This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MCSE.2021.3083216, Computing in Science & Engineering

mpi4py: Status Update After 12 Years of Development

Lisandro Dalcin
Extreme Computing Research Center (ECRC), King Abdullah University of Science and Technology (KAUST), Thuwal 23955, Saudi Arabia

Yao-Lung L. Fang
Computational Science Initiative, Brookhaven National Laboratory, Upton, NY 11973, USA

***Abstract*—MPI for Python (mpi4py) has evolved to become the most used Python binding for the Message Passing Interface (MPI). We report on various improvements and features that mpi4py gradually accumulated over the past decade, including support up to the MPI-3.1 specification, support for CUDA-aware MPI implementations, and other utilities at the intersection of MPI-based parallel distributed computing and Python application development.**

THE MESSAGE PASSING INTERFACE (MPI) is a standardized application program interface (API) targeting parallel distributed computing and featuring a message-passing paradigm for interprocess communication. The MPI standard defines the syntax and semantics of library routines and allows users to write portable parallel applications in the mainstream scientific programming languages (C/C++ and Fortran). Since its initial release in 1994, the MPI specification took parallel distributed application development by storm to quickly become the leading paradigm for large-scale high-performance computing (HPC). We regard MPI dominance as still uncontested. Quality implementations are available from leading open-source projects like MPICH[1] and Open MPI,[2] as well as from software and hardware vendors of HPC systems.

In the quest for raw performance, high-performance computing is traditionally associated with software development using compiled languages like C/C++ and Fortran. These languages are relatively lower-level, stay close to the bare metal, and have years of accumulated experience in code optimization. Since the inception of MPI, scientific computing has experienced a gradual paradigm shift. Although performance remains a must, market forces put programmer productivity in the spotlight as industry demanded faster prototype-to-production cycles. During that time, the Python programming language grew its user base slowly but steadily and eventually became one of the most popular programming languages. Much of Python's popularity is due to the seminal NumPy project[3] and its multidimensional array data structure, which fueled Python's success in a wide variety of applications in data science, machine learning, scientific computing, and data visualization. Since its onset, Python was designed to interoperate with C codes and libraries using extension modules. This feature spurred the development of Python wrappers to popular libraries and specialized tools to ease such wrapping process.

mpi4py[4] was not the first attempt to bring MPI and Python together. However, since its humble beginnings, mpi4py strove to provide Python bindings to MPI following two main principles: a)

[1] https://www.mpich.org
[2] https://www.open-mpi.org
[3] https://numpy.org
[4] https://mpi4py.github.io

being feature-complete and expose to Python as much of MPI as possible, and b) staying close to the MPI standard in both syntax and semantics, without reinventing the wheel with new or foreign APIs, and maximizing user convenience by following common Python idioms and practice.

This article presents improvements and new features added to mpi4py since the last report in the literature in 2008 [1]. The material is organized as follows. First, we review some generalities about mpi4py to set the stage for subsequent discussions. Next, we comment on the codebase modernization mpi4py underwent before its 1.0 release. Afterward, we present our support for the latest revision of the MPI-3 standard.[5] With the rise of GPU (graphic processing unit) programming and CUDA-aware MPI implementations, we discuss how this model is supported in mpi4py. Following that, we present some recent mpi4py features, such as a new package for asynchronous task execution and support for efficient pickle-based communication of large-size buffer-like Python objects. Finally, we mention a few applications using mpi4py and close with conclusions and plans for future work.

Overview

mpi4py adopts an object-oriented API, which is natural in Python. This API is based on the MPI C++ bindings, focusing on staying as close as possible to MPI syntax and semantics.

