# MPI for Python

***Release 3.1.0a0***

**Lisandro Dalcin**

**Sep 29, 2019**

## Contents

**Abstract**

This document describes the *MPI for Python* package. *MPI for Python* provides bindings of the *Message Passing Interface* (MPI) standard for the Python programming language, allowing any Python program to exploit multiple processors.

This package is constructed on top of the MPI-1/2/3 specifications and provides an object oriented interface which resembles the MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of any *picklable* Python object, as well as optimized communications of Python object exposing the single-segment buffer interface (NumPy arrays, builtin bytes/string/array objects)

# 1 Introduction

Over the last years, high performance computing has become an affordable resource to many more researchers in the scientific community than ever before. The conjunction of quality open source software and commodity hardware strongly influenced the now widespread popularity of Beowulf class clusters and cluster of workstations.

Among many parallel computational models, message-passing has proven to be an effective one. This paradigm is specially suited for (but not limited to) distributed memory architectures and is used in today's most demanding scientific and engineering application related to modeling, simulation, design, and signal processing. However, portable message-passing parallel programming used to be a nightmare in the past because of the many incompatible options developers were faced to. Fortunately, this situation definitely changed after the MPI Forum released its standard specification.

High performance computing is traditionally associated with software development using compiled languages. However, in typical applications programs, only a small part of the code is time-critical enough to require the efficiency of compiled languages. The rest of the code is generally related to memory management, error handling, input/output, and user interaction, and those are usually the most error prone and time-consuming lines of code to write and debug in the whole development process. Interpreted high-level languages can be really advantageous for this kind of tasks.

For implementing general-purpose numerical computations, MATLAB[1] is the dominant interpreted programming language. In the open source side, Octave and Scilab are well known, freely distributed software packages providing

[1] MATLAB is a registered trademark of The MathWorks, Inc.

compatibility with the MATLAB language. In this work, we present MPI for Python, a new package enabling applications to exploit multiple processors using standard MPI "look and feel" in Python scripts.

## 1.1 What is MPI?

MPI, [mpi-using] [mpi-ref] the *Message Passing Interface*, is a standardized and portable message-passing system designed to function on a wide variety of parallel computers. The standard defines the syntax and semantics of library routines and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++).

Since its release, the MPI specification [mpi-std1] [mpi-std2] has become the leading standard for message-passing libraries for parallel computers. Implementations are available from vendors of high-performance computers and from well known open source projects like MPICH [mpi-mpich] and Open MPI [mpi-openmpi].

## 1.2 What is Python?

Python is a modern, easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming with dynamic typing and dynamic binding. It supports modules and packages, which encourages program modularity and code reuse. Python's elegant syntax, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. It is easily extended with new functions and data types implemented in C or C++. Python is also suitable as an extension language for customizable applications.

Python is an ideal candidate for writing the higher-level parts of large-scale scientific applications [Hinsen97] and driving simulations in parallel architectures [Beazley97] like clusters of PC's or SMP's. Python codes are quickly developed, easily maintained, and can achieve a high degree of integration with other libraries written in compiled languages.

## 1.3 Related Projects

As this work started and evolved, some ideas were borrowed from well known MPI and Python related open source projects from the Internet.

- OOMPI
  - It has not relation with Python, but is an excellent object oriented approach to MPI.
  - It is a C++ class library specification layered on top of the C bindings that encapsulates MPI into a functional class hierarchy.
  - It provides a flexible and intuitive interface by adding some abstractions, like *Ports* and *Messages*, which enrich and simplify the syntax.
- Pypar
  - Its interface is rather minimal. There is no support for communicators or process topologies.
  - It does not require the Python interpreter to be modified or recompiled, but does not permit interactive parallel runs.
  - General (*picklable*) Python objects of any type can be communicated. There is good support for numeric arrays, practically full MPI bandwidth can be achieved.
- pyMPI

  - It rebuilds the Python interpreter providing a built-in module for message passing. It does permit interactive parallel runs, which are useful for learning and debugging.
  - It provides an interface suitable for basic parallel programing. There is not full support for defining new communicators or process topologies.
  - General (picklable) Python objects can be messaged between processors. There is not support for numeric arrays.
- Scientific Python
  - It provides a collection of Python modules that are useful for scientific computing.
  - There is an interface to MPI and BSP (*Bulk Synchronous Parallel programming*).
  - The interface is simple but incomplete and does not resemble the MPI specification. There is support for numeric arrays.

Additionally, we would like to mention some available tools for scientific computing and software development with Python.

- NumPy is a package that provides array manipulation and computational capabilities similar to those found in IDL, MATLAB, or Octave. Using NumPy, it is possible to write many efficient numerical data processing applications directly in Python without using any C, C++ or Fortran code.
- SciPy is an open source library of scientific tools for Python, gathering a variety of high level science and engineering modules together as a single package. It includes modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms, ODE solvers, and others.
- Cython is a language that makes writing C extensions for the Python language as easy as Python itself. The Cython language is very close to the Python language, but Cython additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code. This makes Cython the ideal language for wrapping for external C libraries, and for fast C modules that speed up the execution of Python code.
- SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages like Perl, Tcl/Tk, Ruby and Python. Issuing header files to SWIG is the simplest approach to interfacing C/C++ libraries from a Python module.

# 2 Overview

MPI for Python provides an object oriented approach to message passing which grounds on the standard MPI-2 C++ bindings. The interface was designed with focus in translating MPI syntax and semantics of standard MPI-2 bindings for C++ to Python. Any user of the standard C/C++ MPI bindings should be able to use this module without need of learning a new interface.

## 2.1 Communicating Python Objects and Array Data

The Python standard library supports different mechanisms for data persistence. Many of them rely on disk storage, but *pickling* and *marshaling* can also work with memory buffers.

The `pickle` modules provide user-extensible facilities to serialize general Python objects using ASCII or binary formats. The `marshal` module provides facilities to serialize built-in Python objects using a binary format specific to Python, but independent of machine architecture issues.

*MPI for Python* can communicate any built-in or user-defined Python object taking advantage of the features provided by the `pickle` module. These facilities will be routinely used to build binary representations of objects to communicate (at sending processes), and restoring them back (at receiving processes).

Although simple and general, the serialization approach (i.e., *pickling* and *unpickling*) previously discussed imposes important overheads in memory as well as processor usage, especially in the scenario of objects with large memory footprints being communicated. Pickling general Python objects, ranging from primitive or container built-in types to user-defined classes, necessarily requires computer resources. Processing is also needed for dispatching the appropriate serialization method (that depends on the type of the object) and doing the actual packing. Additional memory is always needed, and if its total amount is not known *a priori*, many reallocations can occur. Indeed, in the case of large numeric arrays, this is certainly unacceptable and precludes communication of objects occupying half or more of the available memory resources.

*MPI for Python* supports direct communication of any object exporting the single-segment buffer interface. This interface is a standard Python mechanism provided by some types (e.g., strings and numeric arrays), allowing access in the C side to a contiguous memory buffer (i.e., address and length) containing the relevant data. This feature, in conjunction with the capability of constructing user-defined MPI datatypes describing complicated memory layouts, enables the implementation of many algorithms involving multidimensional numeric arrays (e.g., image processing, fast Fourier transforms, finite difference schemes on structured Cartesian grids) directly in Python, with negligible overhead, and almost as fast as compiled Fortran, C, or C++ codes.

## 2.2 Communicators

In *MPI for Python*, `MPI.Comm` is the base class of communicators. The `MPI.Intracomm` and `MPI.Intercomm` classes are sublcasses of the `MPI.Comm` class. The `MPI.Comm.Is_inter()` method (and `MPI.Comm.Is_intra()`, provided for convenience but not part of the MPI specification) is defined for communicator objects and can be used to determine the particular communicator class.

