On the performance of the Python programming language for serial and parallel scientific computations

Xing Cai[a,b], Hans Petter Langtangen[a,b] and Halvard Moe[a]
[a]*Simula Research Laboratory, P.O. Box 134, N-1325 Lysaker, Norway*
E-mail: {xingca,hpl,hm}@simula.no
[b]*Department of Informatics, University of Oslo, P.O. Box 1080, Blindern, N-0316 Oslo, Norway*

Abstract This article addresses the performance of scientific applications that use the Python programming language. First, we investigate several techniques for improving the computational efficiency of serial Python codes. Then, we discuss the basic programming techniques in Python for parallelizing serial scientific applications. It is shown that an efficient implementation of the array-related operations is essential for achieving good parallel performance, as for the serial case. Once the array-related operations are efficiently implemented, probably using a mixed-language implementation, good serial and parallel performance become achievable. This is confirmed by a set of numerical experiments. Python is also shown to be well suited for writing high-level parallel programs.

Keywords Python programming, multi-dimensional array computing, mixed-language programming, parallel computing, MPI

1. Background and introduction

There is a strong tradition among computational scientists to use compiled languages, in particular Fortran 77 and C, for numerical simulation. Demand for increased software flexibility during the last decade has also popularized the more advanced compiled languages C++ and Fortran 95. However, in recent years many computational scientists and engineers have moved away from compiled languages to interpreted problem solving environments, such as Matlab, Maple, Octave, and R (or S-Plus). Matlab has in particular been very popular and is regarded as the preferred development platform for numerical software by a significant portion of the computational science and engineering community. It may seem a paradox that computational scientists, who claim to demand as high performance as possible in their applications, solve lots of their problems in Matlab.

The success of Matlab and similar interpreted environments is due to (i) a simple and clean syntax of the command language, (ii) integration of simulation and visualization, (iii) interactive execution of commands, with immediate feedback, (iv) lots of built-in functions operating efficiently on arrays in compiled code, (v) many numerical operations that are fast enough in plain Matlab, (vi) a rich standardized library of numerical functionality that is conveniently available, and (vii) very good documentation and help support. Many scientists simply feel more productive in Matlab than with compiled languages and separate visualization tools.

The programming language Python is now emerging as a potentially competitive alternative to Matlab, Octave, and other similar environments. Python, when extended with numerical and visualization modules, shares many of Matlab's advantages, especially points (i)–(v) mentioned above. A particular advantage of Python is that the language is very rich and powerful, especially in comparison with Matlab, Fortran, and C. In particular, Python is an interpreted object-oriented language that supports operator overloading and offers a cross-platform interface to operator system functionality. Advanced C++ programmers can easily mirror their software designs in Python and even obtain more flexibility and elegance. Although Matlab supports object-oriented programming, creating classes in Python is much more convenient, and *convenience* seems to be a key issue when scientists choose an interpreted scripting language over a compiled language today.

Another advantage of Python is that interfacing legacy software written in Fortran, C, and C++ is much simpler than in most other environments. This is because Python was designed to be extendible with compiled code for efficiency, and several tools are available to ease the integration of Python and compiled code. With the properties (i)–(v) above, plus the power of the language, the extensive interfacing capabilities, and the support for parallel programming, Python represents a promising environment for doing computational science. The language has already attracted significant interest among computational scientists for some years, and an increasing growth in the scientific use of Python is taking place at the time of this writing. A key question for the community is then the numerical efficiency of Python-based computing platforms. The aim of this paper is to quantify such numerical efficiency.

We should at this point also mention that Python is being used in a much broader context than scientific programming only. In computer science, the language has a strong and steadily increasing position as a general-purpose programming tool for diverse areas such as system administration, dynamic Web sites, distributed systems, software engineering, graphical user interfaces, computational steering, search engines (Google in particular), network communication, education, and even large-scale business applications. The following references [1,4,14,20,26,27,33, 36] give a flavor of Python's versatility.

Investigating a scientific problem with the aid of computers requires software that performs a wide range of different tasks: user interfaces (files, command line, web, graphical windows), I/O management, numerical computing, data analysis, visualization, file and directory manipulation, and report generation. High (i.e., Fortran 77 alike) computational efficiency is normally only needed for a few of these tasks, and the number of statements calling for ultimate efficiency is often just a few percentage of a whole software package. The non-numerical tasks are usually much more efficiently carried out in a scripting environment like Python than in compiled languages. The classical scripting approach [15] is to let a Python script call up stand-alone simulation programs and manage the whole computational and data analysis pipeline. However, as computational scientists and engineers move to the Python platform, they will probably like to implement the numerics in Python too. In this paper we show how this can be done, using several techniques with different degrees of programming convenience and computational efficiency. More precisely, we shall compare the performance of Python with corresponding implementations in Fortran 77 and C. As we go along, we shall also point out inefficient constructs in Python programs.

It is fair to say that the core of the Python programming language is not particularly suitable for scientific applications involving intensive computations. This is mainly due to slow execution of long nested loops and the lack of efficient array data structures. However, the add-on package *Numerical Python* [18], often referred to as NumPy, provides contiguous multi-dimensional array structures with a large library of array operations implemented efficiently in C. The NumPy array computing facilities resemble those of Matlab, with respect to functionality and computational efficiency. The problem with slow loops can be greatly relieved through vectorization [16], i.e., expressing the loop semantics via a set of basic NumPy array operations, where each operation involves a loop over array entries efficiently implemented in C. The same technique is well-known from programming in Matlab and other interpreted environments where loops run slowly.

Vectorized Python code may still run a factor of 3–10 slower than optimized implementations in pure Fortran or C/C++. In those cases, or in cases where vectorization of an algorithm is cumbersome, computation-intensive Python loops should be migrated directly to Fortran or C/C++. In a Python program one cannot distinguish between a function implemented in C/C++/Fortran and a function implemented in pure Python. With the F2PY tool [9], coupling Python with Fortran is done almost automatically. As a conclusion, combining core Python with NumPy and Fortran/C/C++ code migration constitute a convenient and efficient scientific computing environment on serial computers.

Large-scale scientific applications eventually have to resort to parallel computing. The de facto parallel programming standard, with respect to both speed and portability, is to use the message passing library MPI [10,12,19].

Among packages that provide Python wrappers to the MPI routines, pypar and pyMPI [22,23] are most widely used. The pypar package concentrates only on an important subset of the MPI library, offering a simple syntax and sufficiently good performance. On the other hand, the pyMPI package implements a much larger collection of the MPI routines and has better flexibility. Both packages are implemented as a wrapper of the corresponding MPI routines in C. Consequently, some extra overhead arises when invoking such MPI calls in a Python program. To the authors' knowledge, the topic of whether the extra MPI-related overhead in Python will severely slow down a parallel scientific application has not received its deserved attention in the Python community. The follow-up question of whether it is appropriate to implement parallel scientific applications in Python has so far also remained unanswered.

The purpose of this article is thus to address the following three issues:

1. How can we achieve Fortran 77 alike computational efficiency in serial Python applications? The use of the multi-dimensional array objects from the NumPy package is clearly a prerequisite for programming Python scientific applications. However, careful use of performance enhancing techniques is essential for ensuring the full efficiency. More specifically, we will quantify the gain of migrating long native Python for-loops to vectorized form or compiled (Fortran/C) code.
2. Which Python programming structures should be adopted to parallelize scientific applications? We propose an approach of using array slices as buffers needed in exchanging messages between processors. For limiting the extra overhead associated with the parallelization, we will see that reusing the buffer arrays is necessary to avoid the cost of repeatedly constructing new array objects. Moreover, we will also see that collective extraction and filling of array slices is preferred with respect to efficiency, instead of using for-loops that traverse the array entries one by one.
3. How efficient are the resulting parallel Python applications? This is studied in detail by measuring representative parallel applications, comparing Fortran/C implementations and Python involved implementations. The measurements will show that the Python-related overhead in a parallel application is small, as long as suitable programming structures are adopted.

The remainder of this article is organized as follows. Section 2 discusses several important techniques for ensuring good performance of serial Python computations. Then, Section 3 considers the topic of using Python in parallel computations, where the issues of data partitioning, inter-subdomain communication, and Python programming structures are addressed. Later on, Section 4 reports detailed wall-clock time measurements of parallel simulations for solving two- and three-dimensional wave equations. Both pure C implementations and mixed Python-C and mixed Python-Fortran implementations are used for the numerical experiments. Finally, Section 5 concludes the article with some remarks.

In the forthcoming text we will demonstrate the efficiency of various array constructions. The timings in Section 2 were obtained on a 1.2 GHz "Mobile Intel Pentium III Processor M" laptop (model IBM X30) with 512 KB L2 cache, running Debian Linux 2.4 with Python 2.3.3, Numeric 23.1, numarray 0.9, F2PY 2.39, and gcc/g77 3.3.3 with -O3 optimization. For the measurements of parallel codes in Section 4 the same software was used, while the hardware consisted of a small Linux cluster with Pentium III 1GHz processors, inter-connected through a fast Ethernet based network, which has a theoretical peak bandwidth of 100 Mbit/s. The version of pypar was 1.9.1.

The technical content of the paper is written for readers with a basic knowledge of Python, say, from the electronic tutorial [31] or textbooks [13,15]. Knowledge of basic Fortran and C programming is also required. Even without a Python background, Fortran and C programmers should get a glimpse of Python as a platform for convenient and efficient numerical computing. We remark that the present paper only concerns the efficiency and optimization of numerical computations using Python. For other efficiency and optimization issues of Python, such as efficient use of lists and dictionaries, the readers are referred to [17,24].

2. Performance of serial Python codes

We will discuss in this section how to program efficient serial scientific applications using Python. In particular, the issues of using efficient array objects from the NumPy package, the efficiency of Python for-loops, and mixed-language programming will be studied. These issues are also of fundamental importance for an efficient parallelization of scientific applications in Python, which is to be addressed later in Section 3.

For any computation-intensive applications that are implemented in a traditional compiled language, such as Fortran 77 or C, the main data structures are normally composed of arrays. The core of the computations is thus in the form of traversing the arrays and carrying out computing operations in (nested) loops, such as do-loops in Fortran and for-loops in C. Therefore, our efficiency investigations in the present section focus on the typical loops that are used as the building blocks in scientific codes.

