Performance and Productivity of Parallel Python Programming — A study with a CFD Test Case *

Achim Basermann, Melven Röhrig-Zöllner and Joachim Illmer
German Aerospace Center (DLR)
Simulation and Software Technology
Linder Höhe, Cologne, Germany
{Achim.Basermann, Melven.Roehrig-Zoellner}@dlr.de; Joachim.Illmer@gmail.com

ABSTRACT

The programming language Python is widely used to create rapidly compact software. However, compared to low-level programming languages like C or Fortran low performance is preventing its use for HPC applications. Efficient parallel programming of multi-core systems and graphic cards is generally a complex task. Python with add-ons might provide a simple approach to program those systems. This paper evaluates the performance of Python implementations with different libraries and compares it to implementations in C or Fortran. As a test case from the field of computational fluid dynamics (CFD) a part of a rotor simulation code was selected. Fortran versions of this code were available for use on single-core, multi-core and graphic-card systems. For all these computer systems, multiple compact versions of the code were implemented in Python with different libraries. For performance analysis of the rotor simulation kernel, a performance model was developed. This model was then employed to assess the performance reached with the different implementations. Performance tests showed that an implementation with Python syntax is six times slower than Fortran on single-core systems. The performance on multi-core systems and graphic cards is about a tenth of the Fortran implementations. A higher performance was achieved by a hybrid implementation in C and Python using Cython. The latter reached about half of the performance of the Fortran implementation.

PyHPC2015, November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-4010-6/15/11. . . $15.00
DOI: http://dx.doi.org/10.1145/2835857.2835859

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.2 [**Programming languages**]: Language Classifications—*Very high-level languages*

General Terms

Performance

1. INTRODUCTION

The programming language Python is widely used nowadays. In the field of scientific software Python allows simple programming of prototypes similar to the interactive platform MATLAB of MathWorks [14].

In the list of the 500 fastest computer systems of the world from June 2015 we find 90 systems which exploit GPU (**G**raphics **P**rocessing **U**nit) or Xeon Phi accelerators [28]; four of them are among the top ten. Since multi-core architectures and GPUs with high numbers of processing elements are available for everyone today the development of parallel programs with high performance becomes increasingly more important. Efficient parallel programming for multi-core architectures and GPUs is generally difficult. It requires a high percentage of parallelism in the problem to be solved, good load balance and minimal data exchange between parallel processes, efficient process synchronization as well as appropriate performance modelling and testing.

In fluid engineering and material science, e.g., computer simulations replace physical experiments in an increasing extent. Simulations make it possible to check theories which can not be tested experimentally since appropriate experiments are too expensive, too difficult, too slow or too dangerous or since the quantity of interest can not be measured. Simulation codes must exploit modern computer hardware efficiently in order to achieve short execution times.

For efficient parallel programming, we desire a programming language which is productive and allows high performance. Productivity requires a high abstraction level which usually involves a high overhead which degrades performance [29]. Low-level programming languages like C or Fortran, on the other hand, allow machine-oriented programming and thus make highly efficient codes possible, but code development is time-consuming and error-prone. Since performance is the main criterion in the HPC (**H**igh **P**erformance **C**omputing)

area low-level languages are widely used here. Python offers a high abstraction level, but only achieves low performance without additional libraries. Appropriate additional libraries facilitate acceleration of performance critical parts of Python code distinctly without globally loosing a high level software design.

In this paper we discuss parallel programming of numerical algorithms from computational fluid dynamics (CFD) with Python and add-on libraries. Our objective is to achieve a performance of Python implementations close to comparable C or Fortran implementations on modern computer hardware with multi-core processors and GPU accelerators. For this investigation we use a complex application kernel from rotor simulation which can not exploit existing fast building blocks such as BLAS routines [3].

2. BACKGROUND OF THE ROTOR SIMULATION TEST CASE

The code "Freewake" is part of the rotor simulation system S4 of the department "Rotorcraft" of DLR's Institute of Flight Systems. Freewake simulates three-dimensional flows around an actively controlled rotor of a helicopter [2].

Freewake was developed in Fortran between 1994 and 1996. The principal method bases on experimental data from the HART program [27]. Color and thickness of the pipes illustrated in Figure 1 represent strength and diameter of vortices in the wake of the rotor.

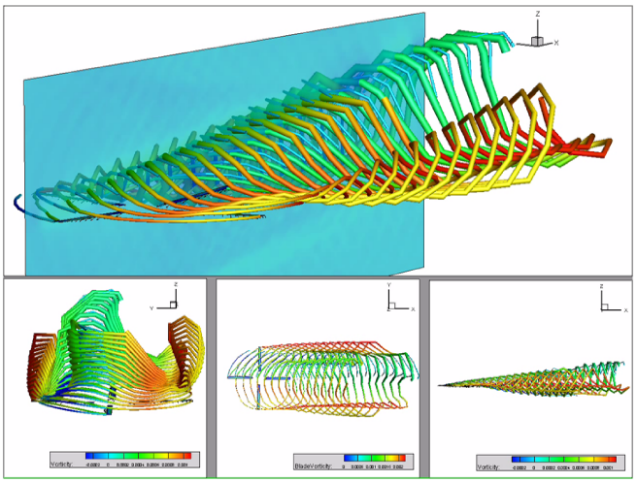

Figure 1: Visualization of the modeled vortex movement behind a helicopter rotor in fast forward flight.

During the forward motion of a helicopter a very non-symmetric flow behavior is generated. The flow behavior causes some of the main problems of rotorcrafts: high loundness, strong vibrations, high energy consumption and flight stability problems. The characteristic noises of a helicopter arise when a rotor blade meets an air vortex [4]. Opposite to traditional CFD simulations, which compute the flow equation for each cell close to the rotor, Freewake applies a two-dimensional grid in a three-dimensional space in order to represent vortices. In each time step, the induced velocity of all vortex elements on all grid nodes is computed by the Biot-Savart law [1]. Depending on the distance between vortex element and grid node different formulae are used.

Originally, Freewake was developed for massively parallel systems with distributed memory. Communication between parallel processes is realized by MPI (**M**essage **P**assing **I**nterface) [15]. In the last years, Freewake was adapted to multi-core CPU and GPU exploitation. For performance investigations, this paper uses a Freewake benchmark kernel which determines the induced velocities with a simplified formula.

