MPI For Python Documentation

Release 0.6.0

Lisandro Dalcin, Rodrigo Paz, Mario Storti

September 16, 2008

CONTENTS

Authors Lisandro Dalcín, Rodrigo Paz, Mario Storti

Organization CIMEC

Address PTLC, (3000) Santa Fe, Argentina

Contact dalcinl@gmail.com

Web Site http://mpi4py.scipy.org

Date September 16, 2008

Revision 0.6.0

Abstract This document describes *MPI for Python* package. It 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 specification 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.

CHAPTER

ONE

Contents

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

1.1.1 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.lam].

1.1.2 What is Python?

Python is a modern, easy to learn, powerful programming language. It has efficient high-level data structures and a

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

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 [SPaSM] [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.

1.1.3 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++ 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.
- 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.

1.2 Design and Interface Overview

MPI for Python provides an object oriented approach to message passing and is based in the MPI-2 C++ bindings. However, some abstractions (like `Port` class) and functionalities (like sending and receiving streams in `Port` instances) are added in order to emulate OOMPI library interface.

The design is simple but effective. The MPI module consists of Python code defining all constants, class hierarchies and functions. This code calls simpler functions from extension modules written in C, which provide access to native MPI-1/MPI-2 handles, constants and function in the C side.

Following pyMPI and Pypar approaches, any Python object to be transmitted is first serialized at sending processes using the standard Python module `cPickle`. After that, string data is communicated (using `MPI_CHAR` datatype). Finally, received strings are unpacked and the original objects are restored at the reciever processes.

The pickling/unpickling appoach can impose important overheads in memory as well as processor usage, specially in the case of communication of objects containing big memory buffers, like long strings or NumPy arrays. In latest releases, MPI for Python was improved to support direct communication of any object exporting single-segment buffer interface. This new feature, and the posibility of constructing user-defined datatypes describing complicated memory layouts, enables the implementation of many numerical applications directly in Python with negligible overhead, almost as fast as compiled C/C++/Fortran codes.

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.

1.2.1 Communicators

Class `Comm` is a base class for `Intracomm` and `Intercomm` classes. Method `Is_inter()` (also `Is_intra()`, nonstandard but provided for convenience) is defined for communicator objects and can be used to determine the particular communicator class.

Two predefined intracommunicators instances are available: `COMM_WORLD` and `COMM_SELF` (or `WORLD` and `SELF`, provided for convenience), which are in fact communicators obtained by duplication of `MPI_COMM_WORLD` and `MPI_COMM_SELF`. The original predefined MPI communication domains can be accessed via `__COMM_WORLD__` and `__COMM_SELF__` instances, but this is discouraged in order to avoid any message conflicts with other modules calling MPI functions.

Communicator size and process rank can be respectively obtained with `Get_size()` and `Get_rank()` methods. Module constants `WORLD_RANK` and `WORLD_SIZE` are convenience shortcuts for `COMM_WORLD.Get_rank()` and `COMM_WORLD.Get_size()`.

Communicator comparisons can be done with (static) method `Compare()` of `Comm` class, which returns a value in module constants `IDENT`, `CONGRUENT`, `SIMILAR` or `UNEQUAL`.

New communicator instances can be obtained with method `Clone()` of `Comm` objects, methods `Dup()` and `Split()` of both `Intracomm` and `Intercomm` objects, and methods `Create_intercomm()` and `Merge()` of `Intracomm` and `Intercomm` objects respectively. Set operations with `Group` objects like like `Union()`, `Intersect()` and `Difference()` are fully supported, as well as the creation of new communicators from groups.

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

Point-to-Point Communications

Methods `Send()`, `Recv()` and `Sendrecv()` of `Comm` class provide support for blocking point-to-point communications.

Non-blocking communications are only supported for objects exporting the single-segment buffer interface. `Request` instances are returned by `Isend()` and `Irecv()` methods of `Comm` class. Persistent communications are also supported. `Prequest` instances are returned by `Send_init()` and `Recv_init()` methods of `Comm` class.

Collective Communications

Methods `Bcast()`, `Scatter()`, `Gather()`, `Allgather()` and `Alltoall()` of communicator objects provide support for collective communications. Global reduction operations `Reduce()`, `Allreduce()`, `Scan()` and `Exscan()` are supported, but they are naively implemented for reductions of general Python objects.