The MPI-1 standard introduced core concepts like process groups, communication domains, and point-to-point and collective communication. The MPI-2 standard extended the feature set with one-sided operations, dynamic process management, and parallel I/O. The MPI-3 standard contributed refinements and extensions to the repertoire. mpi4py supports almost all MPI features and targets complete MPI-3.1 implementations. Nonetheless, the Python bindings can also work with legacy MPI-1 or MPI-2 implementations.

mpi4py supports point-to-point and collective communication of generic Python objects by taking advantage of Python's pickle protocol.[6] This approach involves serialization (deserialization) before (after) issuing any MPI communication operation. Obviously, the pickle-based communication mode necessarily involves CPU and memory overheads. However, its simplicity and convenience may offset any performance penalty, particularly when Python objects with small memory footprints are involved.

mpi4py also features efficient communication of array-like Python objects via Python's buffer protocol.[7] This buffer-based communication mode has negligible overhead. As the implementation involves forwarding memory buffers to the MPI routines, the performance of user codes is on par with equivalent C/C++ or Fortran codes. Besides point-to-point and collective communication operations, Python's buffer protocol is the primary mechanism behind mpi4py's support for one-sided operations and parallel I/O.

Codes using mpi4py usually start with a `from mpi4py import MPI` import statement. The `MPI` module exposes many functions, classes, and methods giving access to the MPI API. The most important components in the `MPI` module are the `Comm`, `Win`, and `File` classes, whose various methods support point-to-point and collective communication, one-sided operations, and parallel I/O, respectively.

Cython implementation

In its early days, before release 1.0 in 2009, mpi4py featured a dual-language implementation. A low-level extension module provided a bridge between the Python language and the MPI library, using hand-written C code calling into the Python C API and the MPI C bindings. Higher-level pure Python code used the low-level C extension module to define and implement the various attributes, functions, classes, and methods that mpi4py exposed to users. Such an implementation approach was cumbersome to extend and maintain, thus hindering evolution.

In 2007, the Cython project [2] emerged, featuring a static Python-to-C compiler with extra language extensions. Besides the obvious potential for speeding up Python code via C translation, Cython allowed for single-language development of object-oriented, pythonic module interfaces with the ability to perform direct invocations to external C functions. This distinctive feature

[5] https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
[6] https://docs.python.org/3/library/pickle.html
[7] https://docs.python.org/3/c-api/buffer.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MCSE.2021.3083216, Computing in Science & Engineering

aligned perfectly with mpi4py's needs. As a result, mpi4py's main author joined the Cython project in its initial development stages, focusing on improving the Cython language and compiler's feature set in aspects related to the development of Python wrappers for external C APIs.

Eventually, mpi4py was rewritten in Cython from scratch. The new Cython-based implementation was released in 2009 as version 1.0. This rewrite made the codebase far easier to extend and maintain, thus lowering the bar for external contributions and enhancements to come.

MPI-3 features

mpi4py release 2.0 in late 2015 brought an update addressing the new features added in the MPI-3 specification. These features include matched probe and receive, nonblocking and neighborhood collective operations, and revamped one-sided operations.

Matched probe and receive

The MPI-1 standard defined a probe mechanism, allowing for inquiring incoming messages without actually receiving them. In mpi4py, users would call the `Probe()` or `Iprobe()` methods of the `Comm` class and then act on the information returned in a `Status` instance. For example, users could query the size of the incoming message and then perform buffer allocation accordingly to prepare for the subsequent `Recv()` operation. In fact, this probe-and-receive approach allowed mpi4py to implement its support for point-to-point communication of generic Python objects via pickle serialization with a single MPI message.

Unfortunately, the MPI-1 probe mechanism is not thread-safe. The MPI-3 standard acknowledged this deficiency and introduced a new matching mechanism to ensure race-free probe-and-receive, even when multiple threads operating on the same communicator context execute concurrently. In mpi4py, users would call `Mprobe()` method of the `Comm` class, which returns a `Message` instance. Afterward, the `Mrecv()` method of the `Message` class should be invoked to receive the actual message content. A nonblocking version named with an "`I`" prefix is also available. This MPI-3 enhancement was instrumental to mpi4py in re-implementing pickle-based communication, allowing for thread-safe receive operations of generic Python objects while maintaining the efficiency of a single MPI message implementation.

Nonblocking collectives

MPI-3 introduced nonblocking collective operations to allow for the overlap of computation and communication. The set of conventional blocking collectives were augmented with a new set of siblings operations named with an "`I`" prefix, such as `Ibarrier()`, `Ibcast()`, `Iallgather()`, `Ialltoall()`, and `Ireduce()`. Like other nonblocking operations in MPI, they return a `Request` instance to be used for subsequent completion with calls to the `Wait()` or `Test()` methods.

