensions, which greatly ease the efforts of its development.

More details on the two extensions are given in the next sections.

2.1 pyMPI

As we mentioned, the major beneficiaries of ACTS tools are those who are developing parallel scientific applications. However, the original design of Python is not targeted for parallel programming although it has a thread model. In order for PyACTS to be able to work in parallel environment, pyMPI [3], a distributed implementation of Python extended with an MPI interface, is adopted. It is a Python extension set designed to provide parallel operations for Python on distributed, parallel machines using MPI. It allows

fast prototyping of parallel code and makes it trivial to integrate Python with explicit parallel extensions written in C, C++, or Fortran, by way of parallel extension modules.

The way in which pyMPI works [4] is that it builds an alternate startup executable for Python, using the installed base of Python code modules, which enables pyMPI to use the same modules, string, system, regular expression, etc., as in Python. By default, pyMPI starts up in SPMD (Single Program Multiple Data) mode by instantiating multiple Python interpreters, each with access to world communicators and each with a unique rank corresponding to a process ID. That is to say, all processes are running the same script. The execution is asynchronous unless synchronization operations are used. The program initializes MPI on startup, sets up interfaces to MPI_COMM_WORLD and its clone PYTHON_COMM_WORLD, and initializes the parallel console I/O facility.

One of the simple ways to use pyMPI is interactively, from the prompt. Another way is to use pyMPI in batch mode, with Python script as the input file on the command line. If starting up pyMPI in the interactive mode, we just fire it up in the same way as executing a parallel job and we get what looks like the standard Python prompt (>>>), as shown in Figure 1. As mentioned earlier, the processes are running asynchronously such that the values printed out may appear in any order.

```
% poe pyMPI -procs 4 -nodes 1
>>>a = 3 + 4
>>>a
7
7
7
7
>>>
```

Figure 1: Running pyMPI interactively on IBM-SP.

2.2 NumPy

NumPy stands for Numeric Python, which is another set of extensions to the Python language [5]. It allows Python to efficiently manipulate large sets of objects organized in grid-like fashion. These sets of objects are called arrays, which can have any number of dimensions.

Poor memory management is one of the main drawbacks of Python, in

particular for large data arrays like the ones found in high performance computing applications. Python uses data structures like tuple, lists and classes that aggregate latency time and degrade the performance of an application. For this reason we rely on the NumPy services to alleviate this problem. We expect PyACTS to be somewhat efficient when tackling large scale applications as well, even when PyACTS is not meant to be used inside the large scale scientific applications. PyACTS should be used only to prototype a scientific application code by experimenting with the functionality of a tool, and later the user needs to call from C, C++, or Fortran the actual tool interfaces to benefit from the full performance of the tool.

Therefore, during the PyACTS development, we reuse the NumPy array type in its C extension modules. The PyACTS extension modules are used to make numerical libraries of ACTS, which are written in C or Fortran, accessible to Python programs. The NumPy array type has the advantage of using the same data layout as arrays in C and Fortran. Furthermore, NumPy arrays provide equally simple access to their contents from Python and from C, making it a convenient tool for running PyACTS as well as developing PyACTS.

3 Implementing ScaLAPACK Module in PyACTS

The first tool in ACTS Collection we are working on is ScaLAPACK [7]. The platform on which PyACTS is developed is the standard AIX operating system running on NERSC IBM SP RS/6000 machine. Since Python 2.2 is available on the system, `distutils` is employed to build the extension module `scalapack` for PyACTS. A `distutils` package contains a driver script, `setup.py`. This is a plain Python file, which, in a simplified way with the `scalapack` case, is presented in Figure 2.

With this `setup.py`, a file `scalapackmodule.c`, and two shared library files `libscalapack.a` and `libpympi.a`, running

```
% pyMPI setup.py build
```

will compile `scalapackmodule.c`, and produce an extension module named `scalapack` in the build directory. Depending on the system, in our case, the module file ends up in a subdirectory `build/lib.aix-5.1-mpi-2.2` and has a name like `scalapack.so`.

In the code `setup.py` as shown in Figure 2, `libpympi.a` will be available if pyMPI is installed. As NumPy array type is exploited in PyACTS, the

```
from distutils.core import setup, Extension
from distutils.sysconfig import mode
mode('parallel')
del mode

module0 = Extension( 'scalapack',
    library_dirs = ['../lib/pyacts', '../lib/pyMPI1.3'],
    libraries = ['scalapack', 'pympi'],
    include_dirs = ['../include/Numeric'],
    sources = ['./scalapackmodule.c'] )

setup( ext_modules = [module0] )
```

Figure 2: A simplified example for `setup.py` file.

directory containing header files for Numeric Python has to be included. The shared library file `libscalapack.a` is created by using all the libraries contained in ScaLAPACK and some miscellaneous help files written in C or Fortran, such as files for getting over the tricky problem of passing `char*` data type from C functions to Fortran subroutines. From our experience, the way of building shared library is very much system-dependent. The `scalapackmodule.c` contains the module method functions, the methods' registration table and initialization function. This is a typical structure to write a C extension module for Python. In addition, ScaLAPACK is coded by Fortran as well as by C, which results in the integration of Fortran routines into Python. However, calling Fortran subroutines from C involves machine-specific peculiarities. A macro `APPEND_FORTRAN` is thus defined to get around it such that for a Fortran subroutine name `SUB`, the corresponding C function is named to be `sub_`, all lower case with an underscore appended if the macro is defined, and `sub`, all lower case, if it is not defined.