The two predefined intracommunicator instances are available: `MPI.COMM_SELF` and `MPI.COMM_WORLD`. From them, new communicators can be created as needed.

The number of processes in a communicator and the calling process rank can be respectively obtained with methods `MPI.Comm.Get_size()` and `MPI.Comm.Get_rank()`. The associated process group can be retrieved from a communicator by calling the `MPI.Comm.Get_group()` method, which returns an instance of the `MPI.Group` class. Set operations with `MPI.Group` objects like like `MPI.Group.Union()`, `MPI.Group.Intersect()` and `MPI.Group.Difference()` are fully supported, as well as the creation of new communicators from these groups using `MPI.Comm.Create()` and `MPI.Comm.Create_group()`.

New communicator instances can be obtained with the `MPI.Comm.Clone()`, `MPI.Comm.Dup()` and `MPI.Comm.Split()` methods, as well methods `MPI.Intracomm.Create_intercomm()` and `MPI.Intercomm.Merge()`.

Virtual topologies (`MPI.Cartcomm`, `MPI.Graphcomm` and `MPI.Distgraphcomm` classes, which are specializations of the `MPI.Intracomm` class) are fully supported. New instances can be obtained from intracommunicator instances with factory methods `MPI.Intracomm.Create_cart()` and `MPI.Intracomm.Create_graph()`.

## 2.3 Point-to-Point Communications

Point to point communication is a fundamental capability of message passing systems. This mechanism enables the transmission of data between a pair of processes, one side sending, the other receiving.

MPI provides a set of *send* and *receive* functions allowing the communication of *typed* data with an associated *tag*. The type information enables the conversion of data representation from one architecture to another in the case of heterogeneous computing environments; additionally, it allows the representation of non-contiguous data layouts and user-defined datatypes, thus avoiding the overhead of (otherwise unavoidable) packing/unpacking operations. The tag information allows selectivity of messages at the receiving end.

### Blocking Communications

MPI provides basic send and receive functions that are *blocking*. These functions block the caller until the data buffers involved in the communication can be safely reused by the application program.

In *MPI for Python*, the `MPI.Comm.Send()`, `MPI.Comm.Recv()` and `MPI.Comm.Sendrecv()` methods of communicator objects provide support for blocking point-to-point communications within `MPI.Intracomm` and `MPI.Intercomm` instances. These methods can communicate memory buffers. The variants `MPI.Comm.send()`, `MPI.Comm.recv()` and `MPI.Comm.sendrecv()` can communicate general Python objects.

### Nonblocking Communications

On many systems, performance can be significantly increased by overlapping communication and computation. This is particularly true on systems where communication can be executed autonomously by an intelligent, dedicated communication controller.

MPI provides *nonblocking* send and receive functions. They allow the possible overlap of communication and computation. Non-blocking communication always come in two parts: posting functions, which begin the requested operation; and test-for-completion functions, which allow to discover whether the requested operation has completed.

In *MPI for Python*, the `MPI.Comm.Isend()` and `MPI.Comm.Irecv()` methods initiate send and receive operations, respectively. These methods return a `MPI.Request` instance, uniquely identifying the started operation. Its completion can be managed using the `MPI.Request.Test()`, `MPI.Request.Wait()` and `MPI.Request.Cancel()` methods. The management of `MPI.Request` objects and associated memory buffers involved in communication requires a careful, rather low-level coordination. Users must ensure that objects exposing their memory buffers are not accessed at the Python level while they are involved in nonblocking message-passing operations.

### Persistent Communications

Often a communication with the same argument list is repeatedly executed within an inner loop. In such cases, communication can be further optimized by using persistent communication, a particular case of nonblocking communication allowing the reduction of the overhead between processes and communication controllers. Furthermore , this kind of optimization can also alleviate the extra call overheads associated to interpreted, dynamic languages like Python.

In *MPI for Python*, the `MPI.Comm.Send_init()` and `MPI.Comm.Recv_init()` methods create persistent requests for a send and receive operation, respectively. These methods return an instance of the `MPI.Prequest` class, a subclass of the `MPI.Request` class. The actual communication can be effectively started using the `MPI.Prequest.Start()` method, and its completion can be managed as previously described.

## 2.4 Collective Communications

Collective communications allow the transmittal of data between multiple processes of a group simultaneously. The syntax and semantics of collective functions is consistent with point-to-point communication. Collective functions communicate *typed* data, but messages are not paired with an associated *tag*; selectivity of messages is implied in the calling order. Additionally, collective functions come in blocking versions only.

The more commonly used collective communication operations are the following.

- Barrier synchronization across all group members.
- Global communication functions
  - Broadcast data from one member to all members of a group.
  - Gather data from all members to one member of a group.
  - Scatter data from one member to all members of a group.

- Global reduction operations such as sum, maximum, minimum, etc.

In *MPI for Python*, the `MPI.Comm.Bcast()`, `MPI.Comm.Scatter()`, `MPI.Comm.Gather()`, `MPI.Comm.Allgather()`, and `MPI.Comm.Alltoall()` `MPI.Comm.Alltoallw()` methods provide support for collective communications of memory buffers. The lower-case variants `MPI.Comm.bcast()`, `MPI.Comm.scatter()`, `MPI.Comm.gather()`, `MPI.Comm.allgather()` and `MPI.Comm.alltoall()` can communicate general Python objects. The vector variants (which can communicate different amounts of data to each process) `MPI.Comm.Scatterv()`, `MPI.Comm.Gatherv()`, `MPI.Comm.Allgatherv()`, `MPI.Comm.Alltoallv()` and `MPI.Comm.Alltoallw()` are also supported, they can only communicate objects exposing memory buffers.

Global reduction operations on memory buffers are accessible through the `MPI.Comm.Reduce()`, *MPI.Comm.Reduce_scatter*, `MPI.Comm.Allreduce()`, `MPI.Intracomm.Scan()` and `MPI.Intracomm.Exscan()` methods. The lower-case variants `MPI.Comm.reduce()`, `MPI.Comm.allreduce()`, `MPI.Intracomm.scan()` and `MPI.Intracomm.exscan()` can communicate general Python objects; however, the actual required reduction computations are performed sequentially at some process. All the predefined (i.e., `MPI.SUM`, `MPI.PROD`, `MPI.MAX`, etc.) reduction operations can be applied.

## 2.5 Support for CUDA-aware MPI

Several MPI implementations, including Open MPI and MVAPICH, support passing CUDA GPU pointers to MPI calls to avoid explict data movement between the host and the device. On the Python side, CUDA GPU arrays have been implemented by many libraries that need GPU computation, such as CuPy, Numba, PyTorch, and PyArrow. In order to increase library interoperability, a `__cuda_array_interface__` attribute is defined and agreed upon. For example, a CuPy array can be passed to a Numba CUDA-jit kernel.

*MPI for Python* provides an experimental support for CUDA-aware MPI. This feature requires:

1. mpi4py is built against a CUDA-aware MPI library.
2. The Python GPU arrays are compliant with the `__cuda_array_interface__` standard.

See the *Tutorial* section for further information. We note that

- Whether or not a MPI call can work for GPU arrays depends on the underlying MPI implementation, not on mpi4py.
- This support is currently experimental (so is the `__cuda_array_interface__` standard) and subject to change in the future.

## 2.6 Dynamic Process Management

In the context of the MPI-1 specification, a parallel application is static; that is, no processes can be added to or deleted from a running application after it has been started. Fortunately, this limitation was addressed in MPI-2. The new specification added a process management model providing a basic interface between an application and external resources and process managers.