Neighborhood Collectives

The MPI-3 standard introduced a new kind of collective operation called neighborhood collectives. These new operations simplify data exchange in stencil-like computation patterns common in partial differential equation (PDE) solvers. The sparse communication pattern is expressed via "topology-aware" (Cartesian or graph) communicators.

Only two collective operations have a neighborhood counterpart: all-to-all and all-gather. mpi4py supports both blocking (`Neighbor_alltoall()` and `Neighbor_allgather()`) and nonblocking (with an "`I`" prefix) neighborhood collectives, as well as variable-count versions (with a "`v`" suffix). Additionally, a generalized neighborhood all-to-all (with a "`w`" suffix) operation is also available.

Improved one-sided operations

The MPI-3 standard added three new atomic one-sided operations, allowing for more flexibility and optimization opportunities. mpi4py exposes these new features as methods `Get_accumulate()`, `Fetch_and_op()`, and `Compare_and_swap()` of the `Win` class.

Another new feature introduced in MPI-3 is the request-based one-sided operations. Some one-sided operations have a new counterpart with an "R" prefix, indicating that a `Request` object is returned after the call; these operations

include `Rput()`, `Rget()`, `Raccumulate()`, and `Rget_accumulate()` as methods of the `Win` class. These request-based one-sided operations allow for an operate-and-query/wait pattern similar to nonblocking point-to-point and collective communication.

Finally, there are three new window types: MPI-allocated windows (created with the factory method `Allocate()` of the `Win` class), shared-memory windows (created with `Allocate_shared()`), and dynamic windows (created with `Create_dynamic()`). Memory for dynamic windows can be attached or detached by the `Attach()` and `Detach()` methods, respectively.

CUDA-aware MPI

In recent years, GPU programming – in particular, CUDA programming for NVIDIA GPUs – has become indispensable in many HPC and machine learning applications. While the MPI standard does not cover concepts like device offloading, nor does it specify the semantics of interacting with GPUs, *CUDA-aware* MPI has become a *de facto* extension of the standard. In a nutshell, most of the MPI communication routines accept pointers to device memory such that users do not have to copy data back and forth between GPU and CPU. Besides the minor coding convenience of eliminating data copy steps, direct device-to-device communication can offer substantial speedups, as shown in Figure 1. Several major MPI implementations, including MPICH, Open MPI, MVAPICH,[8] and Spectrum MPI,[9] already feature CUDA awareness.

To facilitate API interoperability and zero-copy GPU data exchange, several Python GPU libraries jointly define and implement the CUDA Array Interface (CAI) protocol,[10] an effort that originated after the initiative of the Numba project.[11] The CAI largely follows the NumPy array interface protocol[12] and requires that all compliant objects add a new Python attribute `__cuda_array_interface__`, containing the raw GPU buffer address and additional metadata. This way, the attribute filled by a producer can be correctly interpreted by a consumer. The CAI has been implemented by many Python libraries featuring GPU computing, such as Numba, CuPy,[13] PyTorch,[14] cuDF,[15] PyArrow,[16] and PyCUDA.[17]

We have collaborated with the community in the revisions of this protocol and implemented its support in mpi4py, making it straightforward to perform "MPI+CUDA programming" in Python. The use of this feature requires that a) mpi4py is built with a CUDA-aware MPI library and b) Python GPU array objects involved in MPI communication are compliant with the CAI protocol. We note in particular that mpi4py is on itself *CUDA-unaware*. Whether an MPI routine can handle GPU arrays depends entirely on the underlying MPI implementation; mpi4py simply follows the CAI protocol and forwards device pointers to the MPI implementation. CUDA-unawareness in mpi4py has an important consequence: to comply with the usual MPI semantics, a GPU array must be ready (by synchronizing over the CUDA stream on which the array is being prepared/processed) before performing any MPI operation with it.

