John von Neumann Institute for Computing

Improving Ease of Use in BLACS and PBLAS with Python

T. Drummond, V. Galiano, V. Migallón, J. Penadés

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 325-332, 2006.

http://www.fz-juelich.de/nic-series/volume33

Improving ease of use in BLACS and PBLAS with Python

Tony Drummond[a] , Vicente Galiano[b] , Violeta Migallón[c] , José Penadés[c]

[a]Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley CA 94703, USA, LADrummond@lbl.gov

[b]Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, 03202 Elche, Alicante, Spain, vgaliano@umh.es

[c]Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, 03071 Alicante, Spain, {violeta, jpenades}@dccia.ua.es

Many computational applications rely heavily in high performing numerical linear algebra operations. A good number of these applications are data and computation intensive that need to run in high performance computing environments. Researchers and engineers behind these applications have to spend a considerable amount of time efficiently developing and running codes in these environments. Developers would rather devote this time at using their computational applications. To alleviate this, we promote the reuse of robust software libraries like the ones in the ACTS Collection, and here we present our work in a subset of the high-level language interfaces, PyACTS, that help users prototype their codes using these libraries. Lastly, we compare traditional programming practices to our proposed approach. We illustrate some examples of these interfaces and their performance, we also evaluate not only their performance but also how user friendlier they are compared to the original calls.

1. Introduction

In many scientific and engineering areas there is a need to solve computational problems inside simulations to study a particular physical phenomena. Researchers and engineers behind these applications have to spend a considerable amount of time efficiently developing and running codes in high performance computing environments. Developers would rather devote this time at using their computational applications and analyzing their output.

To alleviate this, we promote the reuse of robust software libraries like the ones in the ACTS Collection [8]. ScaLAPACK [2], is one of the tools in the ACTS Collection, and has widely been used in many high performance scientific applications [7]. ScaLAPACK internally implements parallelism and scalability with the use of BLACS [6] and PBLAS [11]. In many instances, scientists and engineers find difficulty understanding the interfaces to libraries like BLACS and PBLAS because they carry arguments that are related to the parallel environment and performance in addition to arguments related to the problem at hand.

We work in the development of a collection of Python interfaces to the ACTS tools, PyACTS, and here we present the work done to the BLACS and PBLAS libraries. These two libraries involve complex distributed data structures that our proposed interfaces, PyBLACS and PyPBLAS, hide from the final user, making them easier to use.

To create these interfaces, we have chosen the Python language due to several reasons that we explain in Section 2. In Section 3, we briefly introduce the PyACTS project and the core work presented here. In Sections 4 and 5, we present the PyBLACS and PyPBLAS libraries, respectively. We compare the performance of these libraries against the performance of their native versions.

2. Python and PyMPI

Python [14] is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing managing systems. New built-in modules are easily written in C or C++.

Nowadays, scripting languages, and particularly Python language, have gained popularity inside the computational sciences community (see e.g., [12], [13]), in part because of the complexity of codes, limited computational resources, large volumes of data produced, setup of large simulation runs, and the integration of the fairly large individual codes. As computational sciences continue to generate advancements in sciences and engineering, we will continue to witness an increase in the complexity of the computational environments, software tools and applications.

In the original design, Python was not targeted for parallel programming although now it has a thread model. In order to be able to work in a parallel environment, we have evaluated two different implementations of the thread model: Scientific Python [9] and PyMPI [10]. Fundamentally, these tools are Python extension sets designed to provide parallel operations for Python on a distributed and parallel machine, using the standard Message Passing Interface (MPI) [5]. We have tested the libraries developed in this work with both interpreters and we have obtained similar results. The interpreter used in this work is PyMPI. In the rest of this section, we explain briefly the use of this interpreter.

PyMPI works by building an alternate startup executable for Python, and with this use all the installed base of Python code modules, which in turn enables PyMPI to use the same Python modules and rich functionality. By default, PyMPI starts up in MIMD mode by initiating multiple Python interpreters, each one with access to world communicators and with a unique rank corresponding to a process ID. That is, all processes are running the same script but they execute different instructions according to the value of the process ID. The execution is asynchronous unless synchronization operations are used. PyMPI can be used in both batch and interactive mode. In the batch mode, a Python script is needed as the input file on the command line (e.g., `mpirun -np 3 pyMPI script.py`). 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.

```
% mpirun -np 3 pyMPI
>>> import mpi
>>> print 'Hello world',mpi.rank
Hello world 0
Hello world 2
Hello world 1
>>>
```

Figure 1. Running PyMPI interactively.