Array computing in Python utilizes the NumPy package. This package has a basic module defining the array data structure and efficient C functions operating on arrays. Two versions of this module exist at present: Numeric is the classical module from the mid 1990s, while numarray is a new implementation. The latter is meant as a replacement of the former, and no further development of Numeric takes place. However, there is so much numerical Python code utilizing Numeric that we expect both modules to co-exist for a long time. A lot of scripts written for Numeric will automatically also work for numarray, since the programming interface of the two modules are similar. However, there are unfortunately some differences between Numeric and numarray, which may require manual editing to replace one module by the other. (The py4cs.numpytools module from [15] helps writing scripts that can run unaltered with both modules.) Many tools for scientific computing with Python, including the F2PY program and pypar module to be used later in this paper, work at present best with (or only with) Numeric. The very useful SciPy package [28] with lots of numerical computing functionality also works best with Numeric. Therefore, most of the experiments reported in this paper involve Numeric.

All experiments reported in this section are collected in software that can be downloaded from [29] and executed in the reader's own computing environment. The source code files from [29] also document precisely how our experiments are implemented and conducted.

2.1. Computations with one-dimensional arrays

Loops over NumPy Arrays. To illustrate computing with one-dimensional arrays in Python, we first generate an array of coordinates $x_i = i/(n-1)$, $i = 0, \ldots, n-2$, and then we compute $y_i = \sin(x_i)\cos(x_i) + x_i^2$. A pure Python implementation employing a plain for-loop is given next.

Example 1 *Filling a one-dimensional array using Python loops.*

```
from Numeric import arange, zeros, Float

n = 1000001
dx = 1.0/(n-1)
x = arange(0, 1, dx)      # x = 0, dx, 2*dx, ...
y = zeros(len(x), Float)   # array of same length as x, zeros
                            # as Float (=double in C) entries
from math import sin, cos  # scalar sin and math functions
for i in xrange(len(x)):    # i=0, 1, 2, ..., length of x - 1
    y[i] = sin(x[i])*cos(x[i]) + x[i]**2
```

The arange function allocates a NumPy array with values from a start value to (but not including) a stop value with a specified increment. Allocating a double precision real-value array of length n is done by the zeros(n,Float) call. Traversing an array can be done by a for-loop as shown, where xrange is a function that returns indices 0, 1, 2, and up to (but not including) the length of x (i.e., len(x)) in this case. Indices in NumPy arrays always start at 0.

The for-loop in Example 1 requires about 9 seconds on our test computer for an array length of one million. As a comparison, a corresponding plain for-loop in Matlab is about 16 times faster than the plain Python for-loop. Switching from Numeric to numarray increases the CPU time by 32%.

Vectorization. Instead of looping over array entries, which is a slow operation in Python, one should vectorize the code [16,18], i.e., translate the loop over a sequence of calls to efficiently implemented array operations in C. To

Table 1
Comparison of various implementations of filling a one-dimensional array of length 10^6. The CPU times have been scaled by the fastest execution

Implementation	CPU time
Example 4, loop in F77	1.0
Example 2, vectorized expression, Numeric	2.4
Example 2, vectorized expression, numarray	2.4
Example 3, vectorized expression, calling I(x)	2.6
Example 1, plain loop, Numeric	22
Example 1, plain loop, numarray	29
Example 1, plain loop, with sin/ cos from Numeric	120
Example 1, plain loop, with sin/ cos from numarray	210

vectorize the present loop, we simply evaluate the expression sin(x)*cos(x) + x**2 with x as a NumPy array. This requires the sin and cos functions from the Numeric module for arrays, as shown in the next example. The loop is thus effectively implemented as calls to five C functions: sin(x), cos(x), x**2, element-wise multiplication of two arrays, and element-wise addition of two arrays. Each of these five functions uses an efficient loop over the array entries implemented in C. However, five temporary array objects with intermediate results are built behind the scene when executing the vector expression sin(x)*cos(x) + x**2.

Example 2 *Filling a one-dimensional array by vectorized code.*

```
from Numeric import arange, sin, cos
n = 1000001
dx = 1.0/(n-1)
x = arange(0, 1, dx)       # x = 0, dx, 2*dx, ...
y = sin(x)*cos(x) + x**2
```

In the present example, there is no need to allocate y beforehand, because an array is created by the vector expression sin(x)*cos(x) + x**2. Table 1 shows performance results for various vectorized versions and other implementations of the present array computation. From Table 1 we see that this vectorized version runs about 9 times faster than the pure Python version in Example 1.

The reader should notice that the expression sin(x)*cos(x) + x**2 works for both scalar and array arguments. (In contrast, Matlab requires special operators like .* for array arithmetics.) We can thus write a function I(x), which evaluates the expression for both scalar and array arguments, as illustrated below.

Example 3 *Filling a one-dimensional array by calling a function.*

```
from Numeric import arange, sin, cos, Float
n = 1000001;  dx = 1.0/(n-1);  x = arange(0, 1, dx)

def I(x):
    return sin(x)*cos(x) + x**2

y = I(x)

# could also compute with a (slow) loop:
y = zeros(len(x), Float)
for i in xrange(len(x)):
    y[i] = I(x[i])
```

The function call I(x[i]) in the above for-loop adds an overhead of about 8% compared with inlining the mathematical expressions in Example 1. In general, function calls in Python are expensive.

Pitfalls. A word of caution is necessary regarding Examples 1 and 3. In the for-loop in Example 3, we call I(x[i]) with a *scalar* argument x[i]. However, the sin and cos functions that are imported from Numeric are aimed at array arguments. When applied to scalar arguments, these versions of sin and cos functions are much slower than the similar scalar functions from the math module, typically a factor of 10 for Numeric and 20 for numarray for an operation like sin(1.2) [15, ch. 4]. A common mistake in scalar Python code is thus to import all the functions and data from the Numeric module:

```
from Numeric import *
```

Suppose we run again the code in Example 1 but drop the statement

```
from math import sin, cos
```

The omission is allowed since the statement from Numeric import * also imports sin and cos. This time we will compute sin(x[i])*cos(x[i]) + x[i]**2 by calling up the *vectorized* Numeric versions of these functions, which are much slower than their scalar counterparts from math. For Example 1 the CPU time increases by a factor of 5 shown in Table 1. If we apply numarray instead, the factor is 7. The lesson learned is that vectorized functions speed up array computations significantly, but may slow down scalar operations. The programmer therefore needs to be careful about importing the right functions.

Migrating the Computations to Compiled Code. The most efficient way of computing the x array is to implement the computations in Fortran 77. For this purpose, we first write a Fortran subroutine:

```
      subroutine loop1(x, n)
      integer n, i
      real*8 x(n), xi
Cf2py intent(in,out) x
      do i = 1, n
         xi = x(i)
         x(i) = sin(xi)*cos(xi) + xi**2
      end do
      return
      end
```

The only non-standard feature of this subroutine is the comment line starting with Cf2py. This helps the translation tool F2PY [9] with information about input and output arguments. Here, x is specified as both an input and an output argument. From Python the loop1 function can be called as

```
y = loop1(x)
# or
loop1(x)      # rely on in-place modifications of x
```

Note that input variables are arguments and output variables are returned. In the case of loop1(x), the Fortran code overwrites the input array with values of the output array. The length of the array x is needed in the Fortran code, but not in the Python call because the length is available within the NumPy array x.

In the Python program x is a NumPy array object, while the Fortran code expects a pointer to a contiguous data segment and the array size. The translation of Python data to and from Fortran data is done by a *wrapper code*. The tool F2PY can read Fortran source code files and automatically generate the wrapper code. It is the wrapper code that extracts the array size and the pointer to the array data segment from a NumPy array object and sends these to the Fortran subroutine. In the present example we may store the loop1 subroutine in a file floops.f and run F2PY like

```
f2py -m floops -c floops.f
```

This command creates an *extension module* floops containing the wrapper code and the loop1 subroutine. The extension module can be imported as any pure Python module as shown in Example 4.

Example 4 *Calling a Fortran function to fill an array.*

```
from Numeric import arange
n = 1000001;  dx = 1.0/(n-1);  x = arange(0, 1, dx)
from floops import loop1
y = loop1(x)
```

For this particular example, we see from Table 1 that migration of the loop to Fortran speeds up the code by a factor of 2.4 compared to vectorization and a factor of 22 compared to the plain Python loops in Example 1. One should notice that the relative performance of the various programming techniques depends on the type of mathematical operations performed inside the loop.

The loop could also be migrated to C. This requires more human work since we need to explicitly translate Python objects to and from C. There are tools for automatic generation of the wrapper code for C and C++ code too, SWIG [8] for instance, but none of the tools integrate Python and C/C++ seamlessly with NumPy arrays as function arguments. We will display an example of migrating loops to C in Section 2.3.

2.2. *Computations with two-dimensional arrays*

Python Loops. One-dimensional arrays appear in almost all scientific applications, whereas multi-dimensional arrays are of special interest in linear algebra, image processing, and computations associated with structured grids. Example 5 below shows how we can compute an array of function values over a uniform two-dimensional grid.

Example 5 *Evaluating a grid function via loops.*

```
from Numeric import arange, Float
from math import sin, cos
# x[i], y[j]: coordinates of grid point (i,j)
u = zeros((len(x),len(y)), Float)  # Float array of zeros

def I(x, y):
    return sin(x)*cos(y)

for i in xrange(len(x)):
    for j in xrange(len(y)):
        u[i,j] = I(x[i], y[j])
```

Note that a multi-dimensional NumPy array is allocated as a contiguous one-dimensional plain C array. This enables fast array processing and interfacing with the C code. It is important to traverse the array as it is stored in memory, i.e., row by row. Doing a Fortran-style column by column traversal in Example 5 increases the CPU time by 15% even in this interpreted loop, where one might expect lots of other overhead than scattered memory access. Inlining the I(x,y) expression showed no efficiency advantage in this particular case.

Vectorization. We may vectorize the code in Example 5 using the same technique as in Example 2. However, the one-dimensional coordinate arrays x and y must be re-shaped to two-dimensional arrays before the I(x,y) function can be called with array arguments.