Additional features

Asynchronous task execution

Implementing the master-worker pattern within embarrassingly parallel task-based applications has been a recurrent matter among mpi4py users. Despite its conceptual simplicity, a production-ready MPI-based master-worker implementation poses some difficulties, particularly for users with limited exposure to MPI programming. mpi4py release 3.0 in late 2017 featured a new `mpi4py.futures` package providing a framework for the asynchronous execution of tasks on a pool of worker processes using MPI for inter-process communication.

The `mpi4py.futures` API is high-level and user-friendly, without requiring

[8] http://mvapich.cse.ohio-state.edu
[9] https://www.ibm.com/products/spectrum-mpi
[10] https://numba.readthedocs.io/en/stable/cuda/cuda_array_interface.html
[11] https://numba.pydata.org
[12] https://numpy.org/doc/stable/reference/arrays.interface.html
[13] https://cupy.dev
[14] https://pytorch.org
[15] https://docs.rapids.ai/api/cudf/stable/
[16] https://arrow.apache.org
[17] https://documen.tician.de/pycuda/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MCSE.2021.3083216, Computing in Science & Engineering

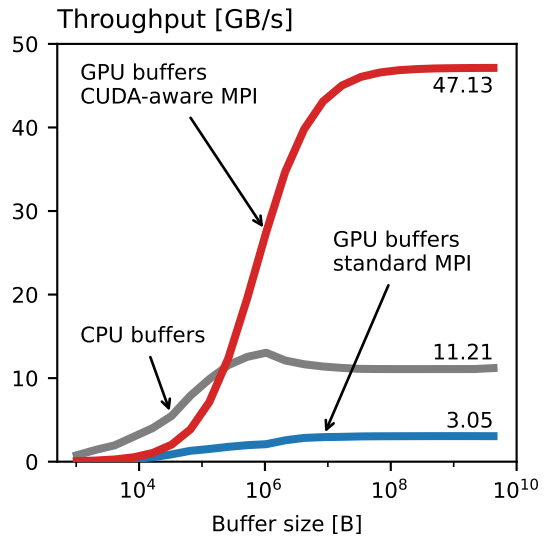

Figure 1. Performance of mpi4py with CUDA-aware MPI. Two Python MPI processes send to and receive from each other a series of CuPy arrays (GPU), with sizes ranging from 1 KiB to 4 GiB, and using mpi4py buffer-based communication with either CUDA-aware MPI or standard MPI with host-device memory copies. To establish a baseline for comparison, we also consider a CPU version of this test using NumPy arrays. From wall-clock time measurements (average of many samples), we determine the effective throughput (gigabytes per second) for each case. The gray line shows the baseline throughput for CPU buffer communication. The blue line shows the significant penalty from the additional host-to-device and device-to-host data transfers required when using a standard MPI. The red line shows the substantial speedup achievable with CUDA-aware MPI for device-to-device data transfers (up to ≈15X for large data sizes).

We executed these performance tests on a workstation running Ubuntu 18.04.4, with Python 3.8.8, Open MPI 4.1.1, CUDA 11.0, CuPy 9.0.0, and all other software components installed from https://conda-forge.org. We used the in-development version of mpi4py. The hardware consisted of a 10-core Intel® Core™ i9-9820X 3.30 GHz CPU, 64 GiB 2666 MT/s DDR4 RAM, and two NVIDIA® GeForce RTX™ 2080 Ti interconnected with an NVIDIA® NVLink™ Bridge providing a nominal maximum bidirectional bandwidth of 50 GB/s. Each MPI process is bound to a different GPU.

the explicit use of any MPI primitives. `mpi4py.futures` is heavily based on the `concurrent.futures` module[18] from the Python standard library. More precisely, `mpi4py.futures` provides the `MPIPoolExecutor` class as a concrete implementation of the abstract `concurrent.futures.Executor` class. The `submit()` method schedules a Python callable (and any companion arguments) to be executed asynchronously and returns a `Future` object representing the outcome of the execution of the callable. At a later time, the `Future` object is queried for the execution result (in case of success) or exception (in case of failure). A set of `Future` objects can be passed to the `wait()` and `as_completed()` functions. The `map()` interface is also supported.

