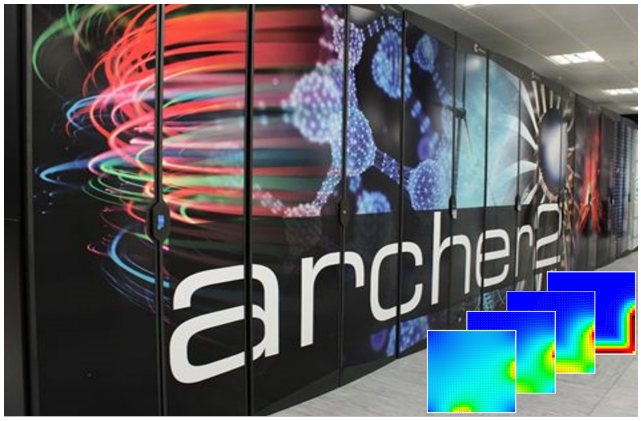
Accelerating Python on AMD CPUs



Alejandro Dinkelberg
Jiahua Zhao

Numerical Python calculations can achieve speeds in a similar range to C and Fortran.

The broad application of Python and its simplicity lead to a huge community of users. The advantages of readability and dynamic typing for writing Python code come along with a slow computational performance which in turn makes it Python unattractive for complex calculations. Nevertheless, its popularity exerts pressure to accelerate Python.

Complex simulations and extensive numerical calculations are highly recommended candidates to execute on High Performance Computing (HPC) systems. Recently, the EPCC[1] topped up their HPC systems and is now using the UK HPC Tier-1 AMD-based Supercomputer ARCHER2. This leads to an interest of evaluating the performance of Python on AMD-based systems since there rarely exist studies on the performance improvements for AMD-based HPC systems.

The languages C and Fortran deliver the highest standards for computational performance and therefore they will play a benchmark role for our Python code. Although naive Python lags far behind, we will show here how to transform Python code to accomplish C/Fortran performance. We will mainly measure the performance on the HPC system ARCHER2 and compare it to an intel-based HPC system (Cirrus). The results will show us what improvements we can expect and how competitive Python is.

Computational Fluid Dynamics

Our performance benchmark is an example from the Computational Fluid Dynamics (CFD). It simulates a fluid flow in cavity approximating a flow pattern. The algorithm discretises the problem on a two-dimensional grid, to approximate the partial differential equations by using the Jacobi algorithm. It can be described as an iterative process which converges to a state with an approximated solution. Each iteration adds precision to the result until reaching a stable state. Additionally, increasing the grid size leads to higher precision but is also more computationally intensive.

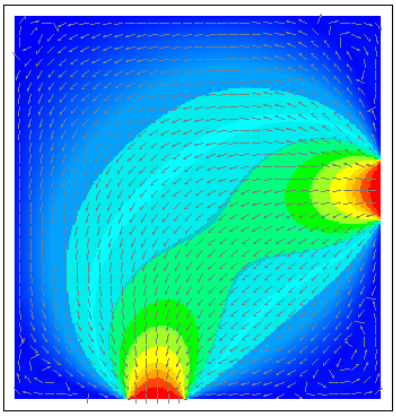


Figure 1: Example of an approximated flow pattern with in and out flow

How is the problem being solved?

The determination of the flow pattern is broken down to a finite problem size and is approximated by using a grid. For every cell , we use the stream function Ψ for zero viscosity to calculate fluid velocity:

$$\nabla^2\Psi = \frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2} = 0 \qquad (1)$$

The finite version of the equation gives us the option to compute the stream by averaging the value of the cell's four nearest neighbours:

$$\begin{aligned} 0 =& \Psi_{i-1,J} + \Psi_{i+1,j} + \Psi_{L,j-1} \\ &+ \Psi_{i,j+1} - 4\Psi_{i,j} \end{aligned} \qquad (2)$$

Scalabilty of CFD approximation

1. The **Scale factor (*sf*)** determines the grid size (minimum size 32 x 32). The grid is represented as a matrix so that *sf* specifies the required memory size in our calculation.

2. The **number of iterations** is the precision parameter of the algorithm.

3. The **Reynolds number** (Re) is the degree of the fluid's viscosity. It adds complexity to the problem while making it more realistic.