Example 6 *Vectorized evaluation of a grid function.*

```
from Numeric import sin, cos
def I(x, y):
    return sin(x)*cos(y)

xv = x[:, NewAxis]
yv = y[NewAxis, :]
u = I(xv, yv)
```

Table 2
Comparison of various implementations of filling a two-dimensional grid function of size 1000 $\times$ 1000. The CPU times have been scaled by the fastest execution

Implementation	CPU time
Example 9, optimized F77 code	1.0
Example 6, vectorized expression, Numeric	1.7
Example 6, ctorized expression, numarray	2.7
Example 7, plain double loop in F77	8.2
Example 8, F77 loops with callback to Python	170
Example 5, plain Python loops, Numeric	560
Example 5, plain Python loops, numarray	580

The syntax x[:,NewAxis] copies the elements in x and adds a second dimension (NewAxis), such that the array shape becomes (len(x),1). Similarly, y[NewAxis,:] makes an array with shape (1,len(y)) where the entries in the second dimension are a copy of the elements in y. In a three-dimensional setting we would create yv = y[NewAxis,:,NewAxis] with corresponding expressions for the two other coordinate arrays x and z. The same technique is used in Matlab as well. The benefit of vectorization is apparent in Table 2.

Migrating the Computations to Fortran 77. Filling the two-dimensional array u with function values can be done in Fortran with the aid of a double do-loop. As in the loop1 example, we treat u as an input-output (intent(in,out)) array in the Fortran subroutine. Strictly speaking, u is only an output argument in the present case, but specifying intent(out) makes the wrapper code allocate the u array in every call. With intent(in,out) we avoid this and get control of all array memory allocation on the Python side. The Fortran subroutine, here named `loop2d_1`, calls a Fortran function I(x,y) for evaluating the point values. We refer to [29] for all details regarding the Fortran implementations used in this paper. In Python, `loop2d_1` is invoked as follows.

Example 7 *Fortran evaluation of a grid function in Fortran.*

```
from floops import loop2d_1
u = loop2d_1(u, x, y)
```

Sometimes it would be more flexible to perform a callback to a user-supplied Python function I(x,y), and F2PY automatically detects such callbacks and provides the necessary code.

Example 8 *Fortran evaluation of a grid function in Python.*

```
from floops import loop2d_2
def I(x, y):
    return sin(x)*cos(y)
u = loop2d_2(u, x, y, I)
```

However, the overhead in doing a callback for every grid point is substantial as can be seen in Table 2.

It turns out that simply moving the Python double loop to Fortran yields a code that is beaten by plain vectorized NumPy expressions. The vectorized expression I(xv,yv) actually runs 4.8 times faster than the plain Fortran version. To beat the NumPy code we need to mimic the NumPy operations directly in Fortran. First, we pre-compute $w_i^{(1)} = \sin x_i$, then $w_i^{(2)} = \cos y_i$, and finally we invoke a double loop to compute $u_{i,j} = w_i^{(1)} w_j^{(2)}$. The essence is to reduce the number of sin/cos calls to $2n$, while the plain double loop in Example 5 (`loop2d_1`) needs n^2 calls, where n is number of coordinates in each space direction. The optimized Fortran subroutine, called `loop2d_3`, needs scratch arrays, but F2PY has nice support for efficiently creating such scratch arrays only once in the wrapper code and feeding them to Fortran when needed. From the Python side we invoke the optimized Fortran code (using internal scratch arrays) exactly as we invoke the plain implementation `loop2d_1`:

Example 9 *Optimized Fortran evaluation of a grid function in Fortran.*

```
from floops import loop2d_3
u = loop2d_3(u, x, y)
```

For the implementation details we refer to the Fortran source code [29]. Table 2 shows that `loop2d_3` runs almost twice as fast as the NumPy vectorized code. This example illustrates that migration to compiled code may not always give a payoff in CPU time unless one is careful with the implementation.

Table 3
Comparison of various implementations of updating a three-dimensional, seven-point finite difference stencil on a 100 × 100 × 100 grid. The CPU times have been scaled by the fastest execution

Implementation	CPU time
Example 17, stand-alone C program	1.0
stand-alone F77 program	1.0
Example 2.3, loops in F77	1.0
Example 16, loops in C	1.0
Example 2.3, C++ loops with weave.inline	1.0
Example 2.3, vectorized slices with weave.blitz	1.0
Example 12, vectorized slices, Numeric	9.5
Example 12, vectorized slices, numarray	9.7
Example 11, Psyco-accelerated plain loops, Numeric	340
Example 10, plain loops, Numeric	430
Example 11, Psyco-accelerated plain loops, numarray	1150
Example 10, plain loops, numarray	1330

2.3. Implementation of finite difference schemes in 3D

As another example of array-based computations with grid data, we address the update of a three-dimensional array using a seven-point stencil. Such computations frequently arise in finite difference approximations of Laplace operators on box-shaped grids. For example, solving the heat equation $\frac{\partial u}{\partial t} = \nabla^2 u$ by an explicit finite difference scheme on a uniform box grid leads basically to loops as in Example 10 below.

Plain Python Loops.

Example 10 *Python implementation of a seven-point-stencil using for-loops.*

```
for i in xrange(1,nx):
    for j in xrange(1,ny):
        for k in xrange(1,nz):
            u[i,j,k] = um[i-1,j,k]+um[i,j-1,k]+um[i,j,k-1] - \
                       6.0*um[i,j,k]+um[i+1,j,k]+um[i,j+1,k]+um[i,j,k+1]
```

We remark that the boundary entries of u, i.e., u(0,j,k), u(nx,j,k), u(i,0,k), u(i,ny,k), u(i,j,0), and u(i,j,nz), typically need to be updated by some additional computation, but this issue will not be discussed here.

Psyco [21] is a kind of just-in-time compiler for pure Python code. We may apply Psyco to the nested loops in Example 10 to see if we can gain performance with very little effort. It is assumed that the loops in Example 10 are encapsulated in a Python function stencil3D.

Example 11 *Psyco-accelerated loops for a seven-point-stencil.*

```
import psyco
stencil3D_psyco = psyco.proxy(stencil3D)  # compile
u = stencil3D_psyco(u, um)                # compute
```

According to Table 3 the improvement is modest, about 25% when using Numeric arrays and about 15% for numarray.

Vectorized Stencil. A vectorized version of the plain Python for-loops appears next.

Example 12 *Vectorized Python implementation of a seven-point-stencil.*

```
u[1:nx,1:ny,1:nz] = um[0:nx-1,1:ny,1:nz] + \
                    um[1:nx,0:ny-1,1:nz] + \
                    um[1:nx,1:ny,0:nz-1] - \
                    6.0*um[1:nx,1:ny,1:nz] + \
                    um[2:nx+1,1:ny,1:nz] + \
                    um[1:nx,2:ny+1,1:nz] + \
                    um[1:nx,1:ny,2:nz+1]
```

It should be observed that *slicing* of array indices, such as 0:nx-1 and 2:nx+1, is heavily used in the above code segment. Slicing is a powerful property belonging to an array object, useful for extracting a desired portion of the array entries. We remark that the index slice 1:nx produces a list containing all the interior indices in the x-direction, i.e., $\{$ 1,2, ..., nx-1$\}$, so the compact syntax u[1:nx,1:ny,1:nz] means an extraction of all the interior entries of the three-dimensional array u. Similarly, the index slice 0:nx-1 produces an index list of $\{$ 0,1, ..., nx-2$\}$. The inefficiency of the above vectorized implementation is primarily due to the break-up of the addition into pair-wise operations and the need for several temporary array objects to store intermediate results. This results in overhead in both execution time and extra memory allocation/deallocation.

Measurements in Table 3 show that the vectorized implementation in Example 12 is 45 times faster than plain loops, but still an order of magnitude slower than a pure C or Fortran implementation. Therefore, if the above seven-point-stencil operation is frequently carried out in a numerical application, the quest for more speed must be achieved through code migration to a compiled language.

Migration to C++ via Weave. A very simple and convenient way of utilizing compiled code inside a Python program is to apply Weave [34], which allows "inline" C++ code. In the present example we express the three nested for-loops in C++ code:

Example 13 *C++ code for a seven-point-stencil, using* weave.inline.

```
code = r"""
for (int i=1; i<nx; i++) {
    for (int j=1; j<ny; j++) {
        for (int k=1; k<nz; k++) {
            u(i,j,k) = um(i-1,j,k)+um(i,j-1,k)+um(i,j,k-1) \
            - 6.0*um(i,j,k)+um(i+1,j,k)+um(i,j+1,k)+um(i,j,k+1);
        }
    }
}
"""
err = weave.inline(code, ['nx', 'ny', 'nz', 'u', 'um'],
      type_converters=converters.blitz, compiler='gcc')
```

Specifying `type_converters=converters.blitz` makes NumPy arrays being translated to Blitz++ arrays [5,32] in C++. Therefore, we can apply the Blitz++ array syntax in the above loop, e.g., indexing by standard parenthesis. The weave.inline function takes the C++ code along with a list of all Python variables to be transferred to C++, creates an extension module, and runs the external C++ code. In the present example u is modified in-place in the C++ code, but it is also possible to return data structures from C++ to Python.

Weave can also be used to compile vectorized array expressions with slices to C++ for possibly increased speed:

Example 14 *C++ code for a seven-point-stencil, using* weave.blitz.

```
expr = """u[1:nx,1:ny,1:nz] = um[0:nx-1,1:ny,1:nz] + \
                              um[1:nx,0:ny-1,1:nz] + \
                              um[1:nx,1:ny,0:nz-1] - \
                              6.0*um[1:nx,1:ny,1:nz] + \
                              um[2:nx+1,1:ny,1:nz] + \
                              um[1:nx,2:ny+1,1:nz] + \
                              um[1:nx,1:ny,2:nz+1]"""
weave.blitz(expr, check_size=0)
```

Note that the `check_size=0` argument turns off some array size checks and should not be used before the code is thoroughly debugged. Both weave.inline and weave.blitz avoid recompilation of the extension module if the source code has not changed since the last compilation. Hence, the overhead in compilation is only experienced in the first call.