3. PYTHON LIBRARIES FOR SOLVING SCIENTIFIC PROBLEMS

This section introduces the Python project Cython as well as the Python libraries NumPy, Numba and "Python Bindings for Global Array Toolkit" and discusses their appropriateness for efficient parallel programming.

3.1 Cython

The open source project Cython provides a Python code compiler and an extended Python programming language [7]. The language extensions allow calls to C functions and the use of C variable types and attributes. Python code is translated into C code and compiled by a C compiler [6]. Since the generated C code does not contain Python calls any more Cython circumvents the problem with Python's GIL (**G**lobal **I**nterpreter **L**ock)[1] and the compiled code can be executed fully in parallel. For shared memory parallelization, Cython offers simple use of OpenMP [26].

3.2 NumPy

The Python library NumPy supports efficient use of large N-dimensional arrays and provides routines for solving linear algebra problems and for applying Fourier transforms, e.g., as well as tools for integration of C, C++ and Fortran code [21]. The combination of Python with NumPy allows similarly comfortable programming as with Matlab. As a rule, execution times of NumPy functions are distinctly faster than standard Python functions since NumPy exploits C implementations for compute intensive methods.

3.3 Numba

Numba is an open source compiling environment for Python and NumPy which generates optimized machine code [17]. As illustrated in Figure 2, Numba applies the LLVM compiler for machine code generation [16]. Numba supports OpenMP, OpenCL [25] and CUDA [22].

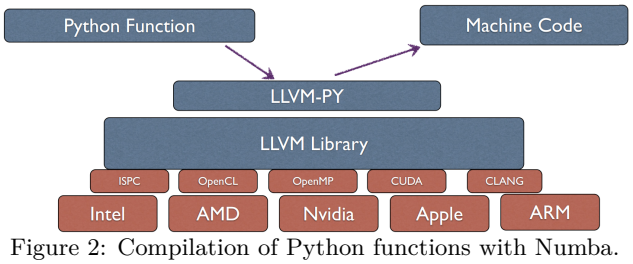

Figure 2: Compilation of Python functions with Numba.

For optimized machine code generation, Numba follows the strategy to process the original Python code as possible without changes. Nevertheless, code annotations may support the optimization process. Annotations, e.g., allow selective

[1]GIL inhibits really parallel execution of Python threads.

compilation of Python functions with Numba and specify target hardware like single- or multi-core CPUs.

NumbaPro is a commercial extension of Numba developed by Continuum Analytics [20]. Compared with Numba, NumbaPro offers additional optimization features for single- and multi-core CPUs as well as via the CUDA interface for NVIDIA GPUs. NumbaPro is able to exploit Intel's "**M**ath **K**ernel **L**ibrary" (MKL) so that BLAS operations with NumPy arrays are executed in a highly optimized parallel way [11].

3.4 Python Bindings for Global Array Toolkit

The Python Bindings for Global Array Toolkit allows access to physically distributed data via a shared memory interface [8]. A distributed array can be handled like an array in a shared memory system. The Global Arrays represent an extension of MPI for message exchange between parallel processors. In the same program, the code developer may switch between the shared memory interface of the Global Arrays and the distributed memory interface of MPI.

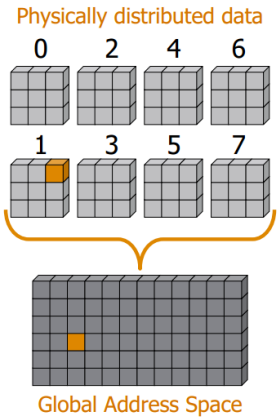

Figure 3: Data access for Global Arrays

Figure 3 illustrates the access to an array distributed to several processors via the Global Array interface. Through the global address space, the marked element can simply be read by *ga.get(a,(3,2))*. With MPI a message would have to be explicitly exchanged between processors 1 and 0 if the marked element has to be used on processor 0.

4. PYTHON IMPLEMENTATIONS OF THE ROTOR SIMULATION KERNEL

This section describes the implementation of the Freewake benchmark kernel with standard Python, Cython, NumPy, Numba and Python Bindings for Global Array Toolkit.

In the Freewake benchmark, the time is measured for simulating the velocities induced by vortices for a rotor rotation of 1 degree. A grid over the rotor discretizes the vortices. For each grid node the induced velocities are computed. The required data per grid node are stored in a 4-dimensional array. The velocity is composed of the induced velocities of the longitudinal and lateral vorticies. Thus the runtime of the algorithm increases quadratically with the number of grid nodes. There are two possibilities for computing the induced velocities: The computational function within one outer loop has an array parameter and operations are performed element-wise on this array, or we add three further loops into the outer loop and call the function in the most inner loop with single value parameters.

4.1 Implementation using the Python Standard Library

Calculations with Python lists are not possible. The reason is that a list can consist of elements with different data types. The interpreter can then not determine which operation shall be performed. However, it is possible to implement the required operations for lists for which the data type of the elements is known. For the Freewake benchmark kernel, computations with whole lists are slower in Python than adding three inner loops and calling the calculation function with single values. This consideration results in the loop structure in the kernel from listing 1.

```
for iblades in range(numberOfBlades):
    for iradial in range(1, dimensionInRadialDirection):
        for iazimutal in range(dimensionInAzimualDirectionTotal):
            for i1 in range(len(vx[0])):
                for i2 in range(len(vx[0][0])):
                    for i3 in range(len(vx[0][0][0])):
                        #wilin call 1
for iblades in range(numberOfBlades):
    for iradial in range(dimensionInRadialDirection):
        for iazimutal in range(1,
            ↪ dimensionInAzimualDirectionTotal):
            for i1 in range(len(vx[0])):
                for i2 in range(len(vx[0][0])):
                    for i3 in range(len(vx[0][0][0])):
                        #wilin call 2
for iDir in range(3):
    for i in range(numberOfBlades):
        for j in range(dimensionInRadialDirection):
            for k in range(dimensionInAzimualDirectionTotal):
                x[iDir][i][j][k] = x[iDir][i][j][k] + dt * vx[iDir
                    ↪ ][i][j][k]
```

Listing 1: Call of the function *wilin* in Python.