Note that the main computational costs are in the iterative process of the algorithm (see Fig. 4, *jacobistepvort*). In our project, we focus on improving this process as the calculation is the same for every iteration.

Methods to improve our Python code

The benchmark for our algorithm improvements is a naively written Python program. To improve its velocity and to reduce computational costs, we introduce the following approaches:

Code optimisation - It can be obtained by vectorize the for-loops with a matrix calculation provided by the *NumPy*[2] package. Additional improvement is promised by the package *NumExpr*[3] that resolves mathematical operations efficiently to accelerate NumPy's matrix calculations.

Divide and Conquer - The serial version of our grid calculation is using just one processor at a time for the calculation. What if we divide the grid and run it on multiple processes, using multiple CPUs? The language-overarching message passing interface (MPI) is a technique for processor communication. *Mpi4py*[4] is a Python package that implements the functionality of MPI.

GPUs instead CPUs - The specialty of the GPUs is the wide-spread parallelisation of the small tasks. Therefore, GPUs often have more than hundreds of device cores. Here, we can use *Cuda* from the *Numba*[5] package to implement the code for the GPU.

Focus: Parallelisation of Python code

We use a one-dimensional domain decomposition[6] and narrow down the problem into smaller ones. In Python, we decompose our grid one-dimensional (x-dimension) to slice it into sub grids in order to allocate each sub grid to a single process. These calculations can run in parallel.

Our Jacobi algorithm is based on information about the cell's neighbours. Splitting up the grid provokes a lack of information at the boundaries of the sub grids. Hence, we have to enable the communication between the processes.

Halos are boundary layers of each sub grid that are shared with another neighbouring sub grid. If the halo of a sub grid is updated and it gets informed by its neighbouring sub grid, this sub grid updates and informs that is was updated (see Fig. 2). The halo swaps ensure that the information is up-to-date.

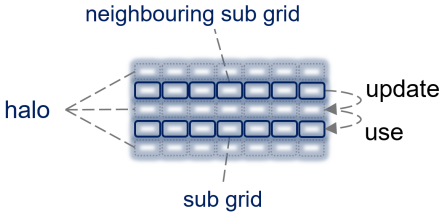


Figure 2: Sub grids performing halo swaps

About ARCHER2

EPCC's ARCHER2 is an HPE Cray EX supercomputing system with an estimated peak performance of 28 PFLOP/s. The final system will have 5,848 dual sockets compute nodes based on AMD EPYC Zen2 (Rome) 64 core CPUs running at 2.25 GHz, given 748,544 cores in total.[7] Each standard compute node has 256 GiB of DDR4 memory. Except for GPU programs (using single NVIDIA Tesla V100-SXM2-16GB on Cirrus[8]), our CPU programs run on standard compute nodes with Cray compiler (cc & ftn) and Cray python (3.8.5.0). See Fig. 3 for details.

One Standard CPU Compute Node	
Processor	2× AMD Zen2 7742, 2.25 GHz, 64-core
Memory per node	256 GiB
Interconnect	HPE Cray Slingshot, 2× 100 Gbps bi-directional per node
Operating system	HPE Cray Linux Environment (based on SLES 15)
Scheduler	Slurm
Compilers	HPE Cray Compiling Environment (CCE)
Python	Cray Python (3.8.5.0)
Parallel libraries	HPE Cray MPICH2

Figure 3: The hardware & software details of ARCHER2: standard compute node.

Profiling and optimizing

First, we run the serial Python benchmark programs using a single rank. Given the problem size $sf = 256$, the naive Python code needs about 896 sec per iteration, whereas the NumPy version only takes about 1.2 sec. We use cProfile to analyze the time consumption during run time (see Fig. 4).

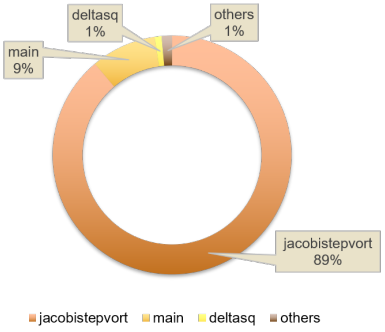


Figure 4: The hotspot functions in Python programs.