Migration to Fortran. The loops involving the seven-point stencil can of course easily be coded directly in a compiled language. With Fortran and F2PY we only need to write the following subroutine:

Example 15 *Fortran 77 implementation of a seven-point-stencil.*

```
      subroutine stencil(u, um, nx, ny, nz)
      integer nx, ny, nz
      real*8 u (0:nx, 0:ny, 0:nz)
      real*8 um(0:nx, 0:ny, 0:nz)
Cf2py intent(in, out) u
      integer i, j, k
      do k = 1, nz-1
        do j = 1, ny-1
          do i = 1, nx-1
             u(i,j,k) = um(i-1,j,k) + um(i,j-1,k) + um(i,j,k-1)
     &                 - 6*um(i,j,k) +
     &                 um(i+1,j,k) + um(i,j+1,k) + um(i,j,k+1)
          end do
        end do
      end do
      return
      end
```

From Python we typically use stencil as follows:

```
from floops import stencil
u = stencil(u, um)
```

Another point of importance for the performance of Python-Fortran codes is the different storage of multi-dimensional arrays in C and Fortran. NumPy arrays are implemented in C and naturally applies the C storage scheme, where the first index has the slowest variation, and the last index has the fastest variation. Fortran employs the opposite scheme: the first index has the fastest variation, and the last index has the slowest variation. For a two-dimensional array this means that any array initialized in C or Python (NumPy) must be transposed before being operated in Fortran. F2PY hides the difference in storage schemes from the user, in the way that the wrapper code makes a transposed array. However, this convenient hiding of the storage difference requires allocation of extra arrays behind the scene. In the stencil call above, both u and um are copied to transposed versions, but u is returned, so if we send u to stencil again, no transposing with copy takes place. The um array, however, will always be copied in each call. This might be crucial if the stencil subroutine in Example 2.3 is called a large number of times (as is usual).

Transposing an array is avoided by the F2PY generated wrapper code if the array has Fortran storage. Such storage can be forced by calling

```
um = floops.as_column_major_storage(um)
u  = floops.as_column_major_storage(u)
```

where um and u are the NumPy array objects and floops is the name of the Fortran extension module. It is a good habit to transpose all the multi-dimensional arrays prior to sending them to Fortran. Fortunately, F2PY can make extension modules write out a message every time the wrapper code allocates an array. In this way we can easily monitor unnecessary copying that may degrade performance. We refer to [15] and the F2PY manual [9] for more detailed information on array storage issues for Python-Fortran applications.

F2PY also copies arrays when calling Fortran subroutines with incompatible array element types. Assume that we allocate an array with Float elements in Python, say by zeros(n, Float), implying that the elements are of double precision type. This corresponds to real*8 or double precision in Fortran. If we pass this array on to a Fortran subroutine expecting a single precision array (Fortran type real*4 or just real), F2PY will transparently copy the data back and forth. The code works, but there is a potentially significant performance loss because of the copying. The right way to interface with single precision Fortran code (from a performance point of view) is to work with single precision arrays in Python too. The corresponding NumPy element type is called Float32. As already mentioned, F2PY can report all copying of data in wrapper functions so performance loss due to undesired copying is easily discovered.

Migration to C. When combining Python with C or C++ code there are no concerns about different storage schemes. Hence, we might prefer C or C++ for migration of Python loops with multi-dimensional arrays. There is, unfortunately, no counterpart to F2PY for Python-C/C++ integration when NumPy arrays are used as arguments in functions. This means that we have to write the wrapper code manually. In Example 16 below we present a C function, to be called from Python, implementing the seven-point stencil. The amount of details is much larger than in the straight Fortran code from Example 2.3 since we explicitly must convert data between Python object representations and plain C data types. We refer to [2,3,15,30] for explanation of the Python-C interfacing details.

Example 16 *C implementation of a seven-point-stencil.*

```
static PyObject *stencil (PyObject *self, PyObject *args)
{
  PyArrayObject *u_array, *um_array;
  double *u, *um;
  int i, j, k, nx, ny, nz, offset0, offset1, pos;

  /* parse the input arguments coming from the Python program */
  PyArg_ParseTuple(args, "OO", &u_array, &um_array);

  nx = u_array->dimensions[0]-1;
  ny = u_array->dimensions[1]-1;
  nz = u_array->dimensions[2]-1;
  offset0 = (ny+1)*(nz+1);
  offset1 = nz+1;

  /* get direct access to the data as 1D C arrays */
  u = (double*) u_array->data;
  um = (double* )um_array->data;

  pos = offset0 + offset1; /* for skipping the lower x-dir boundary layer */
  for (i=1; i<nx; i++) {
    for (j=1; j<ny; j++) {
      for (k=1; k<nz; k++) {
        ++pos;
        u[pos] = um[pos-offset0] + um[pos-offset1] + um[pos-1]
          -6.0*um[pos] + um[pos+offset0] + um[pos+offset1] + um[pos+1];
      }
      pos += 2;       /* for skipping the two boundary layers in z-dir */
    }
    pos += 2*offset1; /* for skipping the two boundary layers in y-dir */
  }
  return Py_BuildValue("");  /* return None */
}
```

One can also return the u_array to the Python code, if desired, but the u_array data are modified in-place so the stencil update is available in Python anyway.

To make stencil accessible in a Python program, we assume that the above implementation of the function is included in a file named cloops.c, which also must contain an array of function names and pointers to be called from Python as well as an initializing function for the module. In this case the name of the module is set as cloops, and the module is created by compiling the C file and making a shared library file out of it. For the technical details on writing C extensions involving array objects, we refer to [2,15]. Finally, to use the C function stencil inside a Python program, only two lines of Python code are necessary:

```
from cloops import stencil
stencil(u, um)
# or u = stencil(u, um) if stencil returns the u array
```

Writing the C function stencil gives us complete control of all details of the Python-C interface, at least the details that are essential for the performance. In particular, it is obvious that only pointers are shuffled, i.e., there is no copying of array data. The C representation of NumPy array objects also has information about the array element type such that incompatibility between element types in Python and compiled code is easy to deal with. Arrays are also stored in the same way in Python and C. The more comprehensive human work with writing C functions may outweigh "hidden" array allocation in F2PY-generated wrapper code. The bottom line is that one either needs to understand the details of Python-C interfaces, or the details of how F2PY treats storage and element type issues.

We also remark that the NumPy array data appear in a "flat" C array, i.e., as a contiguous chunk of memory when we access the array in both C and Fortran.

The overhead in interfacing compiled code is evident from Example 16. First the arguments are packed on the Python side into a Python object (actually a tuple), and then that object must be parsed on the C side to extract the data as C types. After having performed the computations in C, the C data types must be converted to a Python object again. These behind-the-scene conversion steps are the same whether we apply C, C++, or Fortran. However, Table 3 shows that the overhead in converting data is negligible in the present application.

Stand-Alone C/F77 Programs. The last question of the present section is whether a mixed-language implementation, Python-Fortran or Python-C, is able to produce satisfactory computation speed compared with pure C or Fortran code. To this end, we implement the seven-point-stencil operation entirely in C in Example 17.

Example 17 *Pure C implementation of a seven-point-stencil.*

```
for (i=1; i<nx; i++)
  for (j=1; j<ny; j++)
    for (k=1; k<nz; k++)
      u[i][j][k] = um[i-1][j][k] + um[i][j-1][k] +
                   um[i][j][k-1] -6.0*um[i][j][k] +
                   um[i][j][k+1] + um[i][j+1][k] + um[i+1][j][k];
```

The corresponding Fortran 77 code is merely a main program calling the Fortran stencil function. We can observe from Table 3 that the mixed Python-C/C++/F77 implementations give optimal speed for this particular example of seven-point stencil operation, which is approximately ten times faster than that of the vectorized version. The pure Python implementation using for-loops is obviously useless in practice.

Regarding user friendliness, we note that allocating the storage for a three-dimensional array in C requires several lines of code (which we did not show in Example 17), in contrast to a single statement in Python. In addition, the syntax of indexing one entry in a three-dimensional C array, such as u[i][j][k], is also slightly less user-friendly than u[i,j,k] in Python.

An efficiency study similar to the one above, but addressing the two-dimensional problem with a five-point stencil, can be found in [25].

3. Parallelizing serial Python codes

Before running on any parallel computing platform, a serial program must first be parallelized accordingly. In this section, we will explain how to parallelize serial structured Python computations. Our emphasis will be on showing that the high-level programming of Python gives rise to parallel codes of a clean and compact style.

3.1. Message-passing based parallelization

There are several different programming approaches to implementing parallel programs; see e.g. [11,35]. In this paper, however, we have chosen to restrict our attention to message-passing based parallelization [12,19]. This is because the message-passing approach is most widely used and has advantages with respect to both performance and flexibility. We note that a message between two neighboring processors contains simply a sequence of data items, typically a vector of numerical values.

As a concrete example of message passing programming, let us consider the simple case of parallelizing a five-point-stencil operation, which is in fact a two-dimensional simplification of Example 10. That is, the stencil operation is carried out on the entries of a two-dimensional global array um, and the results are stored in another two-dimensional global array u.

Work Load Division. The first step of parallelizing the five-point-stencil operation is to divide the computational work among P processors, which are supposed to form an $N_x \times N_y = P$ lattice. A straightforward work division is to partition the interior array entries of u and um disjointly into $P = N_x \times N_y$ small rectangular portions, using horizontal and vertical "cutting lines". If the dimension of u and um is $(n_x + 1) \times (n_y + 1)$, then the $(n_x - 1) \times (n_y - 1)$ interior entries are divided into $N_x \times N_y$ rectangles. For a processor identified by an index tuple (l, m), where $0 \leqslant l < N_x$ and $0 \leqslant m < N_y$, it is assigned with $(n_x^l - 1) \times (n_y^m - 1)$ interior entries. Load balancing requires that the processors have approximately the same work amount, i.e.,

$$n_x^l - 1 \approx (n_x - 1)/N_x \quad \text{and} \quad n_y^m - 1 \approx (n_y - 1)/N_y.$$

Local Data Structure. The above work division means that processor (l, m) is only responsible for updating $(n_x^l - 1) \times (n_y^m - 1)$ entries of u. To avoid having to repeatedly borrow um values from the neighboring processor when updating those u entries lying immediately beside each internal boundary, as depicted in Fig. 1, the data structure of the local arrays should include one layer of "ghost entries" around the $(n_x^l - 1) \times (n_y^m - 1)$ assigned entries. That is, two local two-dimensional arrays u_loc and um_loc, both of dimension $(n_x^l + 1) \times (n_y^m + 1)$, are used on processor (l, m). Note that no processor needs to construct the entire u and um arrays, and that computing the layer of ghost entries in u_loc is the responsibility of neighboring processors.