With the exploit of NumPy, we add a new function `refvar` to create reference variables (zero dimensional array) to get values back from called functions, and one dimensional arrays to allocate memory needed by the underlying Fortran subroutines or C functions. It has two arguments, data type with either `double` or `int`, and the size of array with zero as default value. However, the way that reference variables behave is somewhat different from the ordinary Python scalars because they are of different type, although sometimes they have the same behaviors. The reference variables might be passed as arguments into functions wherever the Python scalars are accepted but not vice versa. Moreover, the comparisons and arithmetic

operations between reference variables and Python scalars won't work. In order to get rid of the confusion, another function `scalar` is provided to convert reference variables to Python scalars.

All the functions in PyACTS modules have corresponding functions in ACTS, with the same name and argument list. In this way, PyACTS users may easily change the prototyping Python scripts to codes written either in C or in Fortran with few modifications for serious large scale computations.

4 Examples

Here we present two examples to show how to use PyACTS interface to ScaLAPACK. Generally, the first step is to import `scalapack` module from the PyACTS package, then call the module functions in `scalapack` to do the computations.

The first example comes from the example program #2 in ScaLAPACK users' guide [7], which solves a dense linear system of equations using the `PDGESV` subroutine. The coefficient matrix and the right-hand-side matrix both are read from data files. Figure 3 gives some code pieces of the Python script for this example.

The second example comes from another ScaLAPACK example, `sample_pdsyev_call.f`, listed on the web page of ScaLAPACK Example Programs [1]. It is a program to solve symmetric eigenvalue system using subroutine `PDSYEV`, a dense eigenvalue solver. The matrix is being generated inside the code instead of reading from external file. Figure 4 shows part of the Python script for this example.

5 Future Work

The PyACTS infrastructure is built in a way such that it can be extendible, i.e., more subpackages and extension modules could be easily added into it in the future. The top-level structure of PyACTS looks like

```
- PyACTS
    - PyACTSgeneral
    - PyACTSScaLAPACK
    - PyACTSOPT++
    - PyACTSSuperLU
    - ...
```

and each subpackage of PyACTS is composed of several modules, which

might be loaded separately without importing the whole subpackage. For instance, the `PyACTSScaLAPACK` subpackage contains modules of `pblas`, `blacs`, `tools`, and `scalapack`. All of the functions shared through PyACTS are included in the `PyACTSgeneral` subpackage, which will be loaded whenever PyACTS is imported. Currently only part of the ScaLAPACK is made available in PyACTS, while code development is still under way to enrich PyACTS step by step, with more tools from ACTS to be integrated.

References

[1] The ScaLAPACK Example Programs web site:
http://www.netlib.org/scalapack/examples/

[2] Mark Lutz, *Programming Python*, Second Edition, O'Reilly & Associates, Inc., Sebastopol, CA, 2001.

[3] Patrick Miller, pyMPI - An introduction to parallel Python using MPI, www.llnl.gov/computing/develop/python/pyMPI.pdf.

[4] Patrick Miller, Parallel, distributed scripting with Python,
www.linuxclustersinstitute.org/Linux-HPC-Revolution/
Archive/PDF02/10-Miller_P.pdf.

[5] The Numerical Python (NumPy) web site:
http://www.pfdubois.com/numpy/.

[6] The Python language web site:
http://www.python.org/.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM Publications, Philadelphia, 1997.

```
import scalapack
from scalapack import *

# Reference variables
ictxt = refvar( "int" )
myrow = refvar( "int" )

# Initialize the process grid
sl_init( ictxt, nprow, npcol )
Cblacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )

# Distribute the matrix on the process grid
np_1 = max( np, 1 )
desca = refvar( "int", DLEN )
descinit( desca, matsiz, matsiz, blksiz, blksiz, zero, zero,
ictxt, np_1, info )

# Allocate memory for arrays
locsiz_a = desca[LLD-1] * nq
A = refvar( "double", locsiz_a )

# Read matrices A and B from file and distribute them
pdlaread( fileA, A, desca, zero, zero, work )
pdlaread( fileB, B, descb, zero, zero, work )

# Call the SCALAPACK subroutine to solve A*X=B
pdgesv( matsiz, rhssiz, A, ia, ja, desca, ipiv, B, ib, jb,
descb, info )

# Save solution to file
pdlawrite( fileX, matsiz, rhssiz, B, ib, jb, descb, zero,
zero, work )

# Release the process grid and free the BLACS context
Cblacs_gridexit( ictxt )
```

Figure 3: Code pieces of Python script for Example #1.

```
import scalapack
from scalapack import *

# Reference variables
iam = refvar( "int" ); nprocs = refvar( "int" )
ictxt = refvar( "int" ); myrow = refvar( "int" )
mycol = refvar( "int" ); info = refvar( "int" )

# Initialize the BLACS
Cblacs_pinfo( iam, nprocs );

# Initialize a single BLACS context
Cblacs_get( -1, 0, ictxt )
Cblacs_gridinit( ictxt, 'R', nprow, npcol )
Cblacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )

# Initialize the array descriptors
desca = refvar( "int", 50 )
descinit( desca, n, n, nb, nb, zero, zero, ictxt, lda, info )

# Allocate memory for arrays
A = refvar( "double", lda*lda )
W = refvar( "double", maxn )
Z = refvar( "double", lda*lda )

# Build a matrix that you can create with
# a one line matlab command:  hilb(n) + diag([1:-1/n:1/n])

# Ask PDSYEV to compute the entire eigendecomposition
pdsyev( 'V', 'U', n, A, ia, ja, desca, W, Z, iz, jz, descz,
work, lwork, info )

# Print out the eigenvectors
pdlaprnt(n, n, Z, iz, jz, descz, zero, zero, 'Z', nout, work)

# Release the process grid and free the BLACS context
Cblacs_gridexit( ictxt )
```

Figure 4: Code pieces of Python script for Example #2.