MPI for Python

Release 1.3.1

Lisandro Dalcin

August 07, 2013

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 specifications and provides an object oriented interface which closely follows 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 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], Open MPI [mpi-openmpi] or LAM [mpi-lammpi].

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,

---

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

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 Design and Interface 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` (slower, written in pure Python) and `cPickle` (faster, written in C) modules provide user-extensible facilities to serialize generic 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 used-defined Python object taking advantage of the features provided by the mod:*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 generic 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 in 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*, `Comm` is the base class of communicators. The `Intracomm` and `Intercomm` classes are sublcasses of the `Comm` class. The `Is_inter()` method (and `Is_intra()`, provided for convenience, it is 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: `COMM_SELF` and `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 `Get_size()` and `Get_rank()`. The associated process group can be retrieved from a communicator by calling the `Get_group()` method, which returns an instance of the `Group` class. Set operations with `Group` objects like like `Union()`, `Intersect()` and `Difference()` are fully supported, as well as the creation of new communicators from these groups using `Create()`.

New communicator instances can be obtained with the `Clone()` method of `Comm` objects, the `Dup()` and `Split()` methods of `Intracomm` and `Intercomm` objects, and methods `Create_intercomm()` and `Merge()` of `Intracomm` and `Intercomm` objects respectively.

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

2.3 Point-to-Point Communications

Point to point communication is a fundamental capability of message passing systems. This mechanism enables the transmittal 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 `Send()`, `Recv()` and `Sendrecv()` methods of communicator objects provide support for blocking point-to-point communications within `Intracomm` and `Intercomm` instances. These methods can communicate memory buffers. The variants `send()`, `recv()` and `sendrecv()` can communicate generic 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 `Isend()` and `Irecv()` methods of the `Comm` class initiate a send and receive operation respectively. These methods return a `Request` instance, uniquely identifying the started operation. Its completion can be managed using the `Test()`, `Wait()`, and `Cancel()` methods of the `Request` class. The management of `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 `Send_init()` and `Recv_init()` methods of the `Comm` class create a persistent request for a send and receive operation respectively. These methods return an instance of the `Prequest` class, a subclass of the `Request` class. The actual communication can be effectively started using the `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.

MPI for Python provides support for almost all collective calls. Unfortunately, the `Alltoallw()` and `Reduce_scatter()` methods are curently unimplemented.

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

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

2.5 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 processes groups can be created by calling the `Spawn()` method within an intracommunicator (i.e., an `Intracomm` instance). This call returns a new intercommunicator (i.e., an `Intercomm` instance) at the parent process group. The child process group can retrieve the matching intercommunicator by calling the `Get_parent()` (class) method defined in the `Comm` class. 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 `Open_port()` function to open a *port* and the `Publish_name()` function to publish a provided *service*, and next call the `Accept()` method within an `Intracomm` instance. Any client applications can first find a published *service* by calling the `Lookup_name()` function, which returns the *port* where a server can be contacted; and next call the `Connect()` method within an `Intracomm` instance. Both `Accept()` and `Connect()` methods return an `Intercomm` instance. When connection between client/server processes is no longer needed, all of them must cooperatively call the `Disconnect()` method of the `Comm` class. Additionally, server applications should release resources by calling the `Unpublish_name()` and `Close_port()` functions.

2.6 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 `Win` class. New window objects are created by calling the `Create()` method at all processes within a communicator and specifying a memory buffer . When a window instance is no longer needed, the `Free()` method should be called.

The three one-sided MPI operations for remote write, read and reduction are available through calling the methods `Put()`, `Get()`, and `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 `Start()` and `Complete()` methods at the origin process, and target process cooperates by calling the `Post()` and `Wait()` methods. There is also a collective variant provided by the `Fence()` method. Passive target synchronization is more lenient, only the origin process calls the `Lock()` and `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.

2.7 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 forPython*, all MPI input/output operations are performed through instances of the `File` class. File handles are obtained by calling the `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 `Close()` method. Files even can be deleted by calling method `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 `Set_view()` and `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.