Local Computation. Suppose that the dimensions n_x^l and n_y^m are represented as variables nx_loc and ny_loc in a Python program. The parallel execution of a five-point-stencil operation, distributed on $P = N_x \times N_y$ processors, can be implemented as the following Python code segment:

Example 18 *Local Python implementation of a five-point-stencil.*

```
u_loc = zeros((nx_loc+1,ny_loc+1), Float)
um_loc = ones((nx_loc+1,ny_loc+1), Float)
for i in xrange(1,nx_loc):
    for j in xrange(1,ny_loc):
        u_loc[i,j] = um_loc[i,j-1] + um_loc[i-1,j] \
                     -4*um_loc[i,j] + um_loc[i+1,j] + um_loc[i,j+1]
```

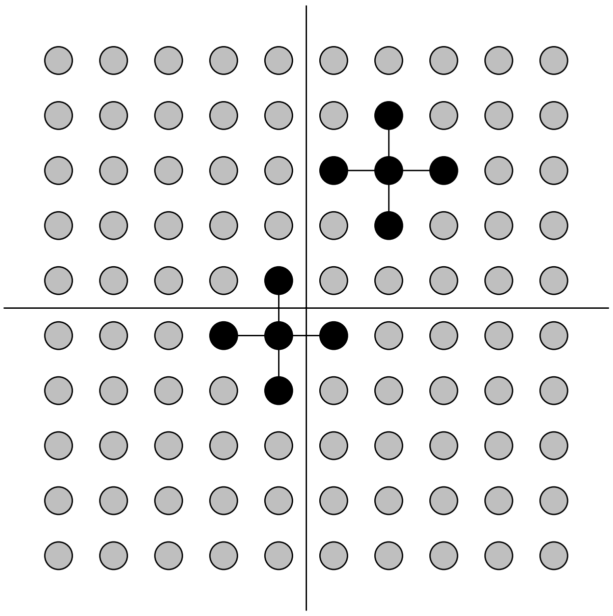
Fig. 1. An example of “data sharing” on the internal boundaries between neighboring processors for a parallel five-point-stencil operation.

We remark that the difference between Example 18 and its corresponding global implementation is that `nx_loc` and `ny_loc` replace nx and ny. Moreover, the local array objects use a “ loc” suffix to their names to indicate that they contain only a portion of the global arrays u and um.

Need for Communication. After the code segment in Example 18 is concurrently executed on each processor, we have only updated the $(n_x^l - 1) \times (n_y^m - 1)$ interior entries of `u_loc`. In a typical parallel application, the `u_loc` array will probably participate in later computations, with a similar role as that of the `um_loc` array in Example 18. Therefore, we also need to update the layer of ghost entries in `u_loc`. This additional update operation means that two and two neighboring processors exchange values across their internal boundary. More specifically, when processor (l, m) has a neighbor $(l + 1, m)$ in the upper x-direction (i.e., $l < N_x - 1$), processor (l, m) needs to send a vector containing values of `u_loc[nx_loc-1,:]`, as a message, to processor $(l + 1, m)$. In return, processor (l, m) receives a vector of values from processor $(l + 1, m)$, and these received values are assigned to the entries of `u_loc[nx_loc,:]`. This procedure of message exchange across an internal boundary has also to be carried out with each of the other possible neighbors, i.e., processors $(l - 1, m)$, $(l, m + 1)$, and $(l, m - 1)$.

3.2. Python implementation of parallelization tasks

Although the parallelization example from Section 3.1 is extremely simple, it nevertheless demonstrates the generic tasks that are present in any parallel numerical code. Using an abstract description, we can summarize these generic tasks as follows:

1. Work load partitioning, i.e., divide the global data into local data to be owned exclusively by the processors.
2. Serial computations using only local data items on each processor.
3. Preparation of the outgoing messages, i.e., extract portions of some local data into vectors of values, where each vector works as a message.
4. Message exchange between neighboring processors.
5. Extraction of the incoming messages, i.e., update portions of some local data (e.g. the ghost entries) using values of the incoming messages.

Of the above five generic tasks, the first task normally only needs to be executed once, in the beginning of a parallel application, whereas the second task concerns purely serial codes. Therefore, under the assumption that the serial codes have good serial performance, the overall performance of a parallelized application depends heavily on the last three tasks. We will show in this section how these communication related tasks can be implemented in Python. The efficiency of the parallel Python implementation is studied by detailed measurements in Section 4.3.4.

Our attention will be restricted to parallelizing numerical applications that are associated with structured computational meshes. This is because ensuring the parallelization quality of an unstructured computing application requires the *same* principles as for a structured computing application. Like a structured computing application, an unstructured computing application also uses arrays to constitute its data structure. The typical difference is that an unstructured computing application only uses flat one-dimensional arrays, and traversing the entries of an array is often in an unstructured fashion, and the number of neighbors may also vary considerably from processor to processor.

Python MPI Bindings Through Pypar. Let us start with looking at message exchanges between the processors in a Python program. For the purpose of efficiency, we have chosen the pypar package because of its light-weight implementation and user-friendly syntax. As we have mentioned before, the pypar package provides a Python wrapper of a subset of the standard MPI routines. The underlying implementation of pypar is done in the C programming language, as an extension module that can be loaded into Python. The following is an example of using the two most important functions in pypar, namely pypar.send and pypar.receive:

Example 19 *An example of sending and receiving messages using pypar.*

```
import pypar                  # import the pypar module
myid = pypar.rank()          # rank of a processor
numprocs = pypar.size()      # total number of processors
msg_out = zeros(100, Float)  # a NumPy array to be communicated

if myid == 0:
    pypar.send (msg_out, destination=1)
    msg_in = pypar.receive(numprocs-1)
else:
    msg_in = pypar.receive(myid-1)
    pypar.send (msg_out, destination=(myid+1)%numprocs)

pypar.finalize()             # finish using pypar
```

In the above simple example, each processor sends out a message to its “downstream” neighbor and receives a message from its “upstream” neighbor. Compared with an equivalent C implementation, we can see that the syntax of the pypar implementation is greatly simplified. The reason is that most of the arguments to the send and receive functions in pypar are optional and have safe default values. Of particular interest to parallel numerical applications, the above example demonstrates that a NumPy array object can be used directly in the send and receive functions. However, to invoke the most efficient version of the send and receive commands, which avoid internal safety checks and transformations between arrays and strings of characters, we must assign the particular argument bypass with value True. That is,

```
    pypar.send (msg_out, destination=to, bypass=True)
    msg_in = pypar.receive (from, buffer=msg_in_buffer, bypass=True)
```

We remark that using bypass=True in a pypar.receive function must also be accompanied by specifying a message buffer, i.e., `msg_in_buffer` in the above example. The message buffer is assumed to be an existing Python array object of appropriate dimension. In this way, we can avoid the situation that pypar creates a new internal array object with storage allocation every time the receive function is invoked. In Section 4.3.1, measurements will show that the pypar-induced overhead of the Python MPI-wrapper is of an acceptable low level.

Array Slicing in Preparing and Extracting Messages. When parallelizing array-based structured computations, we only use portions of a local array (such as its ghost layer) in the message exchanges. The slicing functionality of NumPy arrays is very important for both the task of preparing an outgoing message and the task of extracting an incoming message. The resulting Python implementation is extremely compact and efficient. Let us give a pypar implementation of the operation that updates the ghost layer of the local two-dimensional array `u_loc`, which is needed after executing the code segment in Example 18.

Example 20 *Updating the ghost layer of a local two-dimensional array.*

```
if has_upper_x_neighbor:
    pypar.send (u_loc[nx_loc-1,:], destination=upper_x_neighbor_id,
    bypass=True)
    u_loc[nx_loc,:] = pypar.receive (upper_x_neighbor_id, \
                                     buffer=buffer_x, bypass=True)

if has_lower_x_neighbor:
    pypar.send (u_loc[1,:], destination=lower_x_neighbor_id, bypass=True)
    u_loc[0,:] = pypar.receive (lower_x_neighbor_id, \
                                buffer=buffer_x, bypass=True)

if has_upper_y_neighbor:
    pypar.send (u_loc[:,ny_loc-1], destination=upper_y_neighbor_id,
    bypass=True)
    u_loc[:,ny_loc] = pypar.receive (upper_y_neighbor_id, \
                                     buffer=buffer_y, bypass=True)

if has_lower_y_neighbor:
    pypar.send (u_loc[:,1], destination=lower_y_neighbor_id, bypass=True)
    u_loc[:,0] = pypar.receive (lower_y_neighbor_id, \
                                buffer=buffer_y, bypass=True)
```

It should be noted that the above example has merged together the three generic tasks, namely preparing outgoing messages, exchanging messages, and extracting incoming messages. The actual sequence of invoking the send and receive commands may need to alternate from processor to processor for avoiding dead locks; see e.g. [12]. It is of vital importance for the performance to use the slicing functionality to prepare outgoing messages, instead of using for-loops to copy a desired slice of a multi-dimensional array, entry by entry, into an outgoing message. Similarly, extracting the values of an incoming message should also be accomplished by the slicing functionality. For a receive command that is frequently invoked, it is also important to reuse the *same* array object as the message buffer, i.e., `buffer_x` for the x-direction communication and `buffer_y` for the y-direction in Example 20. These buffer array objects should thus be constructed once and for all, using a contiguous underlying memory storage. The measurements in Section 4.3.4 will show that the array slicing functionality keeps the extra cost of preparing and extracting messages at an acceptably low level. This is especially important for parallel three-dimensional structured computations.

A Simple Python Class Hierarchy. We can observe that four of the five generic tasks in most parallel numerical applications are independent of the specific serial computations. Therefore, to simplify the coding effort of parallelizing array-based structured computations, we have devised a reusable class hierarchy in Python. The name of its base class is BoxPartitioner, which provides a unified Python interface to the generic task of work load partitioning (plus internal data allocation) and the three generic communication-related tasks. In addition to several internal variables, two of the major functions in BoxPartitioner are declared as `prepare_communication` and `update_internal_boundaries`. The first function is for dividing the global computational work and preparing some internal data structures (such as allocating data buffers for the incoming messages), whereas the latter is meant for the update operation as described in Example 20.

Based on BoxPartitioner, three subclasses have been developed, namely BoxPartitioner1D, BoxPartitioner2D, and BoxPartitioner3D. These first-level subclasses are designed to handle space dimension-specific operations, such as extending the `prepare_communication` function in class BoxPartitioner. However, all the MPI-related operations are deliberately left out in the three first-level subclasses. Specific MPI operations are introduced in concrete second-level subclasses. For example, PyMPIBoxPartitioner2D and PyParBoxPartitioner2D are two subclasses of BoxPartitioner2D, using the two different MPI wrapper modules. With the BoxPartitioner hierarchy, the parallel coding effort is reduced to (1) instantiating an object of a suitable second-level subclass of BoxPartitioner at the start of the program and (2) inserting calls to the functions of BoxPartitioner at appropriate locations.

