An Abstraction for Distributed Stencil Computations Using Charm++

Aditya Bhosale(✉), Zane Fink, and Laxmikant Kale

University of Illinois Urbana-Champaign, Champaign, USA
{adityapb,zanef2,kale}@illinois.edu

Abstract. Python has emerged as a popular programming language for scientific computing in recent years, thanks to libraries like Numpy and SciPy. Numpy, in particular, is widely utilized for prototyping numerical solvers using methods such as finite difference, finite volume, and multigrid. However, Numpy's performance is confined to a single node, compelling programmers to resort to a lower-level language for running large-scale simulations. In this paper, we introduce CharmStencil, a high-level abstraction featuring a Numpy-like Python frontend and a highly efficient Charm++ backend. Employing a client-server model, CharmStencil maintains productivity with tools like Jupyter notebooks on the frontend while utilizing a high-performance Charm++ library on the backend for computation. We demonstrate that CharmStencil achieves orders of magnitude better single-threaded performance compared to Numpy and can scale to thousands of CPU cores. Additionally, we showcase superior performance compared to cuNumeric and Numba, popular Python libraries for parallel array computations.

Keywords: Python · Stencil · Distributed · Numpy · Charm++

1 Introduction

In recent years, Python has witnessed significant adoption in various domains, including machine learning, scientific computing, and data analytics. The expressiveness and productivity provided by tools like Jupyter notebooks, along with composable high-performance libraries built on top of foundations such as Numpy [6] and SciPy [11], have made it possible to implement sophisticated algorithms with relatively low programming effort and decent performance.

As a result, traditionally HPC-oriented domains like scientific computing have seen a rise in the popularity of Python. The interactivity and simplicity of Python tools and libraries has made it possible to prototype complex physical systems with relative ease. In particular, Numpy has simplified expressing computations on structured grids used for numerical methods in scientific computing, such as finite difference, finite volume, and multigrid methods. The highly regular nature of these computations has static and locally contained dependence patterns and can be succinctly expressed using slicing notations commonly used

P. Diehl et al. (Eds.): WAMTA 2024, LNCS 14626, pp. 123–134, 2024.
https://doi.org/10.1007/978-3-031-61763-8_12

in Numpy. Figure 1 shows a single iteration of a 2D Laplace equation solver using Numpy.

While Numpy exhibits good single-threaded performance, leveraging efficient C implementations for common array operations, its parallel capabilities are restricted to a single node, relying on multi-threaded BLAS libraries for specific built-in operations. Moreover, for an arbitrary sequence of operations, Numpy suffers from high Python overheads and the creation of temporary arrays. As a result, domain scientists are forced to reimplement their applications using a lower-level programming model to run on a larger scale.

To address these challenges, we developed CharmTyles, a framework for implementing high-performance scalable abstractions on elastic parallel machines while maintaining the productivity offered by Python, along with tools like Jupyter notebooks. CharmTyles uses a client-server architecture with a Python client on the frontend and a parallel Charm++ server on the backend. In this paper, we present CharmStencil, a high-level stencil abstraction developed using the CharmTyles framework with a Python frontend and a Charm++ backend that can execute arbitrary operations on a structured grid on distributed memory machines with minimal modifications to the Numpy code.

To create the stencil abstraction, we developed a frontend library that dynamically constructs an Abstract Syntax Tree (AST) using operations on the fields of the structured grid (Sect. 3.1). Subsequently, this frontend library asynchronously sends the AST to the backend. On the backend, we implemented a code generator that dynamically generates, compiles, and loads the local computation function. Additionally, the backend manages the necessary data movement for the local computation and invokes the local computation function (Sect. 3.2). Finally, we demonstrate the effectiveness of our asynchronous client-server model in overlapping Python overhead from the frontend with useful computation on the backend.