This MPI-2 extension can be really useful, especially for sequential applications built on top of parallel modules, or parallel applications with a client/server model. The MPI-2 process model provides a mechanism to create new processes and establish communication between them and the existing MPI application. It also provides mechanisms to establish communication between two existing MPI applications, even when one did not *start* the other.

In *MPI for Python*, new independent process groups can be created by calling the `MPI.Intracomm.Spawn()` method within an intracommunicator. This call returns a new intercommunicator (i.e., an `MPI.Intercomm` instance) at the parent process group. The child process group can retrieve the matching intercommunicator by calling the `MPI.Comm.Get_parent()` class method. At each side, the new intercommunicator can be used to perform point to point and collective communications between the parent and child groups of processes.

Alternatively, disjoint groups of processes can establish communication using a client/server approach. Any server application must first call the `MPI.Open_port()` function to open a *port* and the `MPI.Publish_name()` function to publish a provided *service*, and next call the `MPI.Intracomm.Accept()` method. Any client applications can first find a published *service* by calling the `MPI.Lookup_name()` function, which returns the *port* where a server can be contacted; and next call the `MPI.Intracomm.Connect()` method. Both `MPI.Intracomm.Accept()` and `MPI.Intracomm.Connect()` methods return an `MPI.Intercomm` instance. When connection between client/server processes is no longer needed, all of them must cooperatively call the `MPI.Comm.Disconnect()` method. Additionally, server applications should release resources by calling the `MPI.Unpublish_name()` and `MPI.Close_port()` functions.

## 2.7 One-Sided Communications

One-sided communications (also called *Remote Memory Access*, *RMA*) supplements the traditional two-sided, send/receive based MPI communication model with a one-sided, put/get based interface. One-sided communication that can take advantage of the capabilities of highly specialized network hardware. Additionally, this extension lowers latency and software overhead in applications written using a shared-memory-like paradigm.

The MPI specification revolves around the use of objects called *windows*; they intuitively specify regions of a process's memory that have been made available for remote read and write operations. The published memory blocks can be accessed through three functions for put (remote send), get (remote write), and accumulate (remote update or reduction) data items. A much larger number of functions support different synchronization styles; the semantics of these synchronization operations are fairly complex.

In *MPI for Python*, one-sided operations are available by using instances of the `MPI.Win` class. New window objects are created by calling the `MPI.Win.Create()` method at all processes within a communicator and specifying a memory buffer . When a window instance is no longer needed, the `MPI.Win.Free()` method should be called.

The three one-sided MPI operations for remote write, read and reduction are available through calling the methods `MPI.Win.Put()`, `MPI.Win.Get()`, and `MPI.Win.Accumulate()` respectively within a `Win` instance. These methods need an integer rank identifying the target process and an integer offset relative the base address of the remote memory block being accessed.

The one-sided operations read, write, and reduction are implicitly nonblocking, and must be synchronized by using two primary modes. Active target synchronization requires the origin process to call the `MPI.Win.Start()` and `MPI.Win.Complete()` methods at the origin process, and target process cooperates by calling the `MPI.Win.Post()` and `MPI.Win.Wait()` methods. There is also a collective variant provided by the `MPI.Win.Fence()` method. Passive target synchronization is more lenient, only the origin process calls the `MPI.Win.Lock()` and `MPI.Win.Unlock()` methods. Locks are used to protect remote accesses to the locked remote window and to protect local load/store accesses to a locked local window.

## 2.8 Parallel Input/Output

The POSIX standard provides a model of a widely portable file system. However, the optimization needed for parallel input/output cannot be achieved with this generic interface. In order to ensure efficiency and scalability, the underlying parallel input/output system must provide a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. Additionally, further efficiencies can be gained via support for asynchronous input/output, strided accesses to data, and control over physical file layout on storage devices. This scenario motivated the inclusion in the MPI-2 standard of a custom interface in order to support more elaborated parallel input/output operations.

The MPI specification for parallel input/output revolves around the use objects called *files*. As defined by MPI, files are not just contiguous byte streams. Instead, they are regarded as ordered collections of *typed* data items. MPI supports sequential or random access to any integral set of these items. Furthermore, files are opened collectively by a group of processes.

The common patterns for accessing a shared file (broadcast, scatter, gather, reduction) is expressed by using user-defined datatypes. Compared to the communication patterns of point-to-point and collective communications, this approach has the advantage of added flexibility and expressiveness. Data access operations (read and write) are defined for different kinds of positioning (using explicit offsets, individual file pointers, and shared file pointers), coordination (non-collective and collective), and synchronism (blocking, nonblocking, and split collective with begin/end phases).

In *MPI for Python*, all MPI input/output operations are performed through instances of the `MPI.File` class. File handles are obtained by calling the `MPI.File.Open()` method at all processes within a communicator and providing a file name and the intended access mode. After use, they must be closed by calling the `MPI.File.Close()` method. Files even can be deleted by calling method `MPI.File.Delete()`.

After creation, files are typically associated with a per-process *view*. The view defines the current set of data visible and accessible from an open file as an ordered set of elementary datatypes. This data layout can be set and queried with the `MPI.File.Set_view()` and `MPI.File.Get_view()` methods respectively.

Actual input/output operations are achieved by many methods combining read and write calls with different behavior regarding positioning, coordination, and synchronism. Summing up, *MPI for Python* provides the thirty (30) methods defined in MPI-2 for reading from or writing to files using explicit offsets or file pointers (individual or shared), in blocking or nonblocking and collective or noncollective versions.

## 2.9 Environmental Management

### Initialization and Exit

Module functions `MPI.Init()` or `MPI.Init_thread()` and `MPI.Finalize()` provide MPI initialization and finalization respectively. Module functions `MPI.Is_initialized()` and `MPI.Is_finalized()` provide the respective tests for initialization and finalization.

---

**Note:** `MPI_Init()` or `MPI_Init_thread()` is actually called when you import the `MPI` module from the `mpi4py` package, but only if MPI is not already initialized. In such case, calling `MPI.Init()` or `MPI.Init_thread()` from Python is expected to generate an MPI error, and in turn an exception will be raised.

---

---

**Note:** `MPI_Finalize()` is registered (by using Python C/API function `Py_AtExit()`) for being automatically called when Python processes exit, but only if `mpi4py` actually initialized MPI. Therefore, there is no need to call `MPI.Finalize()` from Python to ensure MPI finalization.

---

### Implementation Information

- The MPI version number can be retrieved from module function `MPI.Get_version()`. It returns a two-integer tuple `(version,subversion)`.
- The `MPI.Get_processor_name()` function can be used to access the processor name.
- The values of predefined attributes attached to the world communicator can be obtained by calling the `MPI.Comm.Get_attr()` method within the `MPI.COMM_WORLD` instance.

### Timers

MPI timer functionalities are available through the `MPI.Wtime()` and `MPI.Wtick()` functions.

### Error Handling

In order facilitate handle sharing with other Python modules interfacing MPI-based parallel libraries, the predefined MPI error handlers `MPI.ERRORS_RETURN` and `MPI.ERRORS_ARE_FATAL` can be assigned to and retrieved from communicators, windows and files using methods `MPI.{Comm|Win|File}.Set_errhandler()` and `MPI.{Comm|Win|File}.Get_errhandler()`.

When the predefined error handler `MPI.ERRORS_RETURN` is set, errors returned from MPI calls within Python code will raise an instance of the exception class `MPI.Exception`, which is a subclass of the standard Python exception `RuntimeError`.

---

**Note:** After import, mpi4py overrides the default MPI rules governing inheritance of error handlers. The `MPI.ERRORS_RETURN` error handler is set in the predefined `MPI.COMM_SELF` and `MPI.COMM_WORLD` communicators, as well as any new `MPI.Comm`, `MPI.Win`, or `MPI.File` instance created through mpi4py. If you ever pass such handles to C/C++/Fortran library code, it is recommended to set the `MPI.ERRORS_ARE_FATAL` error handler on them to ensure MPI errors do not pass silently.