As a complete parallelization example, let us consider again the parallel execution of the five-point-stencil operation. This time we assume that the global um array contains the following values:

$$um_{i,j} = x_i^2 + 2y_j, \quad x_i = \frac{i}{n_x}, \quad y_j = \frac{j}{n_y}.$$

In the following example, we put the code segment from Example 18 into a complete parallel Python program, where the chosen second-level subclass is PyParBoxPartitioner2D.

Example 21 *A complete Python program using* PyParBoxPartitioner2D.

```
from Numeric import arange, zeros, Float
nx = 1000              # number of global cells in x-dir
ny = 1000              # number of global cells in y-dir
Nx = 4                 # number of procs in x-dir
Ny = 4                 # number of procs in y-dir
dx = 1.0/nx            # help variable
dy = 1.0/ny            # help variable

from BoxPartitioner import *   # pypar is imported inside BoxPartitioner
myid = pypar.rank ()
numprocs = pypar.size ()
box = PyParBoxPartitioner2D (my_id=my_id, num_procs=numprocs,\
                             global_num_cells=[nx,ny], num_parts=[Nx,Ny])
box.prepare_communication ()              # dividing work load etc

nx_loc,ny_loc = box.get_num_loc_cells ()  # dimensions of local arrays
lo_ix0,lo_ix1 = box.get_map_lower_idx ()  # local->global mapping info
u_loc = zeros((nx_loc+1,ny_loc+1), Float) # local array allocation
um_loc= zeros((nx_loc+1,ny_loc+1), Float) # local array allocation

for i in xrange(0,nx_loc+1):              # initialization of um_loc
    for j in xrange(0,ny_loc+1):
        xi = (lo_ix0+i)*dx; yj = (lo_ix1+j)*dy
        um_loc[i,j] = xi**2 + 2.0*yj

for i in xrange(1,nx_loc):                # local computation
    for j in xrange(1,ny_loc):
        u_loc[i,j] = um_loc[i,j-1] + um_loc[i-1,j] \
                     -4*um_loc[i,j] + um_loc[i+1,j] + um_loc[i,j+1]

box.update_internal_boundaries;  # update ghost layer of u_loc
```

In Example 21, `get_num_loc_cells` and `get_map_lower_idx` are two small but very useful functions of BoxPartitioner. They are used to access the work load division results, which become available after the `prepare_communication` function is invoked. Regarding the `get_map_lower_idx` function in particular, the return value `lo_ix0` tells, for the x-direction, from which position in a global array the local array corresponds to. (The return value `lo_ix1` is for the y-direction.) Therefore, `get_num_loc_cells` and `get_map_lower_idx` together determine a mapping from the entries in a local array to those in a (conceptually existing) global array. We remark that the values of $n_x = n_y = 1000$ and $N_x = N_y = 4$ (i.e., $P = 16$) are hard-coded in Example 21, but it is straightforward to modify the program to accept arbitrary values of n_x, n_y, N_x, and N_y at runtime.

4. Experiments and measurements

In this section, the findings from the previous sections will be used in parallelizing a Python program for solving partial differential equations. Our focus will be on comparing the performance of mixed-language (Python-C and Python-Fortran) implementations against that of an implementation in pure C. It will be shown that the three implementations are fully comparable with respect to both serial and parallel performance.

The purpose of this section is to evaluate the performance of parallel Python code in a real scientific application. Considering numerical solution of partial differential equations by finite difference, volume, or element methods on structured grids, the most important type of operation from a performance point of view is the action of a stencil on a grid function. This action occurs in explicit finite difference schemes, matrix-vector products in Krylov subspace methods [7] useful for implicit schemes, as well as restriction, prolongation, and smoothers in multigrid solvers or preconditioners [6]. Even in complicated multi-physics codes, applying a stencil on the grid is usually the primary performance bottleneck. To study the efficiency of stencil actions we have therefore chosen an application where this action constitutes the entire computational kernel: a linear wave equation solved by an explicit finite difference scheme.

We remark that although the chosen mathematical model and numerical scheme are very simple, the measurements obtained in Section 4.3 are sufficiently representative for parallelizing array-based structured computations. This is because the generic parallelization-related tasks remain the same for more complicated programs. In fact, a simple parallel program is best at revealing the communication overhead induced by Python, as more complicated parallel programs tend to have a higher computation/communication ratio.

4.1. Mathematical model and numerical strategy

The mathematical problem reads

$$\frac{\partial^2 u(\boldsymbol{x},t)}{\partial t^2} = c^2\nabla^2 u(\boldsymbol{x},t) + f(\boldsymbol{x},t) \quad \text{in } \Omega, \tag{1}$$

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{on } \partial\Omega, \tag{2}$$

where c is a constant representing the wave speed, and the coordinates $\boldsymbol{x}$ are in either two or three space dimensions. The above mathematical model Eqs (1)–(2) is to be supplemented with initial conditions of the form:

$$\frac{\partial u(\boldsymbol{x},0)}{\partial t} = 0 \quad \text{and} \quad u(\boldsymbol{x},0) = I(\boldsymbol{x}). \tag{3}$$

The numerical scheme of interest here is of the explicit type, which translates into the following scheme for stepping forward one time step in the three-dimensional case:

$$\begin{aligned} u^{l+1}_{i,j,k} = &-u^{l-1}_{i,j,k} + 2u^{l}_{i,j,k} \\ &+ c^2\frac{\Delta t^2}{\Delta x^2}\left(u^l_{i-1,j,k} - 2u^l_{i,j,k} + u^l_{i+1,j,k}\right) \\ &+ c^2\frac{\Delta t^2}{\Delta y^2}\left(u^l_{i,j-1,k} - 2u^l_{i,j,k} + u^l_{i,j-1,k}\right) \\ &+ c^2\frac{\Delta t^2}{\Delta z^2}\left(u^l_{i,j,k-1} - 2u^l_{i,j,k} + u^l_{i,j,k+1}\right) \\ &+ \Delta t^2 f(x_i, y_j, z_k, l\Delta t). \end{aligned} \tag{4}$$

Here, we have assumed that the superscript l indicates the time level and the subscripts i, j, k refer to a uniform spatial computational mesh with constant cell lengths Δx, Δy, and Δz.

4.2. *Computational kernels*

4.2.1. *Python implementations*

In a pure Python serial implementation using while- and for-loops, the entire time stepping procedure of the three-dimensional numerical scheme Eq. (5) looks like:

```
while t <= tstop:
    t_old = t;  t += dt

    # update all inner points:
    for i in xrange(1,nx):
        for j in xrange(1,ny):
            for k in xrange(1,nz):
                u[i,j,k] = - um2[i,j,k] + 2*um[i,j,k] + \
                           Cx2*(um[i-1,j,k]-2*um[i,j,k]+um[i+1,j,k])+ \
                           Cy2*(um[i,j-1,k]-2*um[i,j,k]+um[i,j+1,k])+ \
                           Cz2*(um[i,j,k-1]-2*um[i,j,k]+um[i,j,k+1])+ \
                           dt2*f(x[i], y[j], z[k], t_old)

    # enforce boundary condition (real codes not listed):
    # i=0,  loop over all j,k; u[i,j,k] = g(x[i], y[j], z[k], t)
    # i=nx, loop over all j,k; u[i,j,k] = g(x[i], y[j], z[k], t)
    # j=0,  loop over all i,k; u[i,j,k] = g(x[i], y[j], z[k], t)
    # j=ny, loop over all i,k; u[i,j,k] = g(x[i], y[j], z[k], t)
    # k=0,  loop over all i,j; u[i,j,k] = g(x[i], y[j], z[k], t)
    # k=nz, loop over all i,j; u[i,j,k] = g(x[i], y[j], z[k], t)

    # data shuffle before continuing the next time step
    tmp = um2; um2 = um; um = u; u = tmp
```

Here, the three-dimensional arrays u, um, and um2 refer to u^{l+1}, u^l, u^{l-1}, respectively. The variables Cx2, Cy2, Cz2, dt2 contain the constant values $c^2\Delta t^2/\Delta x^2$, $c^2\Delta t^2/\Delta y^2$, $c^2\Delta t^2/\Delta z^2$, and Δt^2. For simplicity of the presentation, we have skipped the part of enforcing the boundary condition in the above code segment. Moreover, it should be noted that on the final line of "data shuffle", only the references of the array objects are shuffled, without actually copying the contents of the arrays between each other.

For a vectorized implementation, the Python code segment will look like:

```
while t <= tstop:
  t_old = t;  t += dt

  # update all inner points:
  u[1:nx,1:ny,1:nz] = - um2[1:nx,1:ny,1:nz] + 2*um[1:nx,1:ny,1:nz]+ \
     Cx2*(um[0:nx-1,1:ny,1:nz]-2*um[1:nx,1:ny,1:nz]+um[2:nx+1,1:ny,1:nz])+\
     Cy2*(um[1:nx,0:ny-1,1:nz]-2*um[1:nx,1:ny,1:nz]+um[1:nx,2:ny+1,1:nz])+\
     Cz2*(um[1:nx,1:ny,0:nz-1]-2*um[1:nx,1:ny,1:nz]+um[1:nx,1:ny,2:nz+1])+\
     dt2*f(xv[1:nx,1:ny,1:nz],yv[1:nx,1:ny,1:nz],zv[1:nx,1:ny,1:nz],t_old)

  # enforce boundary conditions:
  i = 0;  u[i,:,:] = g(x[i], y[:,NewAxis], z[NewAxis,:], t)
```

```
    i = nx; u[i,:,:] = g(x[i], y[:,NewAxis], z[NewAxis,:], t)
    j = 0;  u[:,j,:] = g(x[:,NewAxis], y[j], z[NewAxis,:], t)
    j = ny; u[:,j,:] = g(x[:,NewAxis], y[j], z[NewAxis,:], t)
    k = 0;  u[:,:,k] = g(x[:,NewAxis], y[NewAxis,:], z[k], t)
    k = nz; u[:,:,k] = g(x[:,NewAxis], y[NewAxis,:], z[k], t)

    # data shuffle before continuing the next time step
    tmp = um2; um2 = um; um = u; u = tmp
```

where the additional three-dimensional arrays xv, yv, and zv arise from "reshaping" three one-dimensional coordinate arrays x, y, and z as indicated in Example 6. Moreover, we remark that g is a Python function that enforces the boundary condition (2) and allows vectorized executions.