3. The PyACTS project

The ACTS Collection [8] is a set of software tools that help programmers to write high performance scientific codes. It differs from other "software tool" projects in that it focuses primarily on software used inside an application, instead of on software used to develop an application. ACTS is an umbrella project that has brought the tools together and is funding developers to provide interoperability. ACTS tools are mostly libraries (some are C libraries, some C++ class libraries, and some are Fortran libraries). They are primarily designed to run on distributed memory computers. Portability and performance were both considerations in their design.

We work in the development of PyACTS as a set of Python based modules that provide a high level user interface to functionality available in the ACTS Collection. With PyACTS, we will provide an interoperable environment, where different libraries can be used interchangeably. For this purpose, we define a new class object in Python: `PyACTS Array`. An instance of a `PyACTS Array` contains the following properties:

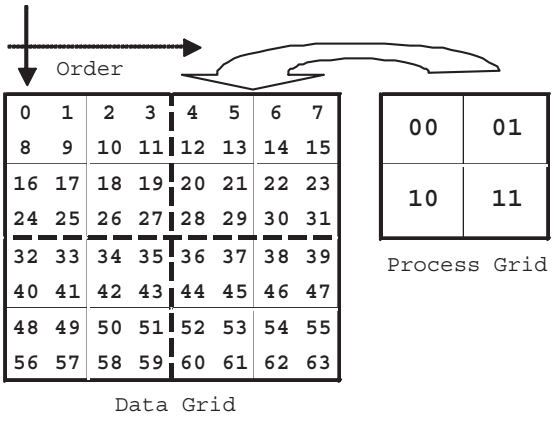
- **lib**: An integer identifies which ACTS library is used. This is useful for embedding the data types and data conversion mechanisms between different ACTS libraries.
- **desc**: Corresponds to the descriptor array of the distributed data in the selected library. Usually, libraries use descriptors to hold the global attributes.
- **data**: This property contains a Numeric Python Array (most frequently called `Numpy`) [1].

In order to provide a user friendly interface and offer a variety of levels of computational services, we must provide several groups of routines. These groups are divided as follows: *Basic Services* (a set of PyACTS routines that allows the creation, destruction, duplication, update and query of information), *I/O Services* (this group implements the PyACTS I/O routines), *Verification and Validation* (in this group, we implement routines to validate arguments and data used in PyACTS and its routines), *Errors and Exceptions* (this group implements the handling of computational or semantic errors detected during the execution) and *Data Conversion* (this group implements interoperability between different data structures used inside the different ACTS libraries).

convert.py

```
import PyACTS
import Numeric
PyACTS.gridinit(nb=2)
n,ACTS_lib=8,1                          # ScaLAPACK library
if PyACTS.iread==1:
   a=Numeric.reshape(range(n*n),[n,n])
else:
   a=None
a=PyACTS.Num2PyACTS(a,ACTS_lib)         # Convert Numpy to PyACTS
print "PyACTS Array Properties in [",
     PyACTS.myrow,",",PyACTS.mycol,"]"
print "       lib=",a.lib,";desc=",a.desc
print "       data=",a.data
PyACTS.gridexit()
```

Figure 2. Converting from Numpy to PyACTS.



(a) Data distribution

```
PyACTS Array Properties in [0,0]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[0  8 32 40  1 9 33 41
           4 12 36 44  5 13 37 45]
PyACTS Array Properties in [1,0]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[16 24 48 56 17 25 49 57
           20 28 52 60 21 29 53 61]
PyACTS Array Properties in [1,1]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[18 26 50 58 19 27 51 59
           22 30 54 62 23 31 55 63]
PyACTS Array Properties in [0,1]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[2 10 34 42  3 11 35 43
           6 14 38 46  7 15 39 47]
```

(b) Output

Figure 3. Results of executing convert.py.