We further compare our abstraction with two state-of-the-art libraries - cuNumeric [1], a drop-in replacement library for Numpy based on the Legion runtime system, and Numba [8], a compiler for a subset of Python, showcasing superior performance. Additionally, we benchmark CharmStencil against a hand-written Charm++ benchmark, showing minimal Python overhead on up to 1024 cores (Sect. 4).

```
u[1:-1, 1:-1] = (u[2:, 1:-1] + u[:-2, 1:-1] + u[1:-1, 2:] + u[1:-1,:-2]) / 4
```

Fig. 1. 2D Laplace solver iteration using Numpy

2 Background

Charm++ [7] is an asynchronous message-driven parallel programming language. Users express their computation in terms of objects that interact with each other

via asynchronous messages. These objects may belong to collections, called chare arrays; each chare array has its index structure, such as a 2D dense array, and supports collective communication operations over its members. The runtime system handles the mapping of these objects to processors. Each Processing Element (PE) has a scheduler and several message queues. When a message is to be sent to an object, the runtime system looks up the location of that object in a distributed location manager and directs the message to the corresponding PE where it is enqueued in one of the message queues. The scheduler picks messages out of the message queues and delivers them to the appropriate object.

The separation of computation from physical processors allows users to write parallel applications with runtime adaptivity without additional programming effort. For example, Charm++ can automatically balance the load every few iterations of an iterative application by migrating objects among processors. Moreover, if the parallel computation is over-decomposed, Charm++ can help facilitate computation-communication overlap.

This runtime adaptivity makes Charm++ an ideal choice for a programming model in applications with dynamic load imbalance and for execution on heterogeneous machines. The support for the migratability of objects also facilitates resource elasticity, making it a strong candidate for developing scalable applications in cloud environments.

3 Methodology

CharmTyles is a framework for developing abstractions based on a client-server model with a Python frontend running in a Jupyter Notebook on the user's machine and a Charm++ backend server running on a parallel machine. The backend has multiple collections of tiles, distributed across an elastic parallel machine such as in the cloud, a cluster, or a supercomputer, orchestrated from the frontend. Figure 2 shows the client-server architecture of CharmTyles.

The Python frontend and the Charm++ backend communicate using the Converse Client-Server (CCS) interface. CCS allows external programs to inject messages into Charm++ message queues. These messages are then picked up by the scheduler and passed to the corresponding handler. In our architecture, we send messages from the Python frontend to PE 0 of the Charm++ backend server. PE 0 then processes the message and sends a broadcast to all PEs specifying the computation.

The communication between the frontend and the backend is asynchronous. The frontend Python execution, i.e. the Python overhead, can thus be overlapped with useful computation on the backend. The frequency of messages from the frontend to the backend is a user-configurable parameter and results in different levels of pipelining as shown in Fig. 3. A smaller frequency of messages between the frontend and the backend results in a lower cost of broadcast on the backend, but will also have a smaller overlap as seen in Fig. 3a. On the other hand, a larger frequency of messages will result in a larger overlap, but will also incur a higher cost of broadcasting on the backend as shown in Fig. 3c.

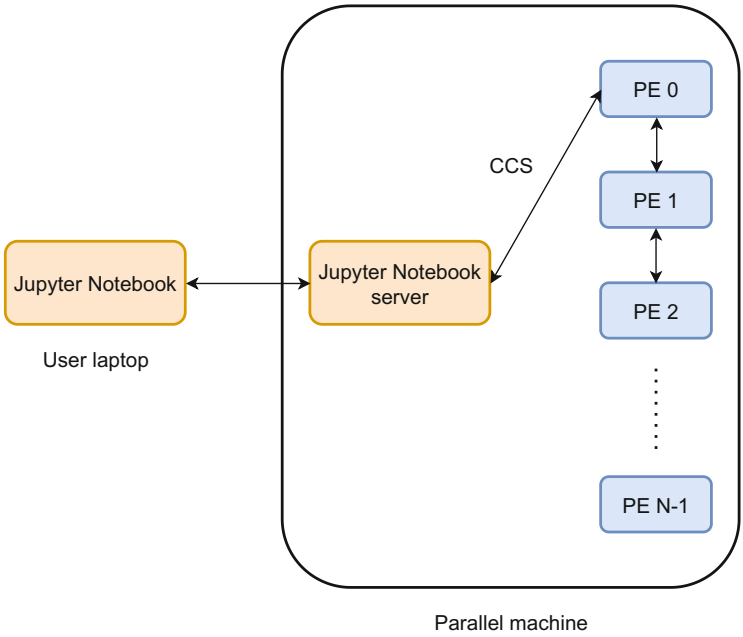


Fig. 2. CharmTyles client-server architecture

CharmStencil is an abstraction for stencil computations written using the CharmTyles framework. The frontend has a Numpy-like interface that expresses the stencil code. The computational grid is decomposed into equally sized tiles on the backend distributed across the parallel machine running the server. The following sections describe the frontend and backend of CharmStencil in more detail.