The function *jacobistepvort* is computationally intense and takes up to 90% of the time, followed by *main* and *deltasq*. This loop implements the equations for finite viscosity which are a much more complicated than Equation (2) but have the same general structure where each point depends on its four neighbours. Further on, we focus on *jacobistepvort*. The naive Python code updates the data at each point by using double for-loops, whereas the NumPy version allocates the memory before calculating and iterates internally to update the data, which is more efficiently implemented. To speed up the Numpy version, we introduce NumExpr to speed up the array calculation. We use the function *evaluate* to convert the original calculation expression into a fast numeric expression and to optimize the data access. In addition, we speed up naive Python by using Numba shifting the *jacobistepvort* calculation to the GPU platform.

Results for serial versions

The serial Python programs, except for Numba version running on a GPU on Cirrus, were tested on ARCHER2 (see Fig. 5).

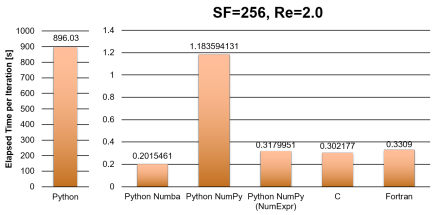


Figure 5: Performance: serial (single CPU core / GPU).

In comparison to the serial versions of the CFD program, the implementation of the GPU-based version is 6 times faster than the NumPy version and even slightly faster than the C and Fortran versions. It is important to ensure that the data copying from CPU to GPU and back is minimised. The NumPy version is nearly 800 times faster than naive Python, and the performance of NumPy version after NumExpr optimization improves greatly, even compared to the performance of C and Fortran versions. This indicates that NumExpr has a good optimization for NumPy arrays calculation at this problem size.

Parallel programming

To improve the comparability and the evaluation of the performance, we use the serial NumPy version as a baseline

(1.18s per iteration). In Fig. 6 we display the results of the parallel versions on one ARCHER2 node for: C-MPI, Fortran MPI and Python with NumPy MPI and with NumPy using NumExpr.

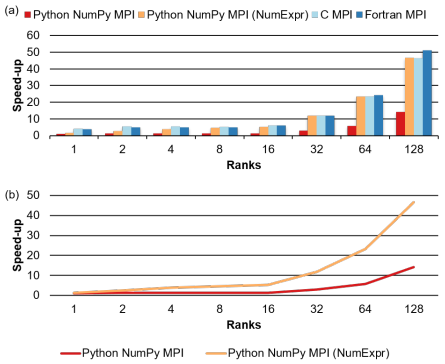


Figure 6: Performance: parallel (single node / multiple ranks).

Overall, the speed-up of all parallel programs increases with the number of ranks (see Fig. 6(a)). For the Python programs, the performance improvement is small with a small number of ranks (1 to 16), as well as for C and Fortran. The NumExpr optimized version is close to C and Fortran at 16 ranks. In the case of a large number of ranks (32 to 128), the performance of all programs improves significantly. The Python NumPy MPI consistently lags behind the other three, while the performance of NumExpr optimized version is still close to C and Fortran. In Fig. 6(b) we see that after more than 16 ranks, the scalability of the NumExpr version is better than non-NumExpr version.

When the number of ranks (multiple compute nodes) increases further (Fig. 7), the scalability of Python programs decreases, especially after 256 ranks. The performance of C and Fortran programs increases sharply, while the speed-up of Python programs increases slowly (see Fig. 7(a)). In particular, it is shown that the scalability of non-NumExpr version is better than NumExpr in Fig. 7(b). It is slightly better than linear while going from 128 to 2048 ranks. The C and Fortran scalability is massively super-linear, presumably because of cache effects: as the local problem size gets smaller, it suddenly fits into cache and gets a huge performance jump (between 256 and 512 ranks). Mpi4py may have more serious blocking problems because the communication overhead across nodes is often larger than that within nodes. We would like to address this problem in future work as we expect also a speed-up with mpi4py on more ranks.