By performing computations in separate processes, `mpi4py.futures` allows to sidestep Python's infamous *global interpreter lock* (GIL). By relying on MPI for the wire-up between processes, `mpi4py.futures` provides a simple and convenient alternative for scaling up applications using `concurrent.futures`: MPI is a pre-configured and ready-to-use framework in almost all supercomputing facilities.

The implementation of `mpi4py.futures` takes advantage of MPI dynamic process management, particularly the capabilities to spawn new MPI processes at runtime. After spawning a new set of worker processes, the master process uses a separate thread to communicate back and forth with the workers. Worker processes dedicate exclusively to execute the tasks submitted by the master until they are signaled for completion.

Legacy MPI-1 implementations and some vendor MPI-2 (or even MPI-3) implementations do not support the dynamic process management features introduced in the MPI-2 standard. Additionally, job schedulers and batch systems in supercomputing facilities may pose additional restrictions or cumbersome extra configuration steps to applications requiring MPI process spawning at runtime. With these issues in mind, `mpi4py.futures` supports an additional, more traditional, SPMD-like (single program, multiple data) usage pattern that

[18] https://docs.python.org/3/library/concurrent.futures.html

makes use of MPI-1 features only. User code is launched in parallel the usual way, e.g., using the `mpiexec` command. Afterward, MPI processes collectively enter a Python context manager set up by the `MPICommExecutor` class. This context manager partitions the set of processes within an existing MPI communicator (usually `COMM_WORLD`) into one master process and many worker processes. The master process is given access to an `MPIPoolExecutor` instance to submit tasks. Meanwhile, the worker processes follow a different execution path and team up to execute the tasks submitted by the master. Besides sidestepping the lack of dynamic process management features in legacy or incomplete MPI implementations, the `MPICommExecutor` context manager may also be useful in classic MPI-based Python applications that can take advantage of the simple master/worker approach available through the `mpi4py.futures` module.

Support for large-size objects

The in-development version of mpi4py[19], scheduled to be released as version 3.1 shortly, includes new facilities for efficient pickle-based communication of large-size Python objects.

As mentioned earlier, mpi4py support for the communication of generic Python objects relies on pickling. The pickle protocol dates back to 1995, and its original and primary concern was supporting on-disk persistence of arbitrary Python objects. Over time, the pickle protocol evolved in successive new versions and found uses in applications related to multi-process parallelism and distributed computing as an essential ingredient for message exchange. Many of these applications, particularly those related to data science, suffered from a longstanding issue: the pickle protocol required multiple and redundant memory copies, thus degrading performance when communicating array-like objects with large memory footprints. As a result, PEP 574 proposed a new pickle protocol version 5[20] which introduces support for out-of-band buffers, allowing for more efficient handling of objects with large memory footprints.

mpi4py uses the traditional in-band handling of buffers. This approach is appropriate for communicating non-buffer Python objects or buffer-like objects with small memory footprints. For point-to-point communication, in-band buffer handling allows for the communication of a pickled stream with a single MPI message at the expense of additional CPU and memory overhead in the pickling and unpickling steps. Communicating pickled streams with out-of-band buffers necessarily involves multiple MPI messages, thus increasing latency and hurting performance for small-size data. However, the zero-copy savings of out-of-band buffer handling can easily outweigh the extra latency costs for large-size data.

The new `mpi4py.util.pkl5` module provides communicator wrapper classes re-implementing the pickle-based point-to-point communication methods using pickle protocol version 5. Using this feature is as simple as wrapping an existing communicator object, either `MPI.Intracomm` or `MPI.Intercomm`, using the `pkl5.Intracomm` or `pkl5.Intercomm` wrapper classes, respectively. These wrapper classes redefine most pickle-based point-to-point communication methods to use pickle protocol 5 and out-of-band buffers. Moreover, these wrapper methods can overcome the infamous $2^{31}-1$ ($\approx$ 2 billion) MPI message count limit by following the approaches described by Hammond et al. [3].

Figure 2 shows the substantial performance improvement of using out-of-band buffers versus the traditional in-band approach.

Applications

Over the years, mpi4py has become the *de facto* standard choice for creating MPI-based applications in Python. In the following, we mention a few out of the many applications that depend on mpi4py.