4.2.2. Mixed-language implementations

As we have learned from Examples 10–17, vectorization will not produce sufficient efficiency for this type of computations. So further optimization lies in a mixed-language implementation of the computational kernel. This is similar to the stencil function in Section 2.3, so we do not list the content of the migrated C or Fortran functions here.

The main extension in the wave simulation application is that we call the source term $f(x, y, z, t)$ as a part of the scheme inside the loop. When this f function is implemented in compiled code performance is ensured. However, it is often convenient to supply such data as a Python function. Callback to Python from Fortran is automatically generated by F2PY if f is provided as argument to stencil, while in the C version of stencil callbacks to Python must be coded manually. Unfortunately, point-wise callbacks inside loops are extremely expensive [15, ch. 9,10]. There exist several techniques [15, ch. 9,10] that can rapidly feed f from Python to stencil, but to avoid the complexity of clever callbacks in the timings of this paper we have simply chosen to specify f in compiled code and by-pass the whole problem.

The previously listed while-loops are not much changed after migrating the code to compiled languages: the Python loops over the internal grid points or the vectorized slice over the same points are replaced by a call like

```
   u = extension_module.update_interior_pts(u, um, um2, x, y, z,
                                            Cx2, Cy2, Cz2, dt2, t_old)
```

For the Fortran version it is smart to force u, um, and um2 to have column major storage before the first call. Otherwise there will be some extra copying during the first time steps until all three arrays have been returned from the extension module function (recall that we switch pointers, e.g., um points to u at the end of each time level).

4.2.3. Parallelization

During the parallelization, the principles presented in Sections 3 apply directly to this type of explicit finite difference schemes. The main ingredients are (1) the global computational mesh is divided into a lattice of $P = N_x \times N_y$ or $P = N_x \times N_y \times N_z$ processors, (2) each processor only constructs its local array objects including the ghost layers, (3) during each time step, message preparation-exchange-extraction is carried out after the local computations are finished.

Using class PyParBoxPartitioner2D or class PyParBoxPartitioner3D, which have been explained in Section 3.2, we are able to straightforwardly transform a serial Python wave simulation program into a parallel one. The parallelization is done in exactly the same way as in Example 21. That is, an object of PyParBoxPartitioner2D or PyParBoxPartitioner3D is instantiated in the beginning, whereas the functions `prepare_communication`, `get_num_loc_cells`, and `get_map_lower_idx` are used to construct the local arrays. Regarding the time usage by this part of parallelization, the overhead associated with work load division is completely negligible, due to the ease of load balancing for structured data.

The needed communication simply invokes the `update_internal_boundaries` function for the distributed u array during each time step. Almost all the MPI details are hidden from the parallel program. On the basis of the serial Python wave simulation program, the parallel program needs to insert fewer than twenty function calls to the

Table 4
Comparison between C-version MPI and pypar-layered MPI on a Linux-cluster, with respect to the latency and bandwidth

	Latency	Bandwidth
C-version MPI	133×10^{-6} s	88.176 Mbit/s
pypar-layered MPI	225×10^{-6} s	88.064 Mbit/s

BoxPartitioner class hierarchy. The amount of migrated code is considerably smaller than the remaining code in Python, and in more complicated programs the percentage of the migrated code is typically below 10%, where pure Python handles many "book-keeping" tasks. Regarding the time usage, a dissection of the communication overhead will be given in Section 4.3.4.

4.3. Measurements

4.3.1. Ping-pong test

First, we compare the actual performance of the C-version MPI and pypar-layered MPI. Two ping-pong test programs are used, namely pytiming and ctiming.c, which are provided in the pypar distribution [23]. Both test programs measure the time usage of exchanging messages of different sizes between two processors. Based on the obtained time measurements, the actual values of latency and bandwidth are estimated using a least squares strategy. Table 4 thus lists the estimated latency and bandwidth values for C-version MPI and pypar-layered MPI. We can observe that there is no difference between C and pypar with respect to the actual bandwidth, which is quite close to the its theoretical peak value. Regarding the latency, it is evident that the extra Python layer of pypar results in larger overhead.

We can also mention that a similarly implemented Python- pyMPI ping-pong test program reports a considerably larger latency value and a significantly lower bandwidth value. This is due to the fact that pyMPI treats array objects as ordinary Python objects. The current implementation of pyMPI (version 2.0a5) is evidently not suitable for high-performance parallel computing yet.

4.3.2. Two-dimensional wave simulations

Two global computational meshes 1000×1000 and 2000×2000 are chosen for the two-dimensional simulations. For each global mesh, the value of P is varied between 1 and 16. Table 5 reports the wall-clock time measurements (in seconds) of the entire while-loop, i.e., the time-stepping part of solving the wave equation. We remark that the number of time steps is determined by the global mesh size, independent of P. The speed-up results use the measurements associated with $P = 1$ (free of communication overhead) as the reference.

We can observe from Table 5 that the wall-clock time consumptions of all three implementations are of the same level. The mixed Python-Fortran implementation is best for small values of P, while the performance relative to the Python-C implementation drops for large values of P.

Comparing the mixed Python-C implementation with the pure C implementation, we can see that the parallel performance of mixed Python-C scales slightly better than that of pure C for these two-dimensional simulations. There are at least two reasons for this somewhat unexpected behavior. First, the Python-C implementation has a better computation/communication ratio, due to the overhead of repeatedly invoking a C function and executing the while-loop in Python. We note that this type of overhead is also present when $P = 1$. Regarding the overhead associated only with communication, we recall that it consists of three parts: preparing outgoing messages, message exchange, and extracting incoming messages, see Section 3.2. The larger latency value of pypar-MPI (than that of C-MPI) thus concerns only the second part, it may have an almost invisible impact on the overall performance. In other words, care should be taken to implement the message preparation and extraction work efficiently, and the index slicing functionality of Python arrays clearly has sufficient efficiency for these purposes; see Section 4.3.4. Second, the better speed-up results of the mixed Python-C implementation may sometimes be attributed to the cache effects. To understand this, we have to realize that any mixed Python-C implementation always uses more memory than its pure C counterpart. Consequently, the mixed Python-C implementation has a better chance of letting bad cache use "spoil" its serial ($P = 1$) performance, and it will thus scale better in some special cases when $P > 1$.

Table 5
A comparison of the two-dimensional performance between three different implementations: mixed Python-Fortran, mixed Python-C, and pure C

	Mixed Python-Fortran		Mixed Python-C		Pure C	
P	Wall-time	Speedup	Wall-time	Speedup	Wall-time	Speedup
	$n_x = n_y = 1000$; 2828 time steps					
1	220.02	N/A	268.62	N/A	229.64	N/A
2	115.56	1.90	146.62	1.83	119.66	1.92
4	90.21	2.44	88.69	3.03	73.76	3.11
8	46.36	4.75	39.83	6.74	43.27	5.31
12	32.71	6.73	26.44	10.16	24.03	9.56
16	26.22	8.39	23.00	11.68	23.39	9.82
	$n_x = n_y = 2000$; 5656 time steps					
1	1727.99	N/A	2137.18	N/A	1835.38	N/A
2	885.41	1.95	1116.41	1.91	941.59	1.95
4	673.98	2.56	649.06	3.29	566.34	3.24
8	347.90	4.97	371.98	5.75	327.98	5.60
12	195.08	8.86	236.41	9.04	227.55	8.07
16	197.23	8.76	193.83	11.03	175.18	10.48

Table 6
A comparison of performance between a mixed Python-C implementation and a pure C implementation for simulations of a three-dimensional wave equation

	Mixed Python-C		Pure C	
P	Wall-time	Speedup	Wall-time	Speedup
	$n_x = n_y = n_z = 100$; 346 time steps			
1	47.75	N/A	50.11	N/A
2	29.62	1.61	32.30	1.55
4	16.47	2.90	16.16	3.10
8	10.59	4.51	9.97	5.03
12	8.49	5.62	8.55	5.86
16	7.31	6.53	7.27	6.89
	$n_x = n_y = n_z = 200$; 692 time steps			
1	735.89	N/A	746.96	N/A
2	426.77	1.72	441.51	1.69
4	259.84	2.83	261.39	2.86
8	146.96	5.01	144.27	5.18
12	112.01	6.57	109.27	6.84
16	94.20	7.81	89.33	8.36

4.3.3. Three-dimensional wave simulations

Similar to the two-dimensional simulations, two global computational meshes $100 \times 100 \times 100$ and $200 \times 200 \times 200$ are used for the three-dimensional simulations. Table 6 reports the corresponding wall-clock time measurements (in seconds).

It can be observed from Table 6 that the wall-clock time consumption associated with the mixed Python-C implementation is again of the same level as the pure C implementation. The exceptionally good serial ($P = 1$) performance of the mixed Python-C implementation is due to the relatively expansive triple indexing used in the pure C implementation (such as `u[i][j][k]` in Example 17) against the single indexing used in the migrated C function (see Example 16). For the serial three-dimensional simulations, this disadvantage of the pure C implementation is clearly larger than Python-C implementation's overhead of repeatedly invoking a C function and executing the `while`-loop in Python. When $P > 1$, the pure C parallel implementation scales better than the mixed Python-C parallel implementation for these three-dimensional cases, unlike the two-dimensional results reported in Table 5. Also as expected, the speed-up results associated with the $200 \times 200 \times 200$ global mesh are better than those associated with the $100 \times 100 \times 100$ mesh. We must remark that the relatively poor speed-up results, for both pure C and mixed Python-C implementations, are due to the relatively small computation/communication ratio in these three-dimensional simulations, plus the use of blocking MPI send/receive routines [10]. The slow communication speed of the Linux cluster also considerably affects the scalability.

4.3.4. Dissection of the communication cost

To understand better the speed-up results obtained in Tables 5 and 6, let us "dissect" the communication overhead of `update_internal_boundaries`, according to the three generic communication-related tasks described in Section 3.2:

– Task 3: preparing the outgoing messages, for which a 3D example of filling an outgoing message can be:

```
upper_x_out_msg = u[nx_loc-1,:,:]
```

We remark that `upper_x_out_msg` is a temporary 2D array providing only a new "view" of the 3D array u. No storage allocation or data copying takes place.