In the first seven lines the induced velocities of the lateral vortices and in lines eight to 14 the induced velocities of the longitudinal vortices are computed. After that, the grid nodes are updated.

```
def wilin(dax, day, daz, dex, dey, dez, ga, ge, wl, vx, vy, vz):
    rcq = 0.1
    daq = dax**2 + day**2 + daz**2
    deq = dex**2 + dey**2 + dez**2
    da = math.sqrt(daq)
    de = math.sqrt(deq)
    dae = dax * dex + day * dey + daz * dez
    sqa = daq - dae
    sqe = deq - dae
    sq = sqa + sqe
    rmq = daq * deq - dae**2
    fak = ((da + de) * (da * de - dae) * (ga * sqe + ge * sqa) +
        ↪  (ga - ge) * (da - de) * rmq) / (sq * (da * de + eps
        ↪ ) * (rmq + rcq * sq))
    fak = fak * wl / math.sqrt( sq )
    vx = vx + fak * (day * dez - daz * dey)
    vy = vy + fak * (daz * dex - dax * dez)
    vz = vz + fak * (dax * dey - day * dex)
    return vx, vy, vz
```

Listing 2: Function for the velocity induction in Python.

When first called the function *wilin* from listing 2 has the parameters:

- x[:][i1][i2][i3] - x[:][iblades][iradial][iazimutal]
- x[:][i1][i2][i3] - x[:][iblades][iradial - 1][iazimutal]
- transversalVorticity[iblades][iradial][iazimutal]
- transversalVorticity[iblades][iradial - 1][iazimutal]
- transversalVortexLength[numberOfBlades-1] [dimensionInRadialDirection-1][iazimutal]
- vx[:][i1][i2][i3]

The return values of this function are stored in vx[:][i1][i2][i3]. In the second call the parameters two and four of *wilin* have incremented or decremented indices *iradial* or *iazimutal*, respectively. Listing 2 shows the function *wilin* which computes the induced velocities of the vortex segments.

4.2 Implementation using Cython

Opposite to all other implementations in this paper, Cython requires variables with types:

```
cdef <Variablentyp> <Variablenname>
```

In the Cython implementation of the Freewake kernel, all arrays are initialized as NumPy arrays (cf. section 4.3) and then immediately transformed into Cython memory views. The latter provide efficient access to NumPy arrays, without Python overhead. For further optimization of the kernel in Cython, the order of the loops was changed, the return values were stored in a "struct" and the loop operations parallelized. The loops with the indices *iblades*, *iradial* and *iazimutal* from listing 1 were shifted behind the loop with index *i3* in order to make parallel execution of the loops possible. This is necessary since the velocities are added together in *vx* over the loops with indices *iblades*, *iradial* and *iazimutal*. The return values of the function from listing 5 are written into a struct. Subsequently, these values have to be stored into associated positions of the array. With these changes the loops do not contain any Python objects any more so that the GIL can be circumvented with the statement *nogil* and loop parallelization can be realized with the function *prange*. The latter parallelizes loops with OpenMP. Best parallelization results are achieved if the *prange* loop is the outermost loop. This avoids overhead through too frequent thread generation and termination for *prange* loop calls. The loop with index *i1* has only four cycles in the test kernel. Since the CPU of our computer system has six cores optimal performance is not possible if this loop is parallelized. The loop with index *i2*, however, has 12 cycles and achieves best performance here. Parallelization of several loops is possible, but does not give better performance due to distinctly increased overhead for thread generation and termination. Interchange of the loops with indices *i1* and *i2* gives a further performance increase by 20% so that the Cython kernel gets the loop order in listing 3. For parallel execution of the Cython code using OpenMP, the Cython compiler and linker require the argument *-fopenmp*.

For the parallelization as described above we need to modify the *wilin* function so that it does not use any Python objects. In order to find all statements which generate access to Python objects Cython offers the option *annotate*.

```
for i2 in prange(dimensionInRadialDirection):
    for i1 in xrange(numberOfBlades):
        for i3 in xrange(dimensionInAzimualDirectionTotal):
            for iblades in xrange(numberOfBlades):
                for iradial in xrange(1,
                    ↪ dimensionInRadialDirection):
                    for iazimutal in xrange(
                        ↪ dimensionInAzimualDirectionTotal):
```

Listing 3: Optimized loop order in Cython.

This option also provides information about the execution time per code line A further optimization option is *Cdivision(True)* which deactivates Python tests for divisions. Python tests may slow down code execution by about 35 %. With *boundscheck(False)* Cython avoids array boundary checks which further accelerates the code. As C Cython can not return several function values. This problem was circumvented by the struct *myret* from listing 4.

```
cdef struct myret:
    double vz1
    double vx1
    double vy1
```

Listing 4: Struct with three doubles as return type in Cython.

The Cython function *wilin* possesses an input parameter of type *myret* and has the return type *myret*. Listing 5 displays the Cython function *wilin* in detail. The square root computation in lines 7, 8 and 15 is performed by C routines through the import of *libc.math*.

```
@cython.cdivision(True)
@cython.boundscheck(False)
cdef myret wilin_mem(double dax, double day, double daz, double
    ↪ dex, double dey, double dez, double ga, double ge,
    ↪ double wl, double vx1, double vy1, double vz1, myret
    ↪ ret)nogil:
    cdef double rcq = 0.1
    cdef double daq = dax**2 + day**2 + daz**2
    cdef double deq = dex**2 + dey**2 + dez**2
    cdef double da = sqrt(daq)
    cdef double de = sqrt(deq)
    cdef double dae = dax*dex + day*dey + daz*dez
    cdef double sqa = daq - dae
    cdef double sqe = deq - dae
    cdef double sq = sqa + sqe
    cdef double rmq = daq * deq - dae**2
    cdef double fak = ((da + de) * (da*de - dae) * (ga*sqe + ge*
        ↪ sqa) + (ga - ge) * (da - de) * rmq) / (sq * (da*de +
        ↪  eps) * (rmq + rcq*sq))
    cdef double fak2 = fak * wl / sqrt(sq)
    ret.vx1 = vx1 + fak2 * (day*dez - daz*dey)
    ret.vy1 = vy1 + fak2 * (daz*dex - dax*dez)
    ret.vz1 = vz1 + fak2 * (dax*dey - day*dex)
    return ret
```

Listing 5: Function for the velocity induction in Cython.