HDF5 for Python (h5py)[21] is the Python wrapper for the HDF5 library, allowing reading and writing HDF5 files with a convenient and pythonic interface. HDF5 optionally supports parallel MPI I/O. At the Python level, mpi4py provides the bridge for h5py to interact with the MPI environment.

The Visualization Toolkit (VTK)[22] is an open-source library for image processing and

[19] https://github.com/mpi4py/mpi4py
[20] https://www.python.org/dev/peps/pep-0574/
[21] https://www.h5py.org
[22] https://vtk.org

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MCSE.2021.3083216, Computing in Science & Engineering

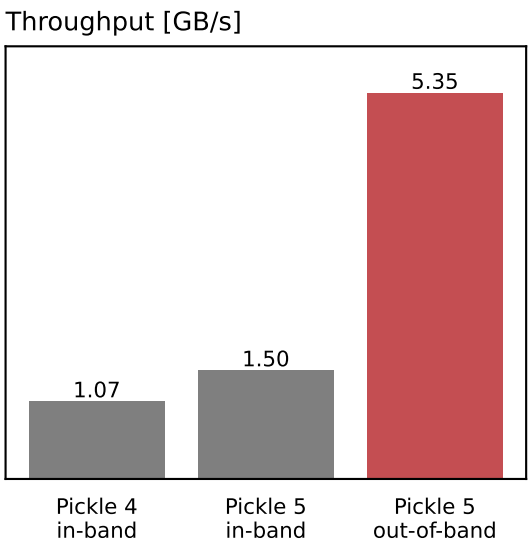

Figure 2. Performance of mpi4py pickle-based communication. Two Python MPI processes send to and receive from each other NumPy arrays with a fixed size of 1.0 GiB using mpi4py pickle-based communication. We determine the effective throughput (gigabytes per second) from wall-clock time measurements (average of ten samples). mpi4py pickle-based communication uses either pickle protocol version 4 or 5 with the traditional in-band handling buffers (grey-colored columns at the left and center). Even with in-band buffers, the newer protocol version 5 allows for faster (≈ 40%) inter-process data transfers compared to the older protocol version 4. The new `mpi4py.util.pkl5` module uses protocol version 5 with out-of-band buffers (red-colored column at the right). The use of out-of-band buffers allows for a substantial speedup (≈ 3.5X) in effective throughput. Software and hardware are the same as described in Figure 1. The Linux kernel was configured at runtime with transparent hugepage support (THP) enabled to "always".

visualization. It includes mpi4py as one of the many third-party libraries exposed as VTK modules for internal use. VTK defines a `vtkMPI4PyCommunicator` class exposing a couple of routines able to convert back and forth between mpi4py and VTK communicator instances and enabling seamless interaction between mpi4py and other VTK components.

yt[23] is an open-source toolkit for analyzing and visualizing simulation data in science and engineering. It provides a high-level abstraction to enable MPI parallelism over the built-in data structures and algorithms by simply calling `yt.enable_parallelism()` at the start of a script.

Dask[24] is a flexible library for parallel computing in Python, which is widely used in data science and HPC projects. It consists of a dynamic task scheduler and parallel data structures and algorithms. The `dask.distributed` module provides a way to configure the distributed worker processes. The `dask_mpi` module[25] builds on top of mpi4py and `dask.distributed`, and provides a bridge for the Dask scheduler (running on MPI rank 0) to interact with the rest of the MPI processes acting as Dask workers. MPI4Dask [4] is planned to replace `dask_mpi` and to be integrated into `dask.distributed` as a communication backend.

mpi4py-fft [5] is a Python package for distributed, multi-dimensional Fast Fourier Transforms (FFTs) [6], which is used in shenfun [7] to solve PDEs using the spectral Galerkin method and in FluidFFT [8] as a backend option to support a unified API for parallel FFT computations.