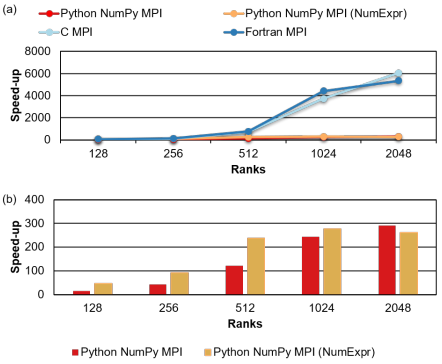


Figure 7: Performance: parallel (multi-node).

Conclusion

Our work demonstrated the feasibility of Python HPC, and we are convinced that Python solves scientific problems in a flexible manner. Python can be used efficiently on HPC systems when combined with scientific acceleration packages such as NumPy, NumExpr, Numba and so on. For allocating the problem over multiple nodes and achieve high-speed parallel computing, Mpi4py can be used. Our results show that the combination of NumPy and NumExpr gives Python programs similar performance to C and Fortran in the serial case. In the parallel case, if the number of cores is at its maximum and there is no cross-node communication, the performance of Python programs (about 48x speed-up) combined with "*NumPy + NumExpr + Mpi4py*" is close to C and Fortran.

Nevertheless, if multi-nodes are used, the NumExpr Python version has no advantage here and the Numpy version is more extensible. Of course, C and Fortran still have unbeatable computational advantages for large-scale computing, so we recommend that researchers choose the right programming language for the actual problem: If you want to solve the problem more convenient, you can use Python although you should use accelerating techniques. Naive Python is not recommended at all. If you want to solve the problem on large-scale nodes, you may prefer C or Fortran with MPI to receive the most efficient results.

Future work

We will explore Python for large-scale computing in the future. It will be interesting to investigate larger problem sizes and optimize Python performance on more nodes on ARCHER2, and we will try to optimize it more for AMD Zen2 architecture.

At present, we are only beginning to use the Heterogeneous Computing Platforms (GPUs) to accelerate Python programs, and we have not yet fully utilized the computing power of GPUs. So in the future, we will improve GPU acceleration performance for Python programs and implement Python programs which support multi-GPU use.

Acknowledgement

Thanks for a fantastic virtual Summer of HPC, we have learned Python HPC skills, which will be of great help to our future. Thanks to our mentors David Henty and Stephen Farr, to the Summer of HPC organisers and to friendly partners.

References

[1] EPCC at The University of Edinburgh. https://www.epcc.ed.ac.uk/

[2] NumPy. https://numpy.org/

[3] NumExpr Documentation Reference. https://numexpr.readthedocs.io/projects/NumExpr3/en/latest/

[4] MPI for Python. https://mpi4py.readthedocs.io/en/stable/

[5] Numba: A High Performance Python Compiler. numba.pydata.org/

[6] Gropp, W. D., & Keyes, D. E. (1992). Domain decomposition methods in computational fluid dynamics.

[7] ARCHER2 Hardware & Software. https://www.archer2.ac.uk/about/hardware.html

[8] Cirrus powered by EPCC. https://www.cirrus.ac.uk/

PRACE SoHPCProject Title

Performance of Parallel Python Programs on ARCHER2

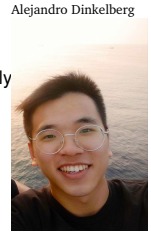
PRACE SoHPCSite

EPCC, UK

PRACE SoHPCAuthors

Alejandro Dinkelberg, University of Limerick, Ireland
Jiahua Zhao, University of Padua, Italy


Alejandro Dinkelberg


Jiahua Zhao

PRACE SoHPCMentor

Dr. David Henty, EPCC, UK

PRACE SoHPCContact

Alejandro, Dinkelberg, University of Limerick
E-mail: Alejandro.Dinkelberg@ul.ie
Jiahua, Zhao, University of Padua
E-mail: jiahua.zhao@studenti.unipd.it

PRACE SoHPCSoftware applied

C, Fortran, Python, cProfile, NumPy, NumExpr, Numba, Mpi4py

PRACE SoHPCMore Information

https://summerofhpc.prace-ri.eu/performance-of-parallel-python-programs-on-archer2/

PRACE SoHPCProject ID

2115