In Figure 2, we show an example of converting data from Numpy to PyACTS. In this example we execute the script in four processes ("`mpirun -np 4 pyMPI convert.py`"). The `gridinit()` call initializes the processes grid according to the number of processes, np, determined by `mpirun`. The default option obtains a square grid or almost square, such that the number of processes in the grid is as close as possible to np. In the example of Figure 2, `mpirun` assigns $np = 4$, and the system obtains a grid configuration of 2×2 processes. However, the values of the grid configuration can be modified by the user (e.g., `gridinit(nb=2,nprow=4,npcol=1)` configures a 4×1 processes grid). Actually, some default parameters like block size in cyclic 2D distribution (nb) are fixed to an appropriated value. Optimal values for these parameters can be obtained using self-adapting software [4].

After executing `Num2PyACTS` in Figure 2, the variable `a` is a `PyACTS Array` instance in each process and its content is showed in Figure 3. Note that before the `Num2PyACTS` call only one processor (`iread==1`) has stored the array `a`. The data distribution between processors is automatically performed by that routine depending on the processes grid previously initialized and using BLACS and MPI routines in lower levels.

4. PyBLACS

PyBLACS is a Python based user friendly interface that we have developed to access the BLACS library. BLACS is a distributed communication library designed for linear algebra. The computational model consists of a one or two dimensional processes grid, where each process stores pieces of the matrices and vectors. The BLACS library includes synchronous send/receive routines to communicate a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions.

The names of the routines in PyBLACS are similar to the names of BLACS routines. However, in a PyBLACS routine we do not need to specify the main argument types, because PyBLACS automatically detects the correct type and calls the corresponding routine. Thus, details are hidden to

the users. In the script of Figure 4, each process sends to process [0,0] its ID; process [0,0] receives it and prints this information. This script has a similar functionality that the examples included in the standard BLACS distribution (www.netlib.org/blacs/BLACS/Examples.html). The original source Fortran code takes up around fifty lines of instructions: variable declaration and memory allocation included. On the other hand, the same functionality is coded in PyBLACS with only fourteen lines. Also, some BLACS environment parameters like `ORDER, TOP, SCOPE, ICTXT`, are hidden in the PyBLACS calls because they are not directly related to the data being processed, but more related to the computational environment and they can misguide for non advanced users.

```
from PyACTS import *
import PyACTS.PyBLACS as PyBLACS
gridinit()                              # Grid initialization
nprow,npcol,myrow,mycol=gridinfo()      # Get grid configuration
icaller = PyBLACS.pnum(myrow,mycol)     # Each process Get the own ID
if myrow==0 and mycol==0:
 for i in range[0,nprow]:
  for j in range[0,npcol]:
   if i!=0 and j!=0:
     icall=PyBLACS.gerv2d(icall,i,j)    # [0,0] receive ID from [i,j]
     print "Received ID:", icall
else:
 PyBLACS.gesd2d(icaller,0,0)            # Send ID
gridexit()
```

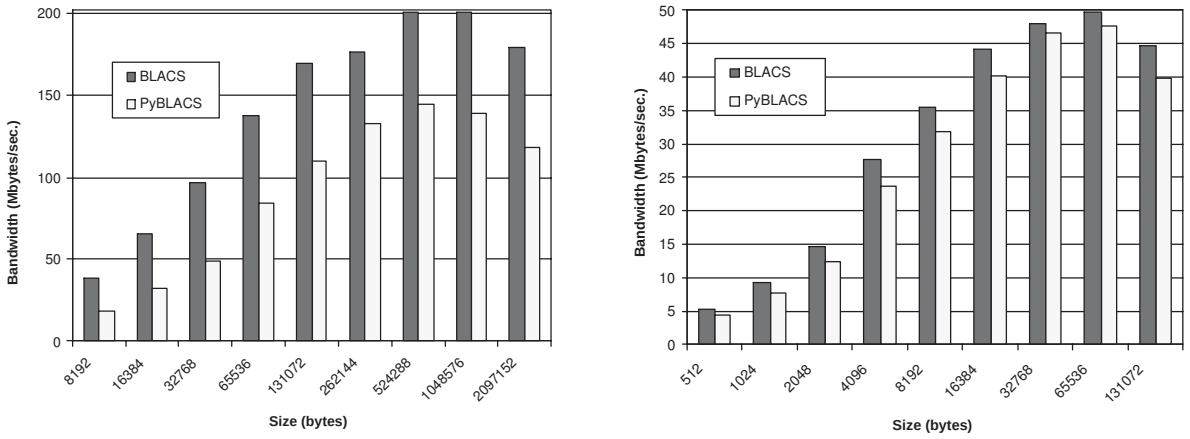
Figure 4. Example using PyBLACS.