4.3 Implementation using NumPy

NumPy provides fast array operations for Python. NumPy arrays can be used rather than standard Python lists in a Freewake kernel implementation. Listing 6 shows how a NumPy array is generated. "shape" defines dimension and size of the array, "dtype" the data type of the array elements.

```
import numpy as np
x = np.ones(shape = (dim1, dim2, dim3, dim4), dtype = np.float64
    ↪ )
```

Listing 6: Generation of a NumPy array.

Operations on NumPy arrays exploit fast C routines. Good performance for our kernel implementation can be expected if the function *wilin* obtains array parameters. Listing 7 shows how *wilin* is called for the exploitation of NumPy arrays.

In the first four lines, the induced velocities of lateral vortices are computed, in lines five to eight the induced velocities of longitudinal vortices.

```
for iblades in xrange(numberOfBlades):
    for iradial in xrange(1, dimensionInRadialDirection):
        for iazimutal in xrange(dimensionInAzimualDirectionTotal)
            ↪ :
            #wilin call 1
for iblades in xrange(numberOfBlades):
    for iradial in xrange(dimensionInRadialDirection):
        for iazimutal in xrange(1,
            ↪ dimensionInAzimualDirectionTotal):
            #wilin call 2
for iDir in range(3):
    for i in range(numberOfBlades):
        for j in range(dimensionInRadialDirection):
            for k in range(dimensionInAzimualDirectionTotal):
                x[iDir][i][j][k] = x[iDir][i][j][k] + dt * vx[iDir
                    ↪ ][i][j][k]
```

Listing 7: Call of the *wilin* function with NumPy.

Afterwards, all grid nodes are updated. This changes the parameters for *wilin*.
x[0][i1][i2][i3] - x[0][iblades][iradial][iazimutal] from listing 2 becomes *x[0] - x[0, iblades, iradial, iazimutal]*. The other parameters change accordingly. The operations +, *, - and / are now element-wise operations with NumPy arrays. Each element of one array is added to, multiplied with, subtracted from or divided by each element at the same position in another array.

wilin in the NumPy implementation looks like the standard Python function from listing 2. Just the NumPy square root function *numpy.sqrt* replaces *math.sqrt* in listing 2.

4.4 Implementation using Numba

With Numba we discuss three optimized kernel versions for a single core of a CPU, for multi-core CPUs and for GPUs, respectively. All three versions considered base on the NumPy implementation.

4.4.1 Single-Core Optimization

Numba tries to execute as much code as possible in the accelerated "nopython" mode. The generation of NumPy arrays is not possible in this mode. The Numba single-core variant annotates the function from listing 2 and the function which includes the loops from listing 7 with *autojit*. This annotation makes Numba compile these functions to machine code. For calls of functions annotated with *autojit*, Numba is able to automatically recognize which data and data types are handed over.

4.4.2 Multi-Core Optimization

Until version 0.11 Numba included the command "prange". This command made loop parallelization possible in a similar way to Cython with OpenMP. An own thread was started for each loop cycle, and the threads were distributed to all availabe CPU cores. From version 0.12 "prange" could not be used any more due stability and performance problems as a Numba developer stated [19].

The "parallel" backend of the *Vectorize* function is another possibility to parallelize functions. The same operation is performed in parallel for all elements of a list [18]. However, the Freewake kernel considered can not be vectorized in this way since a function vectorized with Numba can only possess one return value and since the summation of all velocities is not possible in such a function. Thus multi-core optimization could not be performed with Numba at the time of this investigation.

4.4.3 GPU Optimization

With the CUDA JIT annotation, Numba offers a machine-oriented CUDA entry point. A function annotated in this way is executed on the GPU of a computer system. Such a Freewake kernel implementation for GPUs requires the additional parameters *blockdim* and *griddim*. *griddim* indicates how many thread blocks shall be started in a grid structure; *blockdim* specifies the number of threads to be started per thread block. Figure 4 displays the thread distribution on a GPU. Before GPU computations can be performed data from

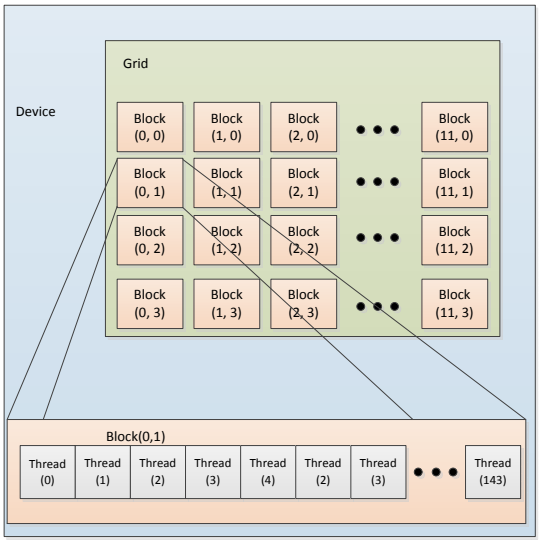

Figure 4: Distribution in blocks and Threads on a GPU.

the main memory of the CPU has to be transmitted into the GPU memory. The command *cuda.to_device(vx)* tranfers the variable *vx* into the memory of the GPU. Subsequently, the GPU can access and operate on these data. As soon as the GPU computations are finished the result data can be transferred back to CPU main memory by *vx.to_host()*. Figure 5 displays the data transfer between CPU and GPU.

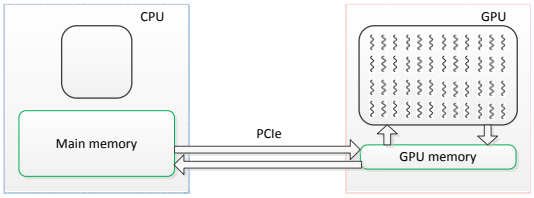

Figure 5: Data transfer between CPU and GPU.

```
i1 = cuda.blockIdx.x
i2 = cuda.blockIdx.y
i3 = cuda.threadIdx.x
```

Listing 8: Identification of CUDA threads.