3.1 Frontend

Figure 4 shows a 2D Laplace equation solver written using CharmStencil. Users create a subclass using the provided `Stencil` class and overload the `iterate` method to express their computation. The users also specify the over-decomposition factor, which determines the number of tiles the domain is decomposed into, the ghost depth for each field, the size of the grid, and the number of fields to create. The call to `initialize` sends a message to the backend to create the stencil with the requested specifications. When `solve` is called on the object of this class, the `iterate` function is called in a loop until it returns `true`. The `exchange_ghosts` function is called in `iterate` to initiate the exchange of ghost data for the fields passed as arguments to the call. The `backend_freq` option sets the number of iterations after which a message is sent to the backend.

The frontend builds an AST for every call to `iterate`. It keeps track of all unique ASTs and a list of which AST needs to be executed for each iteration.

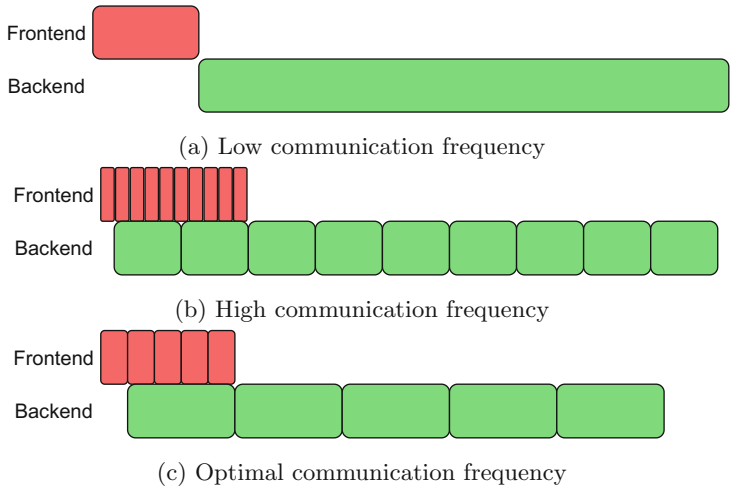


Fig. 3. Pipelined execution between frontend and backend helps hide Python overheads from the frontend

When the number of iterations reaches the maximum threshold specified by the user, the unique ASTs and the list of ASTs to execute at each iteration are serialized and sent to the backend. Figure 5 shows the AST generated for a 2D Laplace equation solver.

3.2 Backend

When the backend receives the creation message, it generates a chare array, with each element corresponding to a tile in the grid decomposition. The size of the chare array is determined by the over-decomposition factor provided by the user and the number of PEs on which the server is running. Each element of the chare array then allocates the requested size for its local data, along with buffers for communicating ghost data.

The backend handler on PE 0 generates C++ code for each unique AST received from the frontend and compiles it to a shared object identified by a 32-bit hash of the AST. The handler on PE 0 then broadcasts the list of the 32-bit hash values of each unique AST, the list of ASTs that need to be executed at each iteration, and the fields that need ghost exchanges at each iteration to all PEs. The compute function corresponding to each hash value is dynamically loaded from the file system on each logical node and cached locally on every PE. Every chare does its ghost exchanges, if any, at each iteration and then calls the compute function.

```
class Jacobi2D(Stencil):
    def __init__(self, n, interface):
        self.x, self.y = self.initialize(
            n, interface=interface, backend_freq=100, odf=2,
            num_fields=2)
        self.itercount = 0
        self.boundary_iter = True

    def iterate(self, nsteps):
        if self.boundary_iter:
            self.boundary(100.)
            self.boundary_iter = False
            return True
        self.exchange_ghosts(self.x)
        self.y[1:-1, 1:-1] = 0.25 * (self.x[:-2, 1:-1] + self.x[2:, 1:-1] +
                                     self.x[1:-1, :-2] + self.x[1:-1, 2:])
        self.x, self.y = self.y, self.x
        self.itercount += 1
        return self.itercount != nsteps

    def boundary(self, bc):
        self.x[0, :] = self.y[0, :] = bc
        self.x[-1, :] = self.y[-1, :] = bc
        self.x[:, 0] = self.y[:, 0] = bc
        self.x[:, -1] = self.y[:, -1] = bc

grid = Jacobi2D(n, interface)
grid.solve(1000)
```

Fig. 4. 2D Laplace equation solver example using CharmStencil

4 Performance Results

The following experiments were run on the NCSA Delta machine. Each node of Delta has 2 sockets with an AMD EPYC 7763 64-Core processor on each socket with HPE Slingshot 11 interconnect. We use the non-SMP Charm++ MPI build with OpenMPI 4.1.2. We compare the performance of CharmStencil with cuNumeric, Numba, Charm++, and sequential Numpy. Because of issues in building the multi-node version on Delta, we show comparisons with only the single-node build of cuNumeric.