We have implemented several tests and evaluations that compare BLACS and PyBLACS on two different computer systems. One of these is an IBM SP RS/6000, named Seaborg, located at the National Energy Research Scientific Computing Center. Seaborg is a distributed memory computer with 6080 processors. The second system that we used in our experiments is a Linux cluster (6 Intel CPU, 2 GHZ) connected with Gigabit ethernet.

One of the tests runs is shown in Figure 5. Here, we show the bandwidth obtained for different message sizes. We compare this bandwidth using the BLACS routines and the PyBLACS interfaces. In order to estimate the overhead introduced by PyBLACS, a *ping-pong* test was programmed under both Fortran and Python. Arrays of increasing size, allocated as doubles, were repeatedly sent and received. We can see that the bandwidth obtained using BLACS routines in its original form (Fortran program) is a little higher than the PyBLACS bandwidth. This is due to the fact that the Python interfaces need to check and pass arguments to the extensions (shared objects programmed in C, C++ or Fortran) [15]. However, this lost of performance is balanced with an improvement in terms of productivity, applicability and its ease of use.

5. PyPBLAS

Another high level interface we have implemented is PyPBLAS, which is a Python based interface to PBLAS [11]. PBLAS is a parallel set of BLAS routines [3]. The BLAS are high quality “building block” routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector



(a) Seaborg

(b) Linux cluster

Figure 5. BLACS and PyBLACS bandwidth comparison.

operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. BLAS and PBLAS routines are efficient, portable, and widely available. They are commonly used in the development of high quality linear algebra software.

In Figure 6, we have written a program used to test the level 3 routine (`pvgemm`). This example reads the data from the text files and store them in PyACTS Arrays. Note that this reading is done by one process (usually, [0,0] in the processes grid) and it sends the data to the rest of processes using PyBLACS to obtain a cyclic 2D distribution. After executing `Txt2PyACTS` in Figure 6, the variables `a`, `b` and `c` are PyACTS Arrays and can be used as parameters in PyACTS routines. In this example, we use the `pvgemm` routine without indicating the data types in the routine name as we must do in the PBLAS routine. This is because PBLAS routines (as well as BLACS routines) names are type-dependent, and in this case the letter '`v`' is replaced by a given data type letter (S, D, C and Z) [11]. However, as we commented in Section 4, in the Python based interfaces, it is not necessary to specify the data type because Numeric Python Arrays automatically cast them, and it is internally reused to call the corresponding routine. As we illustrate in Figure 6, we provide a set of specific tools to convert (`Scal2PyACTS`), and read data (`Txt2PyACTS`) before executing PyPBLAS routines (e.g., `pvgemm`).

For each level, we have compared the performance of both PBLAS and PyPBLAS. We have tested with different matrix and vector sizes and different number of processors and grid configuration. In Figure 7 we show the execution times for getting the solution to PDAXPY (level 1: $x + \alpha y,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n$), PDGER (level 2: $\alpha x y^t + A,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n,\ A \in \mathbb{R}^{n \times n}$) and PDGEMM (level 3: $\alpha AB + \beta C,\ \alpha, \beta \in \mathbb{R},\ A, B, C \in \mathbb{R}^{n \times n}$). The input data used in these operations were random vectors and matrices.