---

**Warning:** Importing with `from mpi4py.MPI import *` will cause a name clashing with the standard Python `Exception` base class.

# 3 Tutorial

**Warning:** Under construction. Contributions very welcome!

*MPI for Python* supports convenient, *pickle*-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

- Communication of generic Python objects

  You have to use **all-lowercase** methods (of the `Comm` class), like `send()`, `recv()`, `bcast()`. An object to be sent is passed as a paramenter to the communication call, and the received object is simply the return value.

  The `isend()` and `irecv()` methods return `Request` instances; completion of these methods can be managed using the `test()` and `wait()` methods of the `Request` class.

  The `recv()` and `irecv()` methods may be passed a buffer object that can be repeatedly used to receive messages avoiding internal memory allocation. This buffer must be sufficiently large to accommodate the transmitted messages; hence, any buffer passed to `recv()` or `irecv()` must be at least as long as the *pickled* data transmitted to the receiver.

  Collective calls like `scatter()`, `gather()`, `allgather()`, `alltoall()` expect a single value or a sequence of `Comm.size` elements at the root or all process. They return a single value, a list of `Comm.size` elements, or `None`.

- Communication of buffer-like objects

  You have to use method names starting with an **upper-case** letter (of the `Comm` class), like `Send()`, `Recv()`, `Bcast()`, `Scatter()`, `Gather()`.

  In general, buffer arguments to these calls must be explicitly specified by using a 2/3-list/tuple like `[data, MPI.DOUBLE]`, or `[data, count, MPI.DOUBLE]` (the former one uses the byte-size of `data` and the extent of the MPI datatype to define `count`).

For vector collectives communication operations like `Scatterv()` and `Gatherv()`, buffer arguments are specified as `[data, count, displ, datatype]`, where `count` and `displ` are sequences of integral values.

Automatic MPI datatype discovery for NumPy/CUDA arrays and PEP-3118 buffers is supported, but limited to basic C types (all C/C99-native signed/unsigned integral types and single/double precision real/complex floating types) and availability of matching datatypes in the underlying MPI implementation. In this case, the buffer-provider object can be passed directly as a buffer argument, the count and MPI datatype will be inferred.

If mpi4py is built against a CUDA-aware MPI, CUDA GPU arrays can be passed to the upper-case methods as long as they have a `__cuda_array_interface__` attribute compliant with the standard specification. Moreover, only C- or Fortran- contiguous CUDA arrays are supported.

## 3.1 Running Python scripts with MPI

Most MPI programs can be run with the command **`mpiexec`**. In practice, running Python programs looks like:

```
$ mpiexec -n 4 python script.py
```

to run the program with 4 processors.

## 3.2 Point-to-Point Communication

- Python objects (`pickle` under the hood):

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Python objects with non-blocking communication:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

- NumPy arrays (the fast way!):

```
from mpi4py import MPI
import numpy
```

(continues on next page)

(continued from previous page)

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)

# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

## 3.3 Collective Communication

- Broadcasting a Python dictionary:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
```

- Scattering Python objects:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

- Gathering Python objects:

```
from mpi4py import MPI
```

(continues on next page)

(continued from previous page)

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None
```

- Broadcasting a NumPy array:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(100, dtype='i')
else:
    data = np.empty(100, dtype='i')
comm.Bcast(data, root=0)
for i in range(100):
    assert data[i] == i
```

- Scattering NumPy arrays:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = None
if rank == 0:
    sendbuf = np.empty([size, 100], dtype='i')
    sendbuf.T[:,:] = range(size)
recvbuf = np.empty(100, dtype='i')
comm.Scatter(sendbuf, recvbuf, root=0)
assert np.allclose(recvbuf, rank)
```

- Gathering NumPy arrays:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = np.zeros(100, dtype='i') + rank
recvbuf = None
if rank == 0:
```

(continues on next page)

(continued from previous page)

```
    recvbuf = np.empty([size, 100], dtype='i')
comm.Gather(sendbuf, recvbuf, root=0)
if rank == 0:
    for i in range(size):
        assert np.allclose(recvbuf[i,:], i)
```

- Parallel matrix-vector product:

```
from mpi4py import MPI
import numpy

def matvec(comm, A, x):
    m = A.shape[0] # local rows
    p = comm.Get_size()
    xg = numpy.zeros(m*p, dtype='d')
    comm.Allgather([x,  MPI.DOUBLE],
                   [xg, MPI.DOUBLE])
    y = numpy.dot(A, xg)
    return y
```

## 3.4 MPI-IO

- Collective I/O with NumPy arrays:

```
from mpi4py import MPI
import numpy as np

amode = MPI.MODE_WRONLY|MPI.MODE_CREATE
comm = MPI.COMM_WORLD
fh = MPI.File.Open(comm, "./datafile.contig", amode)

buffer = np.empty(10, dtype=np.int)
buffer[:] = comm.Get_rank()

offset = comm.Get_rank()*buffer.nbytes
fh.Write_at_all(offset, buffer)

fh.Close()
```

- Non-contiguous Collective I/O with NumPy arrays and datatypes:

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

amode = MPI.MODE_WRONLY|MPI.MODE_CREATE
fh = MPI.File.Open(comm, "./datafile.noncontig", amode)

item_count = 10

buffer = np.empty(item_count, dtype='i')
buffer[:] = rank
```

(continues on next page)

(continued from previous page)

```
filetype = MPI.INT.Create_vector(item_count, 1, size)
filetype.Commit()

displacement = MPI.INT.Get_size()*rank
fh.Set_view(displacement, filetype=filetype)

fh.Write_all(buffer)
filetype.Free()
fh.Close()
```

## 3.5 Dynamic Process Management

- Compute Pi - Master (or parent, or client) side:

```
#!/usr/bin/env python
from mpi4py import MPI
import numpy
import sys

comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['cpi.py'],
                           maxprocs=5)

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
            op=MPI.SUM, root=MPI.ROOT)
print(PI)

comm.Disconnect()
```

- Compute Pi - Worker (or child, or server) side:

```
#!/usr/bin/env python
from mpi4py import MPI
import numpy

comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
h = 1.0 / N; s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
            op=MPI.SUM, root=0)

comm.Disconnect()
```

## 3.6 CUDA-aware MPI + Python GPU arrays

- Reduce-to-all CuPy arrays:

```
from mpi4py import MPI
import cupy as cp

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = cp.arange(10, dtype='i')
recvbuf = cp.empty_like(sendbuf)
assert hasattr(sendbuf, '__cuda_array_interface__')
assert hasattr(recvbuf, '__cuda_array_interface__')
comm.Allreduce(sendbuf, recvbuf)

assert cp.allclose(recvbuf, sendbuf*size)
```

## 3.7 Wrapping with SWIG

- C source:

```
/* file: helloworld.c */
void sayhello(MPI_Comm comm)
{
  int size, rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  printf("Hello, World! "
         "I am process %d of %d.\n",
         rank, size);
}
```

- SWIG interface file:

```
// file: helloworld.i
%module helloworld
%{
#include <mpi.h>
#include "helloworld.c"
}%

%include mpi4py/mpi4py.i
%mpi4py_typemap(Comm, MPI_Comm);
void sayhello(MPI_Comm comm);
```

- Try it in the Python prompt:

```
>>> from mpi4py import MPI
>>> import helloworld
>>> helloworld.sayhello(MPI.COMM_WORLD)
Hello, World! I am process 0 of 1.
```

## 3.8 Wrapping with F2Py

- Fortran 90 source:

```
! file: helloworld.f90
subroutine sayhello(comm)
  use mpi
  implicit none
  integer :: comm, rank, size, ierr
  call MPI_Comm_size(comm, size, ierr)
  call MPI_Comm_rank(comm, rank, ierr)
  print *, 'Hello, World! I am process ',rank,' of ',size,'.'
end subroutine sayhello
```

- Compiling example using f2py

```
$ f2py -c --f90exec=mpif90 helloworld.f90 -m helloworld
```

- Try it in the Python prompt:

```
>>> from mpi4py import MPI
>>> import helloworld
>>> fcomm = MPI.COMM_WORLD.py2f()
>>> helloworld.sayhello(fcomm)
Hello, World! I am process 0 of 1.
```

# 4 mpi4py.futures

New in version 3.0.0.