Figure 6 shows the effect of pipelined execution on a 3D Laplace equation solver on 16 million grid points for 1024 iterations. A smaller `backend_freq` results in more frequent communication between the frontend and the backend resulting in greater overlap but also large communication cost and a higher Python overhead due to the cost of serialization of the AST. Whereas, a larger `backend_freq` results in low communication cost but also low overlap. We can hide almost all of the Python overhead for a range of choices of `backend_freq` between the 2 extremes.

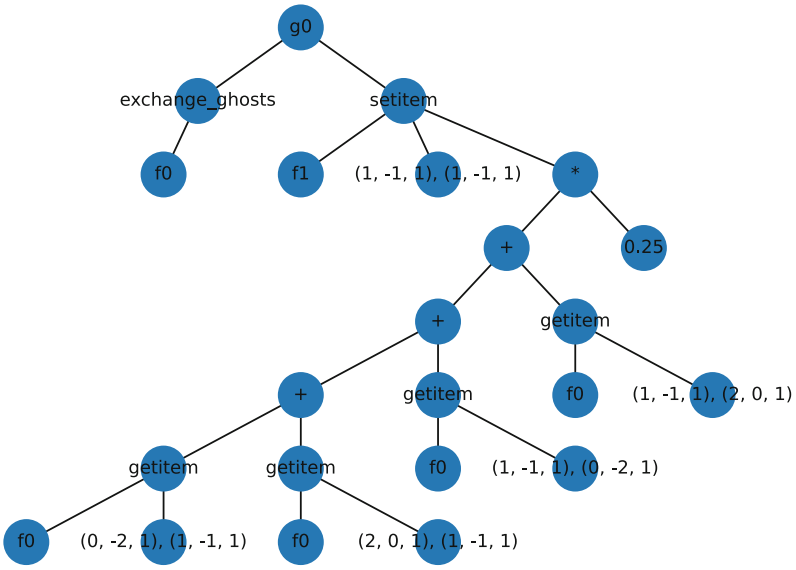


Fig. 5. AST generated by the frontend for a 2D Laplace equation solver

Figure 7 shows the scaling performance for a 2D Laplace equation solver for 128 iterations. The number of cores varies from 1 to 4096 (32 nodes). We see that Numpy cannot parallelize the operations and incurs significant overheads from creating temporary arrays for each operation. cuNumeric performs better than Numpy as the number of cores are scaled up, but still shows significant overhead. Numba exhibits decent scaling performance on a single node but requires users to indicate which loops to parallelize, similar to OpenMP, explicitly. Consequently, without explicit loop tiling from the programmer, Numba shows poor cache performance, resulting in worse strong and weak scaling than CharmStencil and Charm++ as we see in this case. CharmStencil can hide most of the Python overheads and match the Charm++ strong scaling performance until 1024 cores. Beyond the grain size of 3 ms per iteration, the Python overhead and communication costs between the frontend and the backend affect the strong scaling performance.

Figure 8 shows the scaling performance for a 2D Burger's equation which solves the following set of coupled PDEs,

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \end{aligned} \tag{1}$$

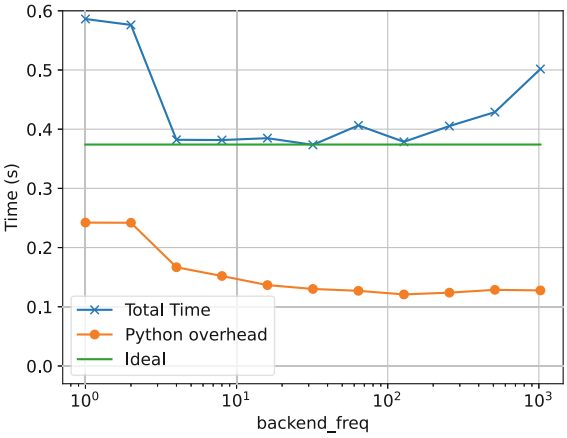


Fig. 6. Effect of pipelined execution on hiding Python overhead

The discretization of Burger's equation leads to a larger number of operations per iteration than the Laplace equation. As a result, the performance difference between Numpy and CharmStencil widens since the latter avoids the creation of intermediate arrays for each operation.