For each cycle of the loops with indices *i1* to *i3*, a thread is generated with blocks of dimension *blockdim* and a block grid of dimension *griddim* as illustrated in Figure 4. Listing 8 shows the identification of the threads in our Freewake kernel implementation. After completion of all cycles of the *iblades* loop the threads are synchronized with *cuda.syncthreads()*. This ensures that all computations have been completed before result data are transferred to the CPU.

4.5 Parallel Implementation using "Python Bindings for Global Array Toolkit"

The Freewake kernel implementation using Global Arrays as well bases on the NumPy implementation. The function *wilin* uses the formulation from listing 2. During array initialization, Global Arrays of same dimension and type as the corresponding NumPy arrays must be generated additionally. Listing 9 shows an example.

```
g_x = ga.create(ga.C_DBL, [dim1, dim2, dim3, dim4])
```

Listing 9: Generation of an Global Array.

This implementation marks Global Arrays with *g_* and local partial arrays with *l_*. Computations with local variables are performed with NumPy arrays. At the end of the initialization the data from the NumPy arrays are written into Global Arrays as shown in listing 10.

```
ga.put(g_x, l_x)
```

Listing 10: Writing into an Global Array.

This implementation has the disadvantage that some arrays are duplicated on each compute node. This increases the memory requirements. The essential Global Array functions *ga.nodeid()* and *ga.nnodes()* determine the current compute node ID and the number of compute nodes, respectively. The functions permit execution of commands on certain nodes only, e.g, timings or outputs. The Global Array Toolkit automatically cares for even distribution of the Global Arrays to the compute nodes if the distribution is not specified particularly. The scope of a Global Array administrated on a compute node can be requested as shown in listing 11.

```
lo,hi = ga.distribution(g_x, node)
```

Listing 11: Determination of the scope of a Global Array administrated on a certain compute node.

Synchronization of all processes as illustrated in Figure 6 is possible with the following Global Array Toolkit function:

```
ga.sync()
```

Listing 12: Synchronization of all processes.

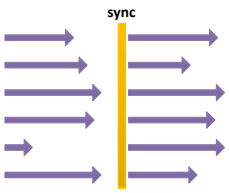

Figure 6: Process synchronization.

This function ensures finalization of all actions on Global Arrays.

The parallelization idea in this implementation is that each process computes the induced velocities only for partial arrays as illustrated in Figure 7 and finally writes its partial result into the Global Array. A partitioning of the problem into smaller partial problems is easily possible for the Freewake kernel since the partial problems are independent from each other. With the command

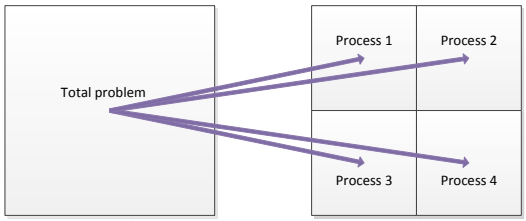

Figure 7: Distribution scheme of a problem in parallel partial problems.

```
l_x = ga.get(g_x,lo,hi)
```

the locally available Global Array elements are loaded into a local variable. These elements are then locally processed and the results finally written from the local array into the associated positions in the Global Array:

```
ga.put(g_x, l_x, lo,hi)
```

The function *wilin* is similarly called as in the NumPy implementation. Only the first array from the parameter *x[0] - x[0, iblades, iradial, iazimutal]*, e.g., is replaced by the local array *l_x*. The second array refers to the Global Array *g_x*. For six cores, the program is started via the MPI framework as follows:

```
mpirun -np 6 ./globalArray.py
```

5. PERFORMANCE ANALYSIS OF THE ROTOR SIMULATION KERNEL

5.1 Environment for Performance Tests

The CPU of the computer system used for performance tests is an Intel Xeon E5645 [9]. The CPU possesses six cores with a clock rate of 2.4 GHz [12] and a shared cache of 12288 KB. The theoretical single precision (SP) peak performance is 57.6 GFLOPS, the double precision (DP) peak performance 28.8 GFLOPS. The measured memory bandwidth for a stream triad benchmark [2] amounts to 19 GB/s.

[2] Addition of two vectors and multiplication of a scalar with another vector.

The system includes an NVIDIA Tesla C2075 GPU. Its 448 CUDA cores are clocked with 1150 MHz. The memory bandwidth is 144 GB/s [23]. The SP peak performance amounts to 1030 GFLOPS, the DP peak performance 515 GFLOPS.

5.2 Roofline Performance Model

The roofline model offers a simple possibility to model performance [30]. It provides the upper performance boundary for an algorithm depending on the ratio of data transfer volume to number of operations.

In the roofline model, this ratio is denoted "balance". The balance of a machine is the ratio of memory bandwidth to CPU peak performance:

$$B_m = \frac{\text{memory bandwidth [GBytes/s]}}{\text{peak performance [GFLOP/s]}}. \quad (1)$$

The code balance indicates the requirements of an algorithm. It computes as the ratio of data tranfer volume to number of floating points operations in the algorithm:

$$B_c = \frac{\text{data traffic [Bytes]}}{\text{floating point operations [FLOP]}}. \quad (2)$$

The data traffic value of an algorithm refers to the limiting data transport path. The ratio of machine to code balance gives the maximally achievable factor of the theoretical peak performance of a CPU or GPU for an algorithm [10]:

$$l = \min\left(1, \frac{B_m}{B_c}\right) \quad (3)$$

Thus the maximum performance of our Freewake kernel amounts to peak performance $\cdot\, l$.

5.3 Machine Balance for CPU and GPU

The machine balance is different for CPU and GPU. All our implementations apply DP arithmetic. Thus the DP peak performance has to be used for machine balance determination.

For the computer system used, the CPU machine balance amounts to

$$B_m = \frac{19\,\text{[GByte/s]}}{28.8\,\text{[GFLOP/s]}} = 0.6597\,. \quad (4)$$

All codes with a code balance larger than 0.6597 are bandwidth limited on this CPU. This means that data can not be fast enough transfered to the CPU to perform calculations continously.