A challenging aspect of 2D (ptychographic) and 3D (tomographic or ptycho-tomographic) image reconstruction is that these applications are data- and computation-intensive. MPI parallelization is critical in reducing memory stress and time to solution. Several image reconstruction packages developed in the computational microscopy community depend on mpi4py for parallel computing, including nsls2ptycho [9], Adorym [10], Tike,[26] PyNX [11], and XPACK [12]. In a recent version of nsls2ptycho [9], the GPU support is changed from PyCUDA to CuPy, allowing a unified CPU/GPU codebase thanks to CuPy's high compatibility with NumPy, including the support for the `__array_function__`

[23] https://yt-project.org/
[24] https://dask.org/
[25] https://mpi.dask.org/en/latest/
[26] https://tike.readthedocs.io

protocol.[27] However, if a performance bottleneck on the GPU code path is identified, the hotspot can be easily re-implemented using `cupy.ElementwiseKernel` or `cupy.RawKernel` (depending on the type of operations needed) to reach native CUDA C++ speed. For CPU (NumPy) arrays, a simple MPI collective call (depending on the algorithm of choice) to `Comm.Allreduce()` is sufficient. For GPU (CuPy) arrays, there are multiple choices of communication paths: the slowest one is to copy the GPU data to and from CPU for communication, but it is usually faster to launch the MPI collective directly on a GPU array if a CUDA-aware MPI is available.

Conclusions

mpi4py has evolved over the years, following the footsteps of upstream efforts closely. To date, mpi4py provides MPI coverage up to the latest MPI-3.1 standard, including matched probe-and-receive, nonblocking collectives, neighborhood collectives, and new one-sided operations. mpi4py has also kept a close watch on Python evolution, providing the new `mpi4py.futures` package for easing task-based computations, and the recent support for pickle protocol version 5 and efficient communication of array-like objects with large memory footprints. Following industry trends like CUDA-aware MPI and the CAI protocol, mpi4py supports current and future applications in the never-ending quest for performance.

Following community guidelines,[28] we plan on dropping support for Python 2 and early Python 3 releases. We also strive to keep providing pre-built binary packages for easier installation. With the ongoing community endeavor to standardize the Python array API,[29] we plan to support the new DLPack-based data exchange protocol,[30] allowing for vendor-neutral GPU-aware MPI. The upcoming MPI-4.0 standard will be a major and welcome update adding new features and fixing longstanding issues like the 2 billion message count limit. We plan on supporting new MPI-4 features once this new version of the standard is implemented in major MPI implementations.

[27] https://numpy.org/neps/nep-0018-array-function-protocol.html

[28] https://numpy.org/neps/nep-0029-deprecation_policy.html

[29] https://data-apis.org/array-api/latest/

[30] The DLPack data structure is specified in https://github.com/dmlc/dlpack, and the `__dlpack__` protocol based on the DLPack struct is defined as part of the Python array API v1.

Acknowledgments

We want to thank all the feedback, comments, and contributions received from users over the years. We also acknowledge the collaboration with upstream MPI developers and maintainers: we thank Hui Zhou from MPICH, Jeff Squyres from Open MPI, and Hari Subramoni and Ching-Hsiang Chu from MVAPICH. We thank Graham Markall for leading the community effort to revise and improve the CAI protocol. LD acknowledges the KAUST Supercomputing Laboratory for the use of its resources and the continuous help and assistance from Bilel Hadri. YLLF acknowledges ATPESC 2018 for the intensive exposure to modern HPC tools and practices, and the Advanced Computing Lab[31] for providing access to the NVIDIA DGX-2 and DGX-A100 Artificial Intelligence Cluster for testing.

LD is supported by ECRC at KAUST. YLLF is supported in part by BNL LDRD 17-029 and 20-024 and DOE Grant No. BES-FWP-PS001.

REFERENCES

1. L. Dalcin, R. Paz, M. Storti, and J. D'Elia, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, 2008. [Online]. Available: https://doi.org/10.1016/j.jpdc.2007.09.005
2. S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011. [Online]. Available: https://doi.org/10.1109/MCSE.2010.118
3. J. R. Hammond, A. Schäfer, and R. Latham, "To INT_MAX... and beyond! exploring large-count support in MPI," in *2014 Workshop on Exascale MPI at Supercomputing Conference*. IEEE, 2014, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ExaMPI.2014.5
4. A. Shafi, J. M. Hashmi, H. Subramoni, and D. K. Panda, "Efficient MPI-based Communication

[31] The Advanced Computing Lab is a component of the Computational Science Initiative at Brookhaven National Laboratory, supported by the U.S. Department of Energy's Office of Science under Contract No. DE-SC0012704.

for GPU-Accelerated Dask Applications," 2021, arXiv:2101.08878.