This package provides a high-level interface for asynchronously executing callables on a pool of worker processes using MPI for inter-process communication.

## 4.1 concurrent.futures

The *mpi4py.futures* package is based on `concurrent.futures` from the Python standard library. More precisely, *mpi4py.futures* provides the *MPIPoolExecutor* class as a concrete implementation of the abstract class `Executor`. The `submit()` interface schedules a callable to be executed asynchronously and returns a `Future` object representing the execution of the callable. `Future` instances can be queried for the call result or exception. Sets of `Future` instances can be passed to the `wait()` and `as_completed()` functions.

---

**Note:** The `concurrent.futures` package was introduced in Python 3.2. A backport targeting Python 2.7 is available on PyPI. The *mpi4py.futures* package uses `concurrent.futures` if available, either from the Python 3 standard library or the Python 2.7 backport if installed. Otherwise, *mpi4py.futures* uses a bundled copy of core functionality backported from Python 3.5 to work with Python 2.7.

---

**See also:**

**Module `concurrent.futures`** Documentation of the `concurrent.futures` standard module.

## 4.2 MPIPoolExecutor

The *MPIPoolExecutor* class uses a pool of MPI processes to execute calls asynchronously. By performing computations in separate processes, it allows to side-step the Global Interpreter Lock but also means that only picklable objects can be executed and returned. The `__main__` module must be importable by worker processes, thus *MPIPoolExecutor* instances may not work in the interactive interpreter.

*MPIPoolExecutor* takes advantage of the dynamic process management features introduced in the MPI-2 standard. In particular, the `MPI.Intracomm.Spawn()` method of `MPI.COMM_SELF()` is used in the master (or parent) process to spawn new worker (or child) processes running a Python interpreter. The master process uses a separate thread (one for each *MPIPoolExecutor* instance) to communicate back and forth with the workers. The worker processes serve the execution of tasks in the main (and only) thread until they are signaled for completion.

---

**Note:** The worker processes must import the main script in order to *unpickle* any callable defined in the `__main__` module and submitted from the master process. Furthermore, the callables may need access to other global variables. At the worker processes,:mod:*mpi4py.futures* executes the main script code (using the `runpy` module) under the `__worker__` namespace to define the `__main__` module. The `__main__` and `__worker__` modules are added to `sys.modules` (both at the master and worker processes) to ensure proper *pickling* and *unpickling*.

---

**Warning:** During the initial import phase at the workers, the main script cannot create and use new *MPIPoolExecutor* instances. Otherwise, each worker would attempt to spawn a new pool of workers, leading to infinite recursion. *mpi4py.futures* detects such recursive attempts to spawn new workers and aborts the MPI execution environment. As the main script code is run under the `__worker__` namespace, the easiest way to avoid spawn recursion is using the idiom `if __name__ == '__main__': ...` in the main script.

**class** mpi4py.futures.**MPIPoolExecutor**(*max_workers=None*, ***kwargs*)

An `Executor` subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, its value is determined from the `MPI4PY_MAX_WORKERS` environment variable if set, or the MPI universe size if set, otherwise a single worker process is spawned. If *max_workers* is lower than or equal to `0`, then a `ValueError` will be raised.

Other parameters:

- *python_exe*: Path to the Python interpreter executable used to spawn worker processes, otherwise `sys.executable` is used.
- *python_args*: `list` or iterable with additional command line flags to pass to the Python executable. Command line flags determined from inspection of `sys.flags`, `sys.warnoptions` and `sys._xoptions` in are passed unconditionally.
- *mpi_info*: `dict` or iterable yielding `(key, value)` pairs. These `(key, value)` pairs are passed (through an `MPI.Info` object) to the `MPI.Intracomm.Spawn()` call used to spawn worker processes. This mechanism allows telling the MPI runtime system where and how to start the processes. Check the documentation of the backend MPI implementation about the set of keys it interprets and the corresponding format for values.
- *globals*: `dict` or iterable yielding `(name, value)` pairs to initialize the main module namespace in worker processes.
- *main*: If set to `False`, do not import the `__main__` module in worker processes. Setting *main* to `False` prevents worker processes from accessing definitions in the parent `__main__` namespace.
- *path*: `list` or iterable with paths to append to `sys.path` in worker processes to extend the module search path.

- *wdir*: Path to set the current working directory in worker processes using `os.chdir()`. The initial working directory is set by the MPI implementation. Quality MPI implementations should honor a `wdir` info key passed through *mpi_info*, although such feature is not mandatory.
- *env*: `dict` or iterable yielding `(name, value)` pairs with environment variables to update `os.environ` in worker processes. The initial environment is set by the MPI implementation. MPI implementations may allow setting the initial environment through *mpi_info*, however such feature is not required nor recommended by the MPI standard.

**submit** (*func*, **args*, ***kwargs*)

Schedule the callable, *func*, to be executed as `func(*args, **kwargs)` and returns a `Future` object representing the execution of the callable.

```
executor = MPIPoolExecutor(max_workers=1)
future = executor.submit(pow, 321, 1234)
print(future.result())
```

**map** (*func*, **iterables*, *timeout=None*, *chunksize=1*, ***kwargs*)

Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently, out-of-order, in separate processes. The returned iterator raises a `TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to *map()*. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator. This method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of one. By default, the returned iterator yields results in-order, waiting for successive tasks to complete . This behavior can be changed by passing the keyword argument *unordered* as `True`, then the result iterator will yield a result as soon as any of the tasks complete.

```
executor = MPIPoolExecutor(max_workers=3)
for result in executor.map(pow, [2]*32, range(32)):
    print(result)
```

**starmap** (*func*, *iterable*, *timeout=None*, *chunksize=1*, ***kwargs*)

Equivalent to `itertools.starmap(func, iterable)`. Used instead of *map()* when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). *map(func, *iterable)* is equivalent to *starmap(func, zip(*iterable))*.

```
executor = MPIPoolExecutor(max_workers=3)
iterable = ((2, n) for n in range(32))
for result in executor.starmap(pow, iterable):
    print(result)
```

**shutdown** (*wait=True*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to *submit()* and *map()* made after *shutdown()* will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the executor instance (waiting as if *shutdown()* were called with *wait* set to `True`).

```
import time
with MPIPoolExecutor(max_workers=1) as executor:
    future = executor.submit(time.sleep, 2)
assert future.done()
```

**bootup**(*wait=True*)
Signal the executor that it should allocate eagerly any required resources (in particular, MPI worker processes). If *wait* is `True`, then *`bootup()`* will not return until the executor resources are ready to process submissions. Resources are automatically allocated in the first call to *`submit()`*, thus calling *`bootup()`* explicitly is seldom needed.

---

**Note:** As the master process uses a separate thread to perform MPI communication with the workers, the backend MPI implementation should provide support for `MPI.THREAD_MULTIPLE`. However, some popular MPI implementations do not support yet concurrent MPI calls from multiple threads. Additionally, users may decide to initialize MPI with a lower level of thread support. If the level of thread support in the backend MPI is less than `MPI.THREAD_MULTIPLE`, *`mpi4py.futures`* will use a global lock to serialize MPI calls. If the level of thread support is less than `MPI.THREAD_SERIALIZED`, *`mpi4py.futures`* will emit a `RuntimeWarning`.