– Task 4: exchanging the messages using pypar, for which a 3D example can be:

```
pypar.send (upper_x_out_msg, destination=upper_x_neighbor_id, \
            bypass=True)
pypar.receive (upper_x_neighbor_id, buffer=x_in_buffer, \
               bypass=True)
```

Note that the array entries that are pointed by `upper_x_out_msg` are not contiguous in memory, the pypar.send command thus needs to *internally* copy the content of `upper_x_out_msg` into a contiguous 1D array before sending it out. This results in larger overhead than sending a message with contiguous storage, such as measured in Section 4.3.1. We also remark that `x_in_buffer` is a pre-allocated 2D array with contiguous storage, which is allocated once and for all by the `prepare_communication` function.

– Task 5: extracting the incoming messages, for which a 3D example of handling an incoming message can be:

```
u[nx_loc,:,:] = x_in_buffer
```

Table 7 shows such a dissection of the communication overhead due to the `update_internal_boundaries` function. The table investigates how the time consumption of each `update_internal_boundaries` call is divided into the three generic tasks, as P varies. As a comparison, we also show the dissection of a corresponding C-MPI implementation of `update_internal_boundaries`. In respect of programming, the Python-pypar implementation is much more compact, because the C-MPI implementation uses explicit for-loops for preparing and extracting the messages. In respect of performance, the C-MPI implementation is understandably better, but not by much. This is due to the efficient array slicing functionality and that the additional overhead of the pypar MPI wrapper is of a limited size (see Table 4). It can be observed in Table 7 that the difference between the C-MPI implementation and the Python- pypar implementation is smaller for the 3D case, because a larger volume of data is exchanged (despite an often larger number of messages). We also remark that the favorable Python- pypar measurements for Task 3 (especially in 3D) are due to the fact that the cost of data copying is hidden in Task 4 of the Python- pypar implementation.

4.3.5. Some comments

It should be noted that non-blocking communication routines can in principle be used for improving such simple parallel applications. For example, in the pure C implementation, routines such as `MPI_Isend` and `MPI_Irecv` may help to overlap communication with computation and thus hide the overhead of communication. At the moment of this writing, the pypar package has not implemented such non-blocking communication routines, whereas pyMPI has non-blocking communication routines but with slow performance. Since the experiments in this paper are meant to provide a precise understanding of the size of the Python-induced communication overhead, we have deliberately sticked to blocking communication routines in our pure C implementations, which give the matching measurements for comparison. Besides, non-blocking communication has limited use in the general case of unstructured computations, so investigating the efficiency of blocking communication routines in Python is important for real-life applications.

Regarding the parallelization of unstructured computations, we believe that the array slicing functionality should also be used for efficiency, instead of writing for-loops in pure Python. The task of work partitioning and load balancing will become more difficult, for which a mixed language implementation is necessary. This is because there already exist several high-quality partitioning software packages in Fortran or C.

Table 7
Dissecting the communication overhead (in seconds) of each update internal boundaries call into three generic tasks

	Python + pypar				C + MPI			
P	Task3	Task4	Task5	Sum	Task3	Task4	Task5	Sum
	`update internal boundaries` in 2D; $n_x = n_y = 2000$							
2	9.49e-5	3.41e-3	1.20e-4	3.63e-3	7.01e-5	3.23e-3	9.50e-5	3.39e-3
4	9.67e-5	4.49e-3	3.64e-4	4.96e-3	1.58e-4	3.69e-3	2.19e-4	4.06e-3
8	7.33e-5	4.67e-3	2.34e-4	4.98e-3	9.42e-4	2.61e-3	6.54e-4	4.20e-3
12	7.67e-5	3.35e-3	3.57e-4	3.78e-3	7.06e-4	2.17e-3	5.25e-4	3.40e-3
16	1.72e-4	3.04e-3	2.53e-4	3.47e-3	3.19e-4	2.35e-3	1.21e-4	2.79e-3
	`update internal boundaries` in 3D; $n_x = n_y = n_z = 200$							
2	4.89e-4	5.86e-2	3.23e-3	6.23e-2	3.82e-3	5.87e-2	3.48e-3	6.60e-2
4	7.73e-4	6.25e-2	3.61e-3	6.69e-2	4.00e-3	5.96e-2	3.10e-3	6.68e-2
8	1.31e-3	5.30e-2	5.73e-3	6.00e-2	1.00e-3	4.84e-2	3.83e-3	5.32e-2
12	1.39e-3	5.65e-2	4.17e-3	6.20e-2	1.45e-3	5.25e-2	3.05e-3	5.70e-2
16	1.30e-3	5.46e-2	1.70e-3	5.66e-2	8.27e-4	5.05e-2	1.43e-3	5.28e-2

5. Concluding remarks

The discussion and analysis presented in Sections 2 and 3, together with the measurements in Section 4.3, have given us reasons to believe that Python is sufficiently efficient for scientific computing. However, "Python" implies in this context the core language together with the Numerical Python package (plus other useful scientific packages) and frequent migration of nested loops to C/C++ and Fortran extension modules. In addition, one must avoid expensive constructs in Python. That is, data structures should use Python arrays and computation-intensive code segments should be implemented and carried out in compiled code, either in a ready-made library such as Numerical Python or in tailored C/C++ or Fortran code. Moreover, the fully comparable measurements of Python-related parallel wave simulations against a pure C implementation also encourage the use of Python in developing parallel applications.

The results obtained in this paper suggest a new way of creating future scientific computing applications. Python, with its clean and simple syntax, its high-level statements, its numerous library modules, its vectorization capabilities, and its bindings to MPI, can be used in large portions of an application where performance is not first priority or when vectorized expressions are sufficient. This will lead to shorter and more flexible code, which is easier to read, maintain, and extend. The parts dealing with nested loops over multi-dimensional arrays can, when necessary, be migrated to a compiled extension module. Our performance tests show that such mixed-language applications may be as efficient as applications written entirely in low level C or Fortran 77.

References

[1] D. Arnold, M.A. Bond, Chilvers and R. Taylor, *Hector: Distributed objects in Python*, In Proceedings of the 4th International Python Conference, 1996.

[2] D. Ascher, P.F. Dubois, K. Hinsen, J. Hugunin and T. Oliphant, *Numerical Python*, Technical report, Lawrence Livermore National Lab., CA, 2001. http://www.pfdubois.com/numpy/numpy.pdf.

[3] D. Beazley, *Python Essential Reference*, (2nd ed.), New Riders Publishing, 2001.

[4] D. Blank, L. Meeden and D. Kumar, *Python Robotics: An environment for exploring robotics beyond LEGOs*, In Proceedings of the 34th SIGCSE technical symposium on Computer Science Education, 2003, pp. 317–321.

[5] Blitz++ software, http://www.oonumerics.org/blitz/.

[6] W.L. Briggs, *A Multigrid Tutorial*. SIAM Books, Philadelphia, 1987.

[7] A.M. Bruaset, *A Survey of Preconditioned Iterative Methods*, Addison-Wesley Pitman, 1995.

[8] D. Beazley et al., *Swig 1.3 Development Documentation*, Technical report, 2004. http://www.swig.org/doc.html.

[9] F2PY software package, http://cens.ioc.ee/projects/f2py2e.

[10] Message Passing Interface Forum, MPI: A message-passing interface standard. *Internat. J. Supercomputer Appl.* **8** (1994), 159–416.

[11] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, (2nd ed.), Addison–Wesley, 2003.

[12] W. Gropp, E. Lusk and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message-Passing Interface*, (2nd ed.), The MIT Press, 1999.

[13] D. Harms and K. McDonald, *The Quick Python Book*, Manning, 1999.

[14] K. Jackson, pyGlobus: A Python interface to the Globus toolkit, *Concurrency and Computation: Practice and Experience* **14** (2002), 1075–1084.
[15] H.P. Langtangen, *Python Scripting for Computational Science*, (Vol. 3), Texts in Computational Science and Engineering, Springer, 2004.
[16] Matlab code vectorization guide, http://www.mathworks.com/support/tech-notes/1100/1109.html.
[17] S. Montanaro, *Python Performance Tips*, http://manatee.mojam.com/∼skip/python/fastpython.html.
[18] Numerical Python software package, http://sourceforge.net/projects/numpy.
[19] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.
[20] R. Plösch, *Design by contract for Python*, In Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference, IEEE Computer Society, 1997, pp. 213–219.
[21] Psyco home page, http://psyco.sourceforge.net/.
[22] PyMPI software package, http://sourceforge.net/projects/pympi.
[23] PyPar software package, http://datamining.anu.edu.au/∼ole/pypar.
[24] Python optimization tips, http://trific.ath.cx/resources/python/optimization/.
[25] P. Ramachandran, Performance of various Python implementations for solving the 2D Laplace equation, http://www.scipy.org/documentation/weave/weaveperformance.html.
[26] C. Ramu and C. Gemuend, *cgimodel: CGI programming made easy with Python*, Linux Journal, 2000.
[27] M.F. Sanner, Python: A programming language for software integration and development, *J. Mol. Graph Model.* **17**(1) (1999), 57–61.
[28] SciPy software package, http://www.scipy.org.
[29] Software for running the computational experiments in the present paper, http://folk.uio.no/xingca/python/efficiency/.
[30] G. van Rossum and F.L. Drake, Extending and Embedding the Python Interpreter, http://docs.python.org/ext/ext.html.
[31] G. van Rossum and F.L. Drake, Python Tutorial, http://docs.python.org/tut/tut.html.
[32] T.L. Veldhuizen, Blitz++: The library that thinks it is a compiler, in: *Advances in Software Tools for Scientific Computing*, (Vol. 10), H.P. Langtangen, A.M. Bruaset and E. Quak, eds, Lecture Notes in Computational Science and Engineering, Springer, 2000, pp. 57–87.
[33] A. Watters, J.C. Ahlstrom and G. van Rossum, *Internet Programming with Python*, Henry Holt and Co., Inc., 1996.
[34] Weave, http://www.scipy.org/documentation/weave, Part of the SciPy package.
[35] B. Wilkinson and M. Allen, *Parallel Programming – Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 1999.
[36] T.-Y.B. Yang, G. Furnish and P.F. Dubois, *Steering object-oriented scientific computations*, In Proceedings of Technology of Object-Oriented Languages and Systems, 1997, pp. 112–119.