5. M. Mortensen, L. Dalcin, and D. E. Keyes, "mpi4py-fft: Parallel Fast Fourier Transforms with MPI for Python," *Journal of Open Source Software*, vol. 4, no. 36, p. 1340, 2019. [Online]. Available: https://doi.org/10.21105/joss.01340
6. Dalcin, Lisandro and Mortensen, Mikael and Keyes, David E, "Fast parallel multidimensional FFT using advanced MPI," *Journal of Parallel and Distributed Computing*, 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.02.006
7. M. Mortensen, "Shenfun: High performance spectral Galerkin computing platform," *Journal of Open Source Software*, vol. 3, no. 31, p. 1071, 2018. [Online]. Available: https://doi.org/10.21105/joss.01071
8. A. V. Mohanan, C. Bonamy, and P. Augier, "FluidFFT: Common API (C++ and Python) for Fast Fourier Transform HPC Libraries," *Journal of Open Research Software*, vol. 7, 2019. [Online]. Available: https://doi.org/10.5334/jors.238
9. Z. Dong, Y.-L. L. Fang *et al.*, "High-Performance Multi-Mode Ptychography Reconstruction on Distributed GPUs," in *2018 New York Scientific Data Summit (NYSDS)*, 2018, pp. 1–5. [Online]. Available: https://doi.org/10.1109/NYSDS.2018.8538964
10. M. Du, S. Kandel, J. Deng, X. Huang, A. Demortiere, T. T. Nguyen, R. Tucoulou, V. De Andrade, Q. Jin, and C. Jacobsen, "Adorym: a multi-platform generic X-ray image reconstruction framework based on automatic differentiation," *Optics Express*, vol. 29, no. 7, pp. 10 000–10 035, 2021. [Online]. Available: https://doi.org/10.1364/OE.418296
11. V. Favre-Nicolin, G. Girard, S. Leake, J. Carnis, Y. Chushkin, J. Kieffer, P. Paleo, and M.-I. Richard, "*PyNX*: high-performance computing toolkit for coherent X-ray imaging based on operators," *Journal of Applied Crystallography*, vol. 53, no. 5, pp. 1404–1413, Oct 2020. [Online]. Available: https://doi.org/10.1107/S1600576720010985
12. S. Marchesini, A. Trivedi, P. Enfedaque, T. Perciano, and D. Parkinson, "Sparse Matrix-Based HPC Tomography," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 248–261. [Online]. Available: https://doi.org/10.1007/978-3-030-50371-0_18

Lisandro Dalcin is a Senior Research Scientist at King Abdullah University of Science and Technology (KAUST). He received his Ph.D. in Engineering Sciences and Computational Mechanics from National University of the Littoral (Argentina) in 2008. He held an Adjunct Research Associate position at National Scientific and Technical Research Council (Argentina) from 2010 to 2017. He is the main author and maintainer of various Python packages related to high-performance parallel distributed computing (mpi4py, petsc4py, slepc4py) and a member of the development team of PETSc. His research interests include numerical methods in computational fluid mechanics, scalable solution methods for PDEs, and parallel computing on distributed memory architectures. Contact him at dalcinl@gmail.com.

Yao-Lung (Leo) Fang is an Assistant Scientist at Brookhaven National Laboratory, New York, USA. He received his B.S. in physics from National Taiwan University in 2009 and his Ph.D. in physics from Duke University in 2017. He is a theoretical physicist by training, specialized in quantum optics and open quantum systems. After joining BNL, his research interests gradually shifted towards high-performance computing and quantum computing. He is best known for his works in waveguide quantum electrodynamics in the non-Markovian regime and GPU acceleration for ptychography. He is a regular contributor to several efforts within the Python scientific computing ecosystem and the CuPy project. He has mentored several GPU Hackathon teams for Python GPU programming and is a member of the Consortium for Python Data API Standards. Contact him at leofang@bnl.gov.