The GPU machine balance of the test system gives

$$B_m = \frac{144\,\text{[GByte/s]}}{515\,\text{[GFLOP/s]}} = 0.2796\,. \quad (5)$$

5.4 Performance Modelling for the Rotor Simulation Kernel

The Freewake kernel essentially stores its data in six arrays. Two of these arrays have a size of $3 \cdot 4 \cdot 12 \cdot 144$ in our test case, the remaining four a size of $4 \cdot 12 \cdot 144$. The size of additional data is not relevant for the following considerations. Thus the kernel uses a data volume of about $2 \cdot (3 \cdot 4 \cdot 12 \cdot 144) \cdot 8\,\text{Bytes} + 4 \cdot (4 \cdot 12 \cdot 144) \cdot 8\,\text{Bytes} = 552960\,\text{Bytes}$.

The function *wilin* consists of circa 70 operations which are performed in each loop cycle. The number of loop cycles nLoops depends on the four variables numberOfBlades (nB), dimensionInRadialDirection (dR), dimensionInAzimutalDirection (dA) and numberOfTurns (nT) and can be determined with (6).

$$\begin{aligned}\text{nLoops} &= \text{nB}^2 \cdot \text{dR} \cdot \text{dA} \cdot \text{nT} \cdot ((\text{dR}-1) \cdot \text{dA} \cdot \text{nT} \\ &\quad + \text{dR} \cdot (\text{dA} \cdot \text{nT} - 1)) \end{aligned} \quad (6)$$

Insertion of the values used gives $4^2 \cdot 12 \cdot 36 \cdot 4 \cdot \big((12-1) \cdot 36 \cdot 4 + 12 \cdot (36 \cdot 4 - 1)\big) = 91,238,400$ loop cycles. 91,238,400 calls of function *wilin* with circa 70 operations per call give about 6.5 billion executed operations in total.

With these values, the code balance of the Freewake kernel can be determined according to (7).

$$B_c = \frac{550,000[Bytes]}{6,500,000,000[FLOP]} = 0.00008\frac{Bytes}{FLOP}. \quad (7)$$

With the code balance from (7) and the machine balances for CPU from (4) and for GPU from (5), we obtain the performance factors of the kernel for CPU and GPU with (8) and (9), respectively.

$$l_{CPU} = \min\left(1, \frac{0.6597}{0.00008}\right) = 1 \quad (8)$$

$$l_{GPU} = \min\left(1, \frac{0.2796}{0.00008}\right) = 1 \quad (9)$$

This means the Freewake kernel is able to exploit the computational peak performance of the CPU or the GPU since its performance is neither limited by the memory bandwidth of the CPU nor by that of the GPU.

With the number of kernel operations and the peak performance values of CPU and GPU, (10) and (11) determine a minimal runtime of 0.23 s on the CPU and of 0.0126 s on the GPU, respectively.

$$T_{CPU} = \frac{6,500,000,000}{28,800,000,000\,\frac{1}{s}} = 0.23s \quad (10)$$

$$T_{GPU} = \frac{6,500,000,000}{515,000,000,000\,\frac{1}{s}} = 0.0126s \quad (11)$$

A serial implementation reduces the CPU performance by a sixth. This gives a minimal serial kernel runtime of 1.35 s according to 12.

$$T_{CPUseriell} = \frac{6,500,000,000}{4,800,000,000\,\frac{1}{s}} = 1.35s \quad (12)$$

6. PERFORMANCE COMPARISON OF THE IMPLEMENTATIONS

In this section, we discuss the performance of the Python implementations developed in comparison with the reference implementations in Fortran. We examine the performance on a single core, on the full multi-core CPU and on the GPU of the test hardware.

6.1 Single-Core Performance

Without additional libraries our standard Python implementation of the Freewake benchmark takes 638 s on a single core

Implementation	Time
Python	638s
NumPy	12.99s
Numba	22.62s
Cython	5.50s
Fortran	2.38s

Table 1: Results of the single-core implementations.

(cf. Table 1). NumPy use accelerates the benchmark run to 12.99 s by execution of efficient C routines in the background. LLVM compilation of the benchmark with Numba generates an overhead so that the runtime increases to 22.62 s compared with that of the pure NumPy variant. The Cython implementation is with 5.5 s circa half as fast as the optimized Fortran implementation which takes 2.38 s. Figure 8 displays the GFLOPS rates of all single-core implementations.

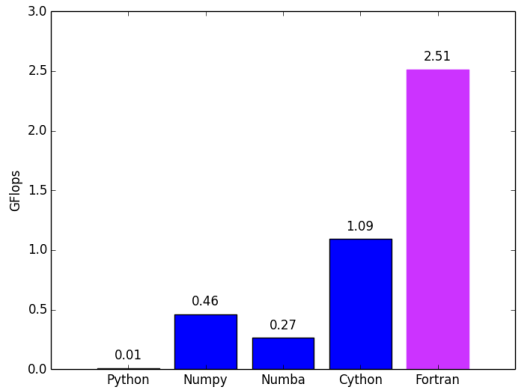

Figure 8: GFLOPS of the serial implementations.

Also the Fortran implementation does not come close to the minimally possible serial time of 1.35 s from (12). The reason is that the simple performance model in section 5 assumes that only multiplications and additions are executed in parallel. The Freewake kernel, however, does include square root operations and divisions in addition. A more accurate prediction would require a further refined model.

An analysis of the Cython run with the performance analysis tool *Likwid* [13] gives that SIMD operations (**S**ingle **I**nstruction **M**ultiple **D**ata) are not exploited. This explains why the Cython version only reaches half of the performance of the Fortran version. The NumPy version exploits SIMD operations, but generates temporary arrays during *wilin* function calls. This significantly increases the data volume so that the cache behavior is negatively effected. The latter causes a significant performance loss. The *JIT* conversion of Numba is a black box so that the developer can not track which operations are executed. A *Likwid* analysis of a run of the Numba version indicates the execution of scalar operations beside SIMD operations. This partially explains performance losses compared with the NumPy version. The standard Python implementation applies dynamic data types which reduce performance drastically.