5 Related Work

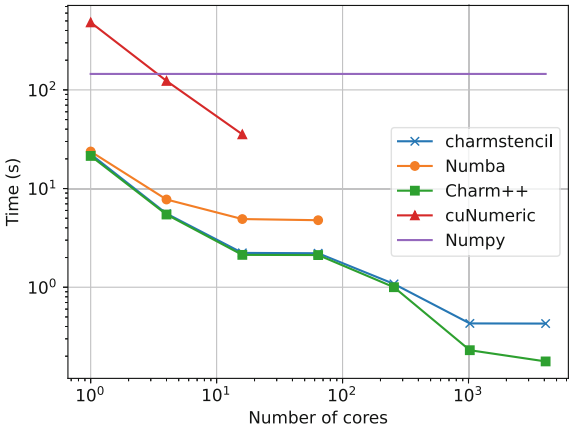
There have been several projects aimed at parallelizing array computations while maintaining the productivity offered by Python. These projects span from parallel languages that can interoperate with Python to high-level libraries that invoke an underlying parallel implementation of a set of algorithms.

Cython [2] is a compiled language that serves as a superset of Python, enabling users to write Python code that interoperates with C/C++. Cython code is compiled to C using the Python-C API for Python objects, and it also supports parallel programming using OpenMP.

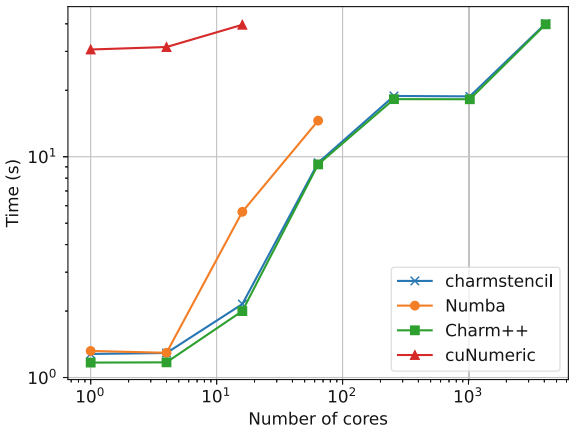
Numba [8] is an LLVM-based JIT compiler that translates a subset of Python into machine code. Numba generates parallel code for multicore CPUs and supports execution on GPUs. However, Numba cannot auto-parallelize sequential code; users need to express parallelism in their code using primitives defined by Numba.

JAX [3] is an XLA-based compiler for array computations on CPUs, GPUs, and TPUs, primarily targeted towards machine learning applications. Similar to Numba, users need to express parallelism in their code for JAX to run on parallel machines.

Dask [10] is a task-based runtime system supporting distributed execution with an array API almost identical to that of Numpy. However, the centralized