In Figures 7(a), 7(b) and 7(c), we present the execution times obtained in the cluster for the levels 1, 2 and 3, respectively. These results show that the times of the PyPBLAS routines are slightly higher than PBLAS times. Neverteless, it can be appreciated in these figures that the performance of PBLAS and PyPBLAS are similar and this shows that the overhead introduced by the Python interfaces is negligible. This fact is clearly appreciated in Figure 7(d), where we show the performance

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
ACTS_lib=1                          # ScaLAPACK ID
PyACTS.gridinit()                   # Grid initialization
alpha=Scal2PyACTS(1.2,ACTS_lib)   # Convert scalar to PyACTS scalar
beta=Scal2PyACTS(2,ACTS_lib)
a=Txt2PyACTS("data_a.txt",ACTS_lib) # Read Text file and
b=Txt2PyACTS("data_b.txt",ACTS_lib) # store in PyACTS Array
c=Txt2PyACTS("data_c.txt",ACTS_lib)
result=PyPBLAS.pvgemm(alpha,a,b,beta,c) # Call level 3 PBLAS routine
PyACTS2Text("data_result.txt",result)   # Write results to Text
PyACTS.gridexit()
```

Figure 6. Example of PyPBLAS: `pvgemm`.

obtained (in terms of MFLOPS) in the cluster using PBLAS from Fortran and from Python. As we can see, the performance is similar in both environments because intensive operations are made, in both cases, using PBLAS routines in lower levels.

References

[1] D. Ascher, P.F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. Numerical Python. Technical Report UCRL-MA-128569, Lawrence Livermore National Laboratory, Livermore, 2001.

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

[3] L.S. Blackford, J. Demmel, J. Dongarra, I. Du, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R.C. Whaley. An updated set of Basic Linear Algebra Subroutines (BLAS). *ACM Trans. Math. Soft*, 28(2), 135–151, 2002.

[4] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11–12):1723–1743, 2003.

[5] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walkel. *MPI: The complete reference*. The MIT Press, Cambridge, MA, 1998.

[6] J. Dongarra, R.C. Whaley. Basic Linear Algebra Communication Subprograms (BLACS). http://www.netlib.org/blacs

[7] L.A. Drummond, V. Hernández, O. Marques, J.E. Román, and V. Vidal. A Study of Robust Scientific Libraries for the Advancement of Sciences and Engineering. Proceedings of the 6th International Conference on High Performance Computing for Computational Science, Valencia, Spain, 2004.

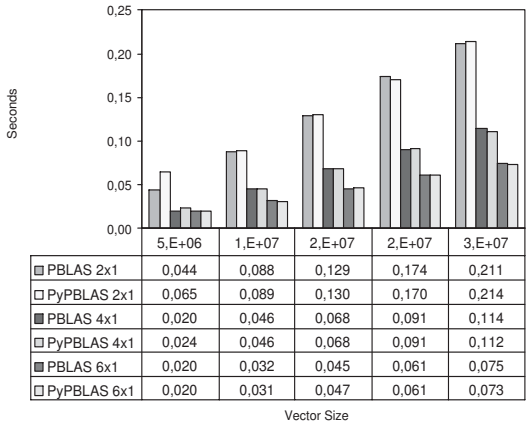