2.8 Environmental Management

Initialization and Exit

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

> **Caution:** `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 `Init()`/`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 Therefore, there is no need to call `Finalize()` from Python to ensure MPI finalization.

Implementation Information

- The MPI version number can be retrieved from module function `Get_version()`. It returns a two-integer tuple `(version,subversion)`.
- The `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 `Get_attr()` method within the `COMM_WORLD` instance.

Timers

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

Error Handling

Error handling functionality is almost completely supported. Errors originated in native MPI calls will raise an instance of the module exception class `Exception`, which is a subclass of the standard Python exception `RuntimeError`.

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

In order facilitate communicator sharing with other Python modules interfacing MPI-based parallel libraries, default MPI error handlers `ERRORS_RETURN`, `ERRORS_ARE_FATAL` can be assigned to and retrieved from communicators, windows and files with methods `{Class}.Set_errhandler()` and `{Class}.Get_errhandler()`.

3 Installation

3.1 Requirements

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

- A working MPI distribution, preferably a MPI-2 one 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.

- A Python 2.4 to 2.7 or 3.0 to 3.3 distribution, with Python library preferably built with shared/dynamic libraries.

  > **Note:** **Mac OS X** users employing a Python distribution built with **universal binaries** may need to temporarily set the environment variables `MACOSX_DEPLOYMENT_TARGET`, `SDKROOT`, and `ARCHFLAGS` to appropriate values in the shell before trying to build/install *MPI for Python*. Check the instructions at *Mac OS X and Universal/SDK Python builds* in the appendix.

  > **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.

3.2 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
> ```

3.3 Using distutils

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

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

- `--mpi=` : let you pass a section with MPI configuration within a special configuration file.

- `--configure` : runs exhaustive tests for checking about missing MPI types/constants/calls. This option should be passed in order to build *MPI for Python* against old MPI-1 implementations, possibly providing a subset of MPI-2.

Downloading

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

```
$ curl -O http://mpi4py.googlecode.com/files/mpi4py-X.X.X.tar.gz
```

```
$ wget http://mpi4py.googlecode.com/files/mpi4py-X.X.X.tar.gz
```

Building

After unpacking the release tarball:

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

the distribution is ready for building.

- If you use a MPI implementation providing a **mpicc** compiler wrapper (e.g., MPICH 1/2, Open MPI, LAM), 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 distribution by editing the file called mpi.cfg. You can use the default section [mpi] or add a new, custom section, for example [my_mpi] (see the examples provided in the mpi.cfg file):

  ```
  [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
```

Installing

After building, the distribution 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, you have two options depending on the target Python version.

- For Python 2.6 and up:

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

- For Python 2.5 and below (assuming your home directory is available through the HOME environment variable):

```
$ python setup.py install --home=$HOME
```

Finally, add $HOME/lib/python or $HOME/lib64/python to your PYTHONPATH environment variable.

Testing

Issuing at the command line:

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

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

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

will launch a five-process run of the Python interpreter and run the test scripts demo/helloworld.py.

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
```

4 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()`. Note that `isend()` is available, but `irecv()` is not.

  Collective calls like `scatter()`, `gather()`, `allgather()`, `alltoall()` expect/return 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`.

  Global reduction operations `reduce()` and `allreduce()` are naively implemented, the reduction is actually done at the designated root process or all processes.

- Communication of buffer-provider objects

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

  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 the `count`).

  Automatic MPI datatype discovery for NumPy 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.

4.1 Point-to-Point Communication

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

```python
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)
```

- NumPy arrays (the fast way!):

```python
from mpi4py import MPI
import numpy

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

# pass explicit MPI datatypes
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:
```

```python
data = numpy.empty(100, dtype=numpy.float64)
comm.Recv(data, source=0, tag=13)
```

4.2 Collective Communication

- Broadcasting a Python dictionary:

```python
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:

```python
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:

```python
from mpi4py import MPI

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
```

- Parallel matrix-vector product:

```python
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
```

4.3 Dynamic Process Management

Compute Pi

- Master (or parent, or client) side:

```python
#!/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()
```

- Worker (or child, or server) side:

```python
#!/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()
```

4.4 Wrapping with SWIG

- C source:

```c
/* 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.
```

4.5 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
```

- 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.
```

5 Appendix

5.1 MPI-enabled Python interpreter

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:

```
$ python2.6-mpi
Python 2.6 (r26:66714, Jun  8 2009, 16:07:26)
[GCC 4.4.0 20090506 (Red Hat 4.4.0-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.executable
'/usr/bin/python2.6-mpi'
>>>
```

5.2 Mac OS X and Universal/SDK Python builds

Mac OS X users employing a Python distribution built with support for Universal applications could have trouble building *MPI for Python*, specially if they want to link against MPI libraries built without such support. Another source of trouble could be a Python build using a specific *deployment target* and *cross-development SDK* configuration. Workarounds for such issues are to temporarily set the environment variables MACOSX_DEPLOYMENT_TARGET, SDKROOT and/or ARCHFLAGS to appropriate values in the shell before trying to build/install *MPI for Python*.

An appropriate value for MACOSX_DEPLOYMENT_TARGET should be any greater or equal than the one used to build Python, and less or equal than your system version. The safest choice for end-users would be to use the system version (e.g, if you are on *Leopard*, you should try MACOSX_DEPLOYMENT_TARGET=10.5).

An appropriate value for SDKROOT is the full path name of any of the SDK's you have at /Developer/SDKs directory (e.g., SDKROOT=/Developer/SDKs/MacOSX10.5.sdk). The safest choice for end-users would be the one matching the system version; or alternatively the root directory (i.e., SDKROOT=/).

Appropriate values for ARCHFLAGS have the form -arch <value>, where <value> should be chosen from the following table:

@	Intel	PowerPC
32-bit	i386	ppc
64-bit	x86_64	ppc64

For example, assuming your Mac is running **Snow Leopard** on a **64-bit Intel** processor and you want to override the hard-wired cross-development SDK in Python configuration, you can build and install *MPI for Python* using any of the alternatives below. Note that environment variables may need to be passed/set both at the build and install steps (because **sudo** may not pass environment variables to subprocesses for security reasons)

- Alternative 1:

```
$ env MACOSX_DEPLOYMENT_TARGET=10.6 \
      SDKROOT=/                      \
      ARCHFLAGS='-arch x86_64'        \
      python setup.py build [options]

$ sudo env MACOSX_DEPLOYMENT_TARGET=10.6 \
           SDKROOT=/                      \
           ARCHFLAGS='-arch x86_64'        \
           python setup.py install [options]
```

- Alternative 2:

```
$ export MACOSX_DEPLOYMENT_TARGET=10.6
$ export SDKROOT=/
$ export ARCHFLAGS='-arch x86_64'
$ python setup.py build [options]

$ sudo -s # enter interactive shell as root
$ export MACOSX_DEPLOYMENT_TARGET=10.6
$ export SDKROOT=/
$ export ARCHFLAGS='-arch x86_64'
$ python setup.py install [options]
$ exit
```

5.3 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 2*

```
$ tar -zxf mpich2-X.X.X.tar.gz
$ cd mpich2-X.X.X
$ ./configure --enable-shared --prefix=/usr/local/mpich2
$ make
$ make install
```

- *Open MPI*

```
$ tar -zxf openmpi-X.X.X tar.gz
$ cd openmpi-X.X.X
$ ./configure --prefix=/usr/local/openmpi
$ make all
$ make install
```

- *LAM/MPI*

```
$ tar -zxf lam-X.X.X.tar.gz
$ cd lam-X.X.X
$ ./configure --enable-shared --prefix=/usr/local/lam
$ make
$ 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 2*

```
MPI_DIR=/usr/local/mpich2
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
```

- *LAM/MPI*

```
MPI_DIR=/usr/local/lam
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.

[mpi-lammpi] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In Proceedings of Supercomputing Symposium, pages 379-386, 1994.

[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.