---

**Warning:** If the level of thread support in the backend MPI is less than `MPI.THREAD_SERIALIZED` (i.e, it is either `MPI.THREAD_SINGLE` or `MPI.THREAD_FUNNELED`), in theory *`mpi4py.futures`* cannot be used. Rather than raising an exception, *`mpi4py.futures`* emits a warning and takes a "cross-fingers" attitude to continue execution in the hope that serializing MPI calls with a global lock will actually work.

## 4.3 MPICommExecutor

Legacy MPI-1 implementations (as well as some vendor MPI-2 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 complications to applications using the `MPI_Comm_spawn()` routine.

With these issues in mind, *`mpi4py.futures`* supports an additonal, more traditional, SPMD-like usage pattern requiring MPI-1 calls only. Python applications are started the usual way, e.g., using the **`mpiexec`** command. Python code should make a collective call to the *`MPICommExecutor`* context manager to partition the set of MPI processes within a MPI communicator in one master processes and many workers processes. The master process gets access to an *`MPIPoolExecutor`* instance to submit tasks. Meanwhile, the worker process follow a different execution path and team-up to execute the tasks submitted from the master.

Besides alleviating the lack of dynamic process managment features in legacy MPI-1 or partial MPI-2 implementations, the *`MPICommExecutor`* context manager may be useful in classic MPI-based Python applications willing to take advantage of the simple, task-based, master/worker approach available in the *`mpi4py.futures`* package.

**class** mpi4py.futures.**MPICommExecutor**(*comm=None*, *root=0*)
Context manager for *`MPIPoolExecutor`*. This context manager splits a MPI (intra)communicator *comm* (defaults to `MPI.COMM_WORLD` if not provided or `None`) in two disjoint sets: a single master process (with rank *root* in *comm*) and the remaining worker processes. These sets are then connected through an intercommunicator. The target of the `with` statement is assigned either an *`MPIPoolExecutor`* instance (at the master) or `None` (at the workers).

```
from mpi4py import MPI
from mpi4py.futures import MPICommExecutor

with MPICommExecutor(MPI.COMM_WORLD, root=0) as executor:
```

(continues on next page)

(continued from previous page)

```
    if executor is not None:
        future = executor.submit(abs, -42)
        assert future.result() == 42
        answer = set(executor.map(abs, [-42, 42]))
        assert answer == {42}
```

**Warning:** If *MPICommExecutor* is passed a communicator of size one (e.g., `MPI.COMM_SELF`), then the executor instace assigned to the target of the `with` statement will execute all submitted tasks in a single worker thread, thus ensuring that task execution still progress asynchronously. However, the GIL will prevent the main and worker threads from running concurrently in multicore processors. Moreover, the thread context switching may harm noticeably the performance of CPU-bound tasks. In case of I/O-bound tasks, the GIL is not usually an issue, however, as a single worker thread is used, it progress one task at a time. We advice against using *MPICommExecutor* with communicators of size one and suggest refactoring your code to use instead a `ThreadPoolExecutor`.

## 4.4 Command line

Recalling the issues related to the lack of support for dynamic process managment features in MPI implementations, *mpi4py.futures* supports an alternative usage pattern where Python code (either from scripts, modules, or zip files) is run under command line control of the *mpi4py.futures* package by passing `-m mpi4py.futures` to the **python** executable. The `mpi4py.futures` invocation should be passed a *pyfile* path to a script (or a zipfile/directory containing a `__main__.py` file). Additionally, `mpi4py.futures` accepts `-m` *mod* to execute a module named *mod*, `-c` *cmd* to execute a command string *cmd*, or even `-` to read commands from standard input (`sys.stdin`). Summarizing, `mpi4py.futures` can be invoked in the following ways:

- `$ mpiexec -n` *numprocs* `python -m mpi4py.futures` *pyfile* `[arg] ...`
- `$ mpiexec -n` *numprocs* `python -m mpi4py.futures -m` *mod* `[arg] ...`
- `$ mpiexec -n` *numprocs* `python -m mpi4py.futures -c` *cmd* `[arg] ...`
- `$ mpiexec -n` *numprocs* `python -m mpi4py.futures - [arg] ...`

Before starting the main script execution, *mpi4py.futures* splits `MPI.COMM_WORLD` in one master (the process with rank 0 in `MPI.COMM_WORLD`) and 16 workers and connect them through an MPI intercommunicator. Afterwards, the master process proceeds with the execution of the user script code, which eventually creates *MPIPoolExecutor* instances to submit tasks. Meanwhile, the worker processes follow a different execution path to serve the master. Upon successful termination of the main script at the master, the entire MPI execution environment exists gracefully. In case of any unhandled exception in the main script, the master process calls `MPI.COMM_WORLD.Abort(1)` to prevent deadlocks and force termination of entire MPI execution environment.

**Warning:** Running scripts under command line control of *mpi4py.futures* is quite similar to executing a single-process application that spawn additional workers as required. However, there is a very important difference users should be aware of. All *MPIPoolExecutor* instances created at the master will share the pool of workers. Tasks submitted at the master from many different executors will be scheduled for execution in random order as soon as a worker is idle. Any executor can easily starve all the workers (e.g., by calling *MPIPoolExecutor.map()* with long iterables). If that ever happens, submissions from other executors will not be serviced until free workers are available.

**See also:**

**Command line** Documentation on Python command line interface.

## 4.5 Examples

The following `julia.py` script computes the Julia set and dumps an image to disk in binary PGM format. The code starts by importing *`MPIPoolExecutor`* from the *`mpi4py.futures`* package. Next, some global constants and functions implement the computation of the Julia set. The computations are protected with the standard `if __name__ == '__main__':` ... idiom. The image is computed by whole scanlines submitting all these tasks at once using the *`map`* method. The result iterator yields scanlines in-order as the tasks complete. Finally, each scanline is dumped to disk.

Listing 1: `julia.py`

```
from mpi4py.futures import MPIPoolExecutor

x0, x1, w = -2.0, +2.0, 640*2
y0, y1, h = -1.5, +1.5, 480*2
dx = (x1 - x0) / w
dy = (y1 - y0) / h

c = complex(0, 0.65)

def julia(x, y):
    z = complex(x, y)
    n = 255
    while abs(z) < 3 and n > 1:
        z = z**2 + c
        n -= 1
    return n

def julia_line(k):
    line = bytearray(w)
    y = y1 - k * dy
    for j in range(w):
        x = x0 + j * dx
        line[j] = julia(x, y)
    return line

if __name__ == '__main__':

    with MPIPoolExecutor() as executor:
        image = executor.map(julia_line, range(h))
        with open('julia.pgm', 'wb') as f:
            f.write(b'P5 %d %d %d\n' % (w, h, 255))
            for line in image:
                f.write(line)
```

The recommended way to execute the script is using the **`mpiexec`** command specifying one MPI process and (optional but recommended) the desired MPI universe size[1].

```
$ mpiexec -n 1 -usize 17 python julia.py
```

The **`mpiexec`** command launches a single MPI process (the master) running the Python interpreter and executing the main script. When required, *`mpi4py.futures`* spawns 16 additional MPI processes (the children) to dynamically allocate the pool of workers. The master submits tasks to the children and waits for the results. The children receive incoming tasks, execute them, and send back the results to the master.