(a) Strong scaling



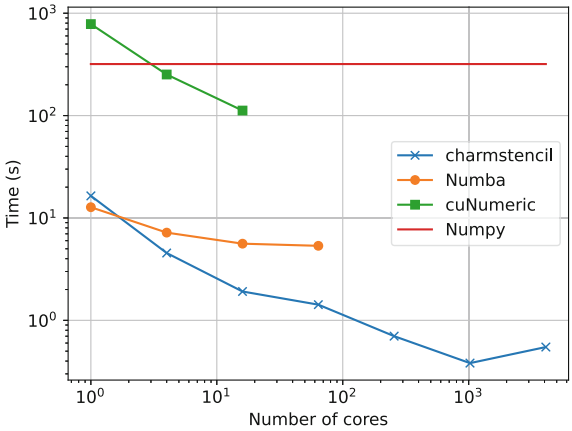
(b) Weak scaling

Fig. 7. Scaling performance results for 2D Laplace equation solver

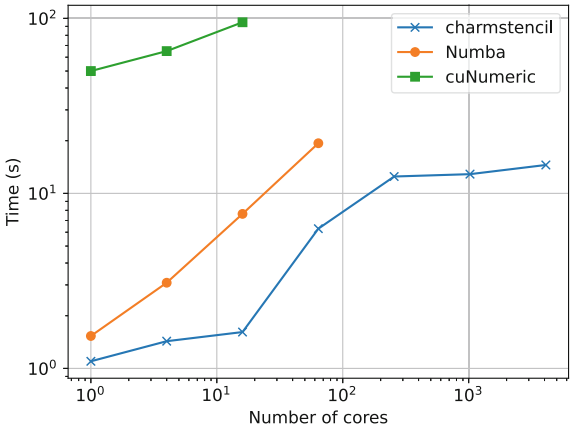
dynamic task scheduler in Dask does not scale well on large machines. Moreover, local array operations in Dask are handled by Numpy, thus suffering from the same overheads as Numpy.

cuNumeric [1], as mentioned earlier, is another drop-in replacement library for Numpy built on top of the Legion runtime system. cuNumeric exhibits impressive scaling performance on multiple GPUs, but at least in the current implementations, our CPU performance is better.

Arkouda [9] is a Python library with Numpy-like arrays and Pandas-like dataframes designed for exploratory data analytics. Arkouda also utilizes a client-server model with a Python frontend and a Chapel server on the backend.



(a) Strong scaling



(b) Weak scaling

Fig. 8. Scaling performance results for 2D Burger's equation solver

Apart from these, there also exist low-level libraries for distributed programming such as MPI4Py [4] and Charm4Py [5] that provide a thin Python wrapper around MPI and Charm++. These libraries follow the semantics of the corresponding C libraries and demand considerable programming effort to implement parallel applications.

6 Future Work

There are several planned improvements for CharmStencil aimed at enhancing both the usability and performance of the abstraction. By allocating boundary

layers separately from the internal grid, we can avoid copying boundaries into a contiguous memory buffer to send ghost data at every iteration, thus reducing the message packing cost incurred in each iteration.

While we demonstrate weak scaling performance comparable to a hand-written Charm++ implementation, strong scaling performance can still be enhanced by reducing the Python overhead and communication cost between the frontend and the backend. This can be achieved by introducing a loop construct in the AST, avoiding time-stepping iterations on the frontend. Instead, the outer loop can be encoded in the AST, sent to the backend only once, with minimal overhead on the frontend.

A departure from Numpy in our current implementation is the necessity of the `exchange_ghosts` call. Inferring ghost exchanges from array access patterns in the generated AST would significantly improve the abstraction's usability.

In addition to these optimizations, we plan to add the ability to check for convergence on the backend. We also plan to use the same frontend AST to generate CUDA code on the backend and leverage Charm++ HAPI to support execution on multiple GPUs.

7 Conclusion

Numpy has been extensively utilized for prototyping scientific computing applications. However, parallelizing these applications demands significant programming effort. While drop-in Numpy replacements exist to run the same applications on parallel machines, these alternatives often suffer from high overheads or necessitate a substantial rewrite of the source code. In this project, we introduced a highly scalable library designed for stencil computations on distributed memory machines using a Python frontend and a Charm++ backend, requiring users to make minimal changes to their sequential Numpy code. Our implementation was compared to two state-of-the-art Numpy alternatives, showcasing superior scaling performance.

References

1. Bauer, M., Garland, M.: Legate NumPy: accelerated and distributed array computing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019. Association for Computing Machinery, New York (2019)
2. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: the best of both worlds. Comput. Sci. Eng. **13**(2), 31–39 (2011)
3. Bradbury, J., et al.: JAX: composable transformations of Python+NumPy programs (2018)
4. Dalcín, L., Paz, R., Storti, M.: MPI for Python. J. Parallel Distrib. Comput. **65**(9), 1108–1115 (2005)
5. Galvez, J.J. Senthil, K., Kale, L.: CharmPy: a Python parallel programming model. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 423–433 (2018)

6. Harris, C.R., et al.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020)
7. Kalé, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Paepcke, A. (ed.) Proceedings of OOPSLA 1993, pp. 91–108. ACM Press (1993)
8. Lam, S.K., Pitrou, A., Seibert, S.: Numba: a LLVM-based python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, pp. 1–6 (2015)
9. Merrill, M., Reus, W., Neumann, T.: Arkouda: interactive data exploration backed by chapel. In: Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop, CHIUW 2019, p. 28. Association for Computing Machinery, New York (2019)
10. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling. In: SciPy (2015)
11. Virtanen, P., et al.: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat. Methods **17**, 261–272 (2020)