Implementation	Time	Speed-Up
Cython	1.03s	5.3
Global Arrays	4.34s	3.7
Fortran	0.440s	5.4

Table 2: Results of the multi-core implementations.

Implementation	Time
Numba	0.770s
Fortran + OpenACC	0.086s

Table 3: Results of the GPU implementations.

6.2 Multi-Core Performance

On the multi-core CPU with six cores, a run of our Cython implementation parallelized with OpenMP took 1.03 s. A run of the version exploiting Global Arrays was with 4.34 s circa four times slower. The Fortran kernel is the fastest variant with 0.44 s (cf. Table 2). As in the serial case, the parallel Fortran implementation is circa half as fast as the performance model predicted in (10). For the Cython implementation, OpenMP parallelization results in a performance increase from 1.09 GFLOPS on one core to 5,78 GFLOPS on six cores (cf. Figure 8 and Figure 9). The performance on six cores is with 1.38 GFLOPS distinctly lower for the Global Array version compared with the Cython version. The Global Array version, however, maintains the clear syntax of Python while the Cython version essentially shows C style syntax. The third column of Table 2 lists speed-ups of the runs on

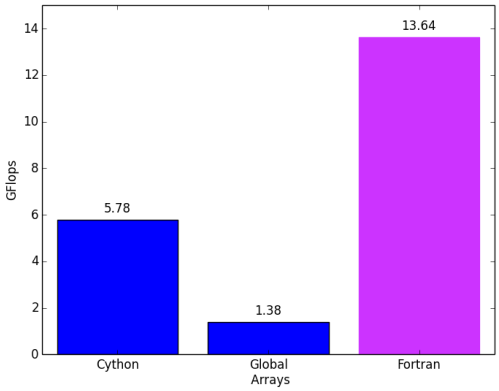

Figure 9: GFLOPS of the parallel implementations.

six cores compared with single-core runs. While speed-ups for the Cython and Fortran version are with 5.3 and 5.4 close to the maximum of six, the Global Array version only achieves a speed-up of 3.7. Main reasons for this are data duplication and the communication between local arrays and Global Array (cf. section 4.5).

6.3 GPU Performance

The maximum number of independent threads for execution of the Freewake kernel on the GPU of the test hardware is given by the product of the values of the variables numberOfBlades (nB), dimensionInRadialDirection (dR), dimensionInAzimutalDirection (dA) and numberOfTurns (nT)

according to (13).

$$\text{maxThreads} = \text{nB} \cdot \text{dR} \cdot \text{dA} \cdot \text{nT} = 4 \cdot 12 \cdot 36 \cdot 4 = 6912 \quad (13)$$

Each of these threads computes the sum of the induced velocities of all vortex segments for one grid node. This makes possible to subdivide each thread in circa 6000 further threads and to add the contributions of all vortex segments as the last step. On the GPU used, execution of an arithmetic operation requires 22 clock cycles [5]. In order to avoid this latency an operation should be available in every clock cycle. For 488 GPU cores, this requires at least circa 10,000 threads. This is given through the subdivision of the threads; the maximum number of threads is then 6000^2. Numba does not automatically perform the summation of the contributions in the end. Thus the code has to be significantly restructured. The restructuring does not match our goal of an as simple and as clear as possible implementation in Python. Therefore we just exploit the 6912 independent threads from (13) in our Numba inplementation and do not use the capacity of the GPU completely. A higher thread number than that of our test problem would give an increased performance on the GPU.

Our Numba implementation of the Freewake benchmark achieved a runtime of 0.77 s on the GPU with a performance of 7.79 GFLOPS. This was the shortest execution time we achieved with a Python implementation. A run of the Fortran implementation exploiting GPU parallelization with OpenACC [24] took 0.086 s with a performance of 69.77 GFLOPS (cf. Table 3 and Figure 10). While the GPU Numba version could be fast produced the development of the GPU Fortran version with OpenACC parallelization was extremely time-consuming.

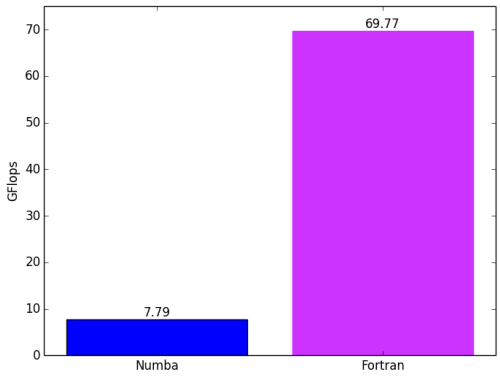

Figure 10: GFLOPS of the GPU implementations.

7. CONCLUSION

This investigation confirms that standard Python implementations without additional libraries as a rule achieve distinctly less compute performance than implementations in low-level programming languages like C or Fortran. For the CFD benchmark considered here, the standard Python version merely achieved three-thousandths of the performance of the Fortran version on a single CPU core. NumPy exploitation could increase performance to a sixth of that of Fortran, Cython exploitation to half of the Fortran performance. However, Cython does not maintain the clear syntax of Python, but rather supports C style programming.

On multi-core systems with shared memory, the Cython with OpenMP implementation again achieves half of the performance of the parallel Fortran version. The only parallel implementation examined which maintains a clear Python syntax exploits *Global Arrays* and reaches a tenth of the Fortran version performance.

The Numba implementation for GPUs is faster than all CPU Python implementations in this investigation. However, the Numba implementation performance is far from the optimum; the performance of the Fortran with OpenACC version is about 10 times higher on the GPU.

We conclude that Python programs for complex application kernels as a rule can not achieve the performance of corresponding programs in low-level languages on parallel hardware. However, moderate and in many cases satisfactory performance can be reached if appropriate Python libraries are exploited. In this investigaton, all parallel Python versions showed satisfactory speed-ups on a shared-memory multi-core system. On the other hand, development of the parallel Python kernels was considerably faster than that of the parallel Fortran kernels. This is a clear plus in parallel code productivity for Python and matters particularly if large complex application codes and not only small benchmark kernels have to be developed. We definitively recommend Python for parallel prototype development. If the runtime of a parallel application is not too critical and performance losses do not harm too much software development with Python plus extensions for parallel programming is a real option. From our investigation, we prefer Python with *Global Arrays* for parallel programming on multi-core systems since it provides an appropriate high programming level. Cython with parallel constructs promises higher performance, but the syntax is essentially C style.