[1] This **`mpiexec`** invocation example using the `-usize` flag (alternatively, setting the `MPIEXEC_UNIVERSE_SIZE` environment variable) assumes the backend MPI implementation is an MPICH derivative using the Hydra process manager. In the Open MPI implementation, the MPI universe size can be specified by setting the `OMPI_UNIVERSE_SIZE` environment variable to a positive integer. Check the documentation of your actual MPI implementation and/or batch system for the ways to specify the desired MPI universe size.

Alternatively, users may decide to execute the script in a more traditional way, that is, all the MPI process are started at once. The user script is run under command line control of *mpi4py.futures* passing the -m flag to the **python** executable.

```
$ mpiexec -n 17 python -m mpi4py.futures julia.py
```

As explained previously, the 17 processes are partitioned in one master and 16 workers. The master process executes the main script while the workers execute the tasks submitted from the master.

# 5 mpi4py.run

New in version 3.0.0.

At import time, `mpi4py` initializes the MPI execution environment calling `MPI_Init_thread()` and installs an exit hook to automatically call `MPI_Finalize()` just before the Python process terminates. Additionally, `mpi4py` overrides the default `MPI.ERRORS_ARE_FATAL` error handler in favor of `MPI.ERRORS_RETURN`, which allows translating MPI errors in Python exceptions. These departures from standard MPI behavior may be controversial, but are quite convenient within the highly dynamic Python programming environment. Third-party code using `mpi4py` can just `from mpi4py import MPI` and perform MPI calls without the tedious initialization/finalization handling. MPI errors, once translated automatically to Python exceptions, can be dealt with the common `try...except...finally` clauses; unhandled MPI exceptions will print a traceback which helps in locating problems in source code.

Unfortunately, the interplay of automatic MPI finalization and unhandled exceptions may lead to deadlocks. In unattended runs, these deadlocks will drain the battery of your laptop, or burn precious allocation hours in your supercomputing facility.

Consider the following snippet of Python code. Assume this code is stored in a standard Python script file and run with **mpiexec** in two or more processes.

```
from mpi4py import MPI
assert MPI.COMM_WORLD.Get_size() > 1
rank = MPI.COMM_WORLD.Get_rank()
if rank == 0:
    1/0
    MPI.COMM_WORLD.send(None, dest=1, tag=42)
elif rank == 1:
    MPI.COMM_WORLD.recv(source=0, tag=42)
```

Process 0 raises `ZeroDivisionError` exception before performing a send call to process 1. As the exception is not handled, the Python interpreter running in process 0 will proceed to exit with non-zero status. However, as `mpi4py` installed a finalizer hook to call `MPI_Finalize()` before exit, process 0 will block waiting for other processes to also enter the `MPI_Finalize()` call. Meanwhile, process 1 will block waiting for a message to arrive from process 0, thus never reaching to `MPI_Finalize()`. The whole MPI execution environment is irremediably in a deadlock state.

To alleviate this issue, `mpi4py` offers a simple, alternative command line execution mechanism based on using the -m flag and implemented with the `runpy` module. To use this features, Python code should be run passing `-m mpi4py` in the command line invoking the Python interpreter. In case of unhandled exceptions, the finalizer hook will call `MPI_Abort()` on the `MPI_COMM_WORLD` communicator, thus effectively aborting the MPI execution environment.

**Warning:** When a process is forced to abort, resources (e.g. open files) are not cleaned-up and any registered finalizers (either with the `atexit` module, the Python C/API function `Py_AtExit()`, or even the C standard

library function `atexit()`) will not be executed. Thus, aborting execution is an extremely impolite way of ensuring process termination. However, MPI provides no other mechanism to recover from a deadlock state.

## 5.1 Interface options

The use of `-m mpi4py` to execute Python code on the command line resembles that of the Python interpreter.

- `mpiexec -n` *`numprocs`* `python -m mpi4py` *`pyfile`* `[arg] ...`
- `mpiexec -n` *`numprocs`* `python -m mpi4py -m` *`mod`* `[arg] ...`
- `mpiexec -n` *`numprocs`* `python -m mpi4py -c` *`cmd`* `[arg] ...`
- `mpiexec -n` *`numprocs`* `python -m mpi4py - [arg] ...`

**<pyfile>**
: Execute the Python code contained in *pyfile*, which must be a filesystem path referring to either a Python file, a directory containing a `__main__.py` file, or a zipfile containing a `__main__.py` file.

**-m** <mod>
: Search `sys.path` for the named module *mod* and execute its contents.

**-c** <cmd>
: Execute the Python code in the *cmd* string command.

**-**
: Read commands from standard input (`sys.stdin`).

**See also:**

**Command line** Documentation on Python command line interface.

# 6 Citation

If MPI for Python been significant to a project that leads to an academic publication, please acknowledge that fact by citing the project.

- L. Dalcin, P. Kler, R. Paz, and A. Cosimo, *Parallel Distributed Computing using Python*, Advances in Water Resources, 34(9):1124-1139, 2011. http://dx.doi.org/10.1016/j.advwatres.2011.04.013
- 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, 68(5):655-662, 2008. http://dx.doi.org/10.1016/j.jpdc.2007.09.005
- L. Dalcin, R. Paz, and M. Storti, *MPI for Python*, Journal of Parallel and Distributed Computing, 65(9):1108-1115, 2005. http://dx.doi.org/10.1016/j.jpdc.2005.03.010

# 7 Installation

## 7.1 Requirements

You need to have the following software properly installed in order to build *MPI for Python*:

- A working MPI implementation, preferably supporting MPI-3 and built with shared/dynamic libraries.

---

**Note:** If you want to build some MPI implementation from sources, check the instructions at *Building MPI from sources* in the appendix.

---

- Python 2.7, 3.5 or above.

---

**Note:** Some MPI-1 implementations **do require** the actual command line arguments to be passed in `MPI_Init()`. In this case, you will need to use a rebuilt, MPI-enabled, Python interpreter executable. *MPI for Python* has some support for alleviating you from this task. Check the instructions at *MPI-enabled Python interpreter* in the appendix.

---

## 7.2 Using pip or easy_install

If you already have a working MPI (either if you installed it from sources or by using a pre-built package from your favourite GNU/Linux distribution) and the **`mpicc`** compiler wrapper is on your search path, you can use **`pip`**:

```
$ [sudo] pip install mpi4py
```

or alternatively *setuptools* **`easy_install`** (deprecated):

```
$ [sudo] easy_install mpi4py
```

---

**Note:** If the **`mpicc`** compiler wrapper is not on your search path (or if it has a different name) you can use **`env`** to pass the environment variable `MPICC` providing the full path to the MPI compiler wrapper executable:

```
$ [sudo] env MPICC=/path/to/mpicc pip install mpi4py

$ [sudo] env MPICC=/path/to/mpicc easy_install mpi4py
```

---

## 7.3 Using distutils

The *MPI for Python* package is available for download at the project website generously hosted by Bitbucket. You can use **`curl`** or **`wget`** to get a release tarball.

- Using **`curl`**:

  ```
  $ curl -O https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-X.Y.tar.gz
  ```

- Using **`wget`**:

  ```
  $ wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-X.Y.tar.gz
  ```

After unpacking the release tarball:

```
$ tar -zxf mpi4py-X.Y.tar.gz
$ cd mpi4py-X.Y
```

the package is ready for building.

*MPI for Python* uses a standard distutils-based build system. However, some distutils commands (like *build*) have additional options:

**--mpicc**=
: Lets you specify a special location or name for the **mpicc** compiler wrapper.

**--mpi**=
: Lets you pass a section with MPI configuration within a special configuration file.

**--configure**
: Runs exhaustive tests for checking about missing MPI types, constants, and functions. This option should be passed in order to build *MPI for Python* against old MPI-1 or MPI-2 implementations, possibly providing a subset of MPI-3.