[8] L.A. Drummond and O. Marques. The ACTS Collection. Robust and high-performance tools for scientific computing: Guidelines for tool inclusion and retirement. Technical Report LBNL/PUB-3175, Computational research division, Lawrence Berkeley National Laboratory, 2002.

[9] K. Hinsen. ScientificPython User's Guide. Centre de Biophysique Moleculaire CNRS, Grenoble, France, 2002.

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

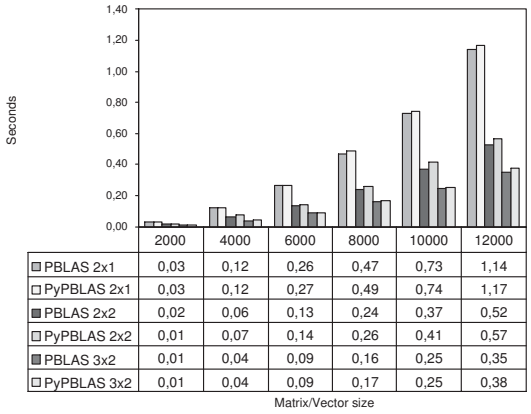
[11] NETLIB: Parallel Basic Linear Algebra Subroutines (PBLAS). http://www.netlib.org/scalapack/pblas_qref.html

[12] J. Painter and E.A. Merritt. mmLib Python toolkit for manipulating annotated structural models of



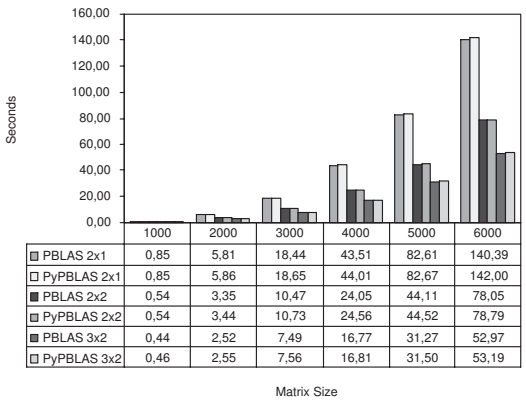
	5,E+06	1,E+07	2,E+07	2,E+07	3,E+07
PBLAS 2x1	0,044	0,088	0,129	0,174	0,211
PyPBLAS 2x1	0,065	0,089	0,130	0,170	0,214
PBLAS 4x1	0,020	0,046	0,068	0,091	0,114
PyPBLAS 4x1	0,024	0,046	0,068	0,091	0,112
PBLAS 6x1	0,020	0,032	0,045	0,061	0,075
PyPBLAS 6x1	0,020	0,031	0,047	0,061	0,073

(a) Level 1: PDAXPY. Execution Time



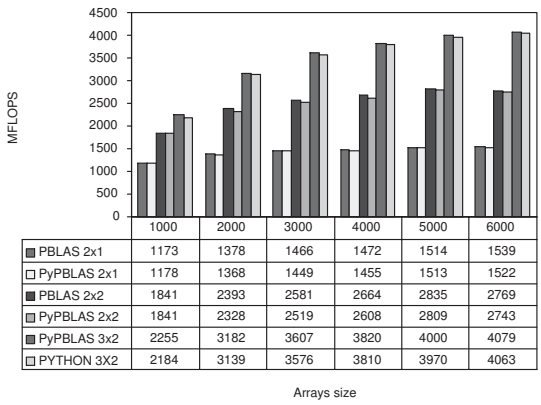
	2000	4000	6000	8000	10000	12000
PBLAS 2x1	0,03	0,12	0,26	0,47	0,73	1,14
PyPBLAS 2x1	0,03	0,12	0,27	0,49	0,74	1,17
PBLAS 2x2	0,02	0,06	0,13	0,24	0,37	0,52
PyPBLAS 2x2	0,01	0,07	0,14	0,26	0,41	0,57
PBLAS 3x2	0,01	0,04	0,09	0,16	0,25	0,35
PyPBLAS 3x2	0,01	0,04	0,09	0,17	0,25	0,38

(b) Level 2: PDGER. Execution Time



	1000	2000	3000	4000	5000	6000
PBLAS 2x1	0,85	5,81	18,44	43,51	82,61	140,39
PyPBLAS 2x1	0,85	5,86	18,65	44,01	82,67	142,00
PBLAS 2x2	0,54	3,35	10,47	24,05	44,11	78,05
PyPBLAS 2x2	0,54	3,44	10,73	24,56	44,52	78,79
PBLAS 3x2	0,44	2,52	7,49	16,77	31,27	52,97
PyPBLAS 3x2	0,46	2,55	7,56	16,81	31,50	53,19

(c) Level 3: PDGEMM. Execution Time



	1000	2000	3000	4000	5000	6000
PBLAS 2x1	1173	1378	1466	1472	1514	1539
PyPBLAS 2x1	1178	1368	1449	1455	1513	1522
PBLAS 2x2	1841	2393	2581	2664	2835	2769
PyPBLAS 2x2	1841	2328	2519	2608	2809	2743
PyPBLAS 3x2	2255	3182	3607	3820	4000	4079
PYTHON 3X2	2184	3139	3576	3810	3970	4063

(d) Level 3: PDGEMM. MFLOPS.

Figure 7. PBLAS versus PyPBLAS.

biological macromolecules. *Journal of Applied Crystallography*, 37:174–178, Part 1, 2004.

[13] J. Sáenz, J. Zubillaga, and J. Fernández. Geophysical data analysis using Python. *Computers & Geosciences*, 24(4):457–465, 2002.

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

[15] G. van Rossum and F.L. Drake Jr. *Extending and Embedding the Python Interpreter*. Network Theory Ltd, 2003.