In our investigation, Numba offered simple access to GPU performance. With Numba, CFD kernel development for GPUs was significantly faster than with Fortran and OpenACC. Parallelization via OpenACC directives should be easy, but requires to pay attention to several intricacies. For example, not all Fortran, C or C++ language features are applicable within an OpenACC region. Numba is a simple option for kernel acceleration by GPU usage. The programming style is clear and structured. However, GPU performance of a Numba kernel can be far from optimum.

We expect that the community behind NumPy, Numba, Cython and Global Array projects, e.g., will actively further develop their solutions so that parallel programming with Python will become increasingly more efficient on modern parallel hardware. Together with the clear productivity advantage of parallel Python programming to parallel programming in low-level languages, this would justify a more general use of Python with parallel extensions in the HPC area.

8. ACKNOWLEDGMENTS

The authors would like to thank Prof. Dr. Harald Kornmayer from Duale Hochschule Baden-Württemberg Mannheim for

co-supervision of Joachim Illmer's work in this investigation and for giving valuable hints.

9. REFERENCES

[1] H. M. Atassi. The biot-savart law. https://www3.nd.edu/~atassi/Teaching/ame%2060639/Notes/biotsavart.pdf, 2015. Accessed: 4th September 2015.

[2] A. Basermann, M. Röhrig-Zöllner, and J. Hoffmann. Porting a parallel rotor wake simulation to gpgpu accelerators using openacc. http://www.t-systems-sfr.com/e/deu/abstract.2014_7.php, 2014. Accessed: 3rd September 2015.

[3] Blas — basic linear algebra subprograms. http://www.netlib.org/blas/, 2015. Accessed: 4th September 2015.

[4] D. A. Boxwell, F. H. Schmitz, W. R. Splettstößer, and K. J. Schultz. Helicopter model rotor-blade vortex interaction impulsive noise: Scalability and parametric variations. *Journal of the American Helicopter Society*, 32(1):3–12, 1. Januar 1987.

[5] Cuda toolkit documentation - multiprocessor level. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level, 2015. Accessed: 3rd September 2015.

[6] The cython compiler for writing c extensions for the python language. https://pypi.python.org/pypi/Cython/, 2015. Accessed: 3rd September 2015.

[7] Using the cython compiler to write fast python code. http://www.behnel.de/cython200910/talk.html, 2015. Accessed: 3rd September 2015.

[8] J. Daily, P. Saddayappan, B. Palmer, S. K. Manojkumar Krishnan, A. Vishnu, D. Chavarría, and P. Nichols. High performance computing in python using numpy and the global arrays toolkit, 08 2011. Remarks by Chairman Alan Greenspan at the Annual Dinner and Francis Boyer Lecture of The American Enterprise Institute for Public Policy Research, Washington, D.C. [Accessed: 3rd September 2015].

[9] Intel xeon processor e5645 specifications. http://ark.intel.com/de/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI?q=e5645, 2010. Accessed: 3rd September 2015.

[10] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2010.

[11] Intel math kernel library. https://software.intel.com/en-us/intel-mkl, 2015. Accessed: 28th July 2015.

[12] Intel xeon processor 5600 series. http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf, 2011. Accessed: 3rd September 2015.

[13] Likwidbench wiki. https://code.google.com/p/likwid/wiki/LikwidBench, 2015. Accessed: 3rd September 2015.

[14] Homepage of matlab. http://de.mathworks.com/products/matlab/, 2015. Accessed: 3rd September 2015.

[15] The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/, 2015. Accessed: 3rd September 2015.

[16] Numba — mode of operation. http://on-demand.gputechconf.com/supercomputing/2013/presentation/SC3121-Programming-GPU-Python-Using-NumbaPro.pdf, 2015. Accessed: 3rd September 2015.

[17] Homepage of numba. http://numba.pydata.org/, 2015. Accessed: 3rd September 2015.

[18] Ways to parallelize - numba-users mailinglist. https://groups.google.com/a/continuum.io/forum/#!topic/numba-users/UN4sDSr8Iew, 2014. Accessed: 3rd September 2015.

[19] Numba mailinglist. https://groups.google.com/a/continuum.io/forum/#!topic/numba-users/iOnkSJTcF0A, 2014. Accessed: 3rd September 2015.

[20] Numbapro — continuum analytics. http://docs.continuum.io/numbapro/index, 2015. Accessed: 3rd September 2015.

[21] Homepage of numpy. http://www.numpy.org/, 2015. Accessed: 3rd September 2015.

[22] Nvidia cuda. https://developer.nvidia.com/about-cuda, 2015. Accessed: 3rd September 2015.

[23] Nvidia tesla c2075 companion processor. http://www.nvidia.de/content/PDF/data-sheet/NV_DS_Tesla_C2075_Sept11_US_HR.pdf, 2011. Accessed: 3rd September 2015.

[24] Homepage der openacc api. http://www.openacc-standard.org/, 2015. Accessed: 3rd September 2015.

[25] Opencl - the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/, 2015. Accessed: 3rd September 2015.

[26] Openmp specification for parallel programming. http://openmp.org/wp/, 2015. Accessed: 3rd September 2015.

[27] W. Splettstößer, R. Kube, U. Seelhorst, W. Wagner, A. Boutier, F. Micheli, and K. Pengel. Higher harmonic control aeroacustic rotor test (hart) - test documentation and representative results. http://elib.dlr.de/36398/, 1996. Accessed: 3rd September 2015.

[28] Top500 list - june 2015. http://www.top500.org/list/2015/06/, 2014. Accessed: 3rd September 2015.

[29] The abstraction-optimization tradeoff. http://blog.vivekhaldar.com/post/12785508353/the-abstraction-optimization-tradeoff, 2015. Accessed: 3rd September 2015.

[30] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. UCB/EECS 2008-134, Univ. of California, Berkeley, CA, oct 2008.