If you use a MPI implementation providing a **mpicc** compiler wrapper (e.g., MPICH, Open MPI), it will be used for compilation and linking. This is the preferred and easiest way of building *MPI for Python*.

If **mpicc** is located somewhere in your search path, simply run the *build* command:

```
$ python setup.py build
```

If **mpicc** is not in your search path or the compiler wrapper has a different name, you can run the *build* command specifying its location:

```
$ python setup.py build --mpicc=/where/you/have/mpicc
```

Alternatively, you can provide all the relevant information about your MPI implementation by editing the file called `mpi.cfg`. You can use the default section `[mpi]` or add a new, custom section, for example `[other_mpi]` (see the examples provided in the `mpi.cfg` file as a starting point to write your own section):

```
[mpi]

include_dirs         = /usr/local/mpi/include
libraries            = mpi
library_dirs         = /usr/local/mpi/lib
runtime_library_dirs = /usr/local/mpi/lib

[other_mpi]

include_dirs         = /opt/mpi/include ...
libraries            = mpi ...
library_dirs         = /opt/mpi/lib ...
runtime_library_dirs = /op/mpi/lib ...

...
```

and then run the *build* command, perhaps specifying you custom configuration section:

```
$ python setup.py build --mpi=other_mpi
```

After building, the package is ready for install.

If you have root privileges (either by log-in as the root user of by using **sudo**) and you want to install *MPI for Python* in your system for all users, just do:

```
$ python setup.py install
```

The previous steps will install the `mpi4py` package at standard location *prefix*`/lib/python`*X.X*`/site-packages`.

If you do not have root privileges or you want to install *MPI for Python* for your private use, just do:

```
$ python setup.py install --user
```

### 7.4 Testing

To quickly test the installation:

```
$ mpiexec -n 5 python -m mpi4py.bench helloworld
Hello, World! I am process 0 of 5 on localhost.
Hello, World! I am process 1 of 5 on localhost.
Hello, World! I am process 2 of 5 on localhost.
Hello, World! I am process 3 of 5 on localhost.
Hello, World! I am process 4 of 5 on localhost.
```

If you installed from source, issuing at the command line:

```
$ mpiexec -n 5 python demo/helloworld.py
```

or (in the case of ancient MPI-1 implementations):

```
$ mpirun -np 5 python `pwd`/demo/helloworld.py
```

will launch a five-process run of the Python interpreter and run the test script `demo/helloworld.py` from the source distribution.

You can also run all the *unittest* scripts:

```
$ mpiexec -n 5 python test/runtests.py
```

or, if you have nose unit testing framework installed:

```
$ mpiexec -n 5 nosetests -w test
```

or, if you have py.test unit testing framework installed:

```
$ mpiexec -n 5 py.test test/
```

## 8 Appendix

### 8.1 MPI-enabled Python interpreter

> **Warning:** These days it is no longer required to use the MPI-enabled Python interpreter in most cases, and, therefore, is not built by default anymore because it is too difficult to reliably build a Python interpreter across different distributions. If you know that you still **really** need it, see below on how to use the *build_exe* and *install_exe* commands.

Some MPI-1 implementations (notably, MPICH 1) **do require** the actual command line arguments to be passed at the time `MPI_Init()` is called. In this case, you will need to use a re-built, MPI-enabled, Python interpreter binary executable. A basic implementation (targeting Python 2.X) of what is required is shown below:

```
#include <Python.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
   int status, flag;
   MPI_Init(&argc, &argv);
   status = Py_Main(argc, argv);
   MPI_Finalized(&flag);
   if (!flag) MPI_Finalize();
   return status;
}
```

The source code above is straightforward; compiling it should also be. However, the linking step is more tricky: special flags have to be passed to the linker depending on your platform. In order to alleviate you for such low-level details, *MPI for Python* provides some pure-distutils based support to build and install an MPI-enabled Python interpreter executable:

```
$ cd mpi4py-X.X.X
$ python setup.py build_exe [--mpi=<name>|--mpicc=/path/to/mpicc]
$ [sudo] python setup.py install_exe [--install-dir=$HOME/bin]
```

After the above steps you should have the MPI-enabled interpreter installed as *prefix*`/bin/python`*X.X*`-mpi` (or `$HOME/bin/python`*X.X*`-mpi`). Assuming that *prefix*`/bin` (or `$HOME/bin`) is listed on your `PATH`, you should be able to enter your MPI-enabled Python interactively, for example:

```
$ python2.7-mpi
Python 2.7.8 (default, Nov 10 2014, 08:19:18)
[GCC 4.9.2 20141101 (Red Hat 4.9.2-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.executable
'/usr/bin/python2.7-mpi'
>>>
```

## 8.2 Building MPI from sources

In the list below you have some executive instructions for building some of the open-source MPI implementations out there with support for shared/dynamic libraries on POSIX environments.

- *MPICH*

  ```
  $ tar -zxf mpich-X.X.X.tar.gz
  $ cd mpich-X.X.X
  $ ./configure --enable-shared --prefix=/usr/local/mpich
  $ make
  $ make install
  ```

- *Open MPI*

  ```
  $ tar -zxf openmpi-X.X.X tar.gz
  $ cd openmpi-X.X.X
  ```

(continues on next page)

(continued from previous page)

```
$ ./configure --prefix=/usr/local/openmpi
$ make all
$ make install
```

- *MPICH 1*

```
$ tar -zxf mpich-X.X.X.tar.gz
$ cd mpich-X.X.X
$ ./configure --enable-sharedlib --prefix=/usr/local/mpich1
$ make
$ make install
```

Perhaps you will need to set the `LD_LIBRARY_PATH` environment variable (using **`export`**, **`setenv`** or what applies to your system) pointing to the directory containing the MPI libraries . In case of getting runtime linking errors when running MPI programs, the following lines can be added to the user login shell script (`.profile`, `.bashrc`, etc.).

- *MPICH*

```
MPI_DIR=/usr/local/mpich
export LD_LIBRARY_PATH=$MPI_DIR/lib:$LD_LIBRARY_PATH
```

- *Open MPI*

```
MPI_DIR=/usr/local/openmpi
export LD_LIBRARY_PATH=$MPI_DIR/lib:$LD_LIBRARY_PATH
```

- *MPICH 1*

```
MPI_DIR=/usr/local/mpich1
export LD_LIBRARY_PATH=$MPI_DIR/lib/shared:$LD_LIBRARY_PATH:
export MPICH_USE_SHLIB=yes
```

**Warning:** MPICH 1 support for dynamic libraries is not completely transparent. Users should set the environment variable `MPICH_USE_SHLIB` to `yes` in order to avoid link problems when using the **`mpicc`** compiler wrapper.

## References

[mpi-std1] MPI Forum. MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications, volume 8, number 3-4, pages 159-416, 1994.

[mpi-std2] MPI Forum. MPI: A Message Passing Interface Standard. High Performance Computing Applications, volume 12, number 1-2, pages 1-299, 1998.

[mpi-using] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: portable parallel programming with the message-passing interface. MIT Press, 1994.

[mpi-ref] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI - The Complete Reference, volume 1, The MPI Core. MIT Press, 2nd. edition, 1998.

[mpi-mpich] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, 22(6):789-828, September 1996.

[mpi-openmpi] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004.

[Hinsen97] Konrad Hinsen. The Molecular Modelling Toolkit: a case study of a large scientific application in Python. In Proceedings of the 6th International Python Conference, pages 29-35, San Jose, Ca., October 1997.

[Beazley97] David M. Beazley and Peter S. Lomdahl. Feeding a large-scale physics application to Python. In Proceedings of the 6th International Python Conference, pages 21-29, San Jose, Ca., October 1997.

# Python Module Index

## m

# Index

## Symbols



## B



## C



## E



## L



## M



## O



## P



## S