1.2.2 One-Sided Communications

Class `Win` provides all the MPI-2 features for one-sided communications (also known as *remote memory access (RMA)*). Methods `Put()`, `Get()`, and `Accumulate()` can be used for remote writes, reads, and reductions. All synchronization calls (fence, active target, and lock) are fully supported.

1.2.3 I/O

Class `File` provides all the MPI-2 features for parallel input/output. All data access operations, for all kind of positioning (explicit offsets, individual file pointers, and shared file pointers), synchronism (bloking, nonblocking, and split collective), and coordination (noncollective and collective) are fully supported.

1.2.4 Environmental Management

- *Initialization and Exit*

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

- *Implementation Information*

  - The MPI version number can be retrieved from module function `Get_version()`. It returns the tuple `(version,subversion)`.
  - Communicator attributes are not currently supported. However, `MPI_COMM_WORLD` standard attributes can be accessed from module constants `TAG_UB`, `HOST`, `IO` and `WTIME_IS_GLOBAL`. Module function `Get_processor_name()` and module constant `PROCESSOR_NAME` can be used to access the processor name.

- *Timers*

  MPI timer functionality is available through module functions `Wtime()` and `Wtick()`. Module constant `WTIME_IS_GLOBAL` indicates whether clocks at all processes in `COMM_WORLD` communicator are synchronized.

- *Error Handling*

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

  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 `Set_errhandler()` and `Get_errhandler()`.

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

1.2.5 Extensions

MPI for Python adds some extensions to the standard MPI syntax. The rationale is simplified usage and conformance with some Python idioms and facilities.

An elegant abstraction for message-passing borrowed from OOMPI is introduced: communicators can be seen as containers of *ports*}. A port is a tiny object with references to a communication domain and a process *id*. This *id* is used as source or destination process in point-to-point communications, or root process in collective communications. Communicators can now be treated as container of `Port` instances and indexing/iteration can be defined for them. Data streams can be messaged between `Port` instances using << and >> operators.

Accessors methods for different objects are mapped to *properties*, i.e., managed attributes. For example, communicator rank and size of `COMM_WORLD` can be directly obtained with `COMM_WORLD.rank` and `COMM_WORLD.size` instead of calling `Get_rank()` and `Get_size()` methods.

Some constants are added for convenience. Integers `rank` and `size` are shortcuts for the accessor methods `Get_rank()` and `Get_size()` of `COMM_WORLD` instance. Booleans `zero`, `last`, `even` and `odd` have values related to process rank in `COMM_WORLD`.

There is also support for transparently passing any MPI objects to other modules generated with SWIG.

1.2.6 Documentation

The standard Python on-line help mechanism will provide information about module constants, classes and functions using their documentation strings.

1.3 Downloads

MPI For Python is available for download at the Python Package Index .

1.4 Installation

1.4.1 Requirements

You need to have the following software properly installed in order to build the MPI module and the companion parallelized version of the Python interpreter:

- A Python 2.3/2.4/2.5 distribution, with Python library preferably compiled as a dynamic library.

  Note: It is suggested to use a MPI-parallelized version of the Python interpreter supporting interactive parallel sessions.

- A working MPI distribution for your architecture, compiled with dynamic libraries. For example, on a GNU/Linux box this requirement can be accomplished by typing:

  – *MPICH 2*

  ```
  $ tar -zxf mpich2-X.X.X.tar.gz
  $ cd mpich2-X.X.X
  $ ./configure --enable-sharedlibs=gcc --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*

  ```
  $ 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/mpich
  $ make
  $ make install
  ```

  Note: Perhaps the user will have to set his `LD_LIBRARY_PATH` environmental variable (using `export`, `setenv` or what applies to his system) pointing to the MPI library directory. In case of getting runtime linking error 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=$LD_LIBRARY_PATH:$MPI_DIR/lib
  ```

  – *Open MPI*

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

– *LAM*

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

– *MPICH 1*

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

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

1.4.2 Building

After downloading and unpacking the module distribution:

```
$ 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, OpenMPI, 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 `mpi.cfg`. You can use the default section `[mpi]` or add a new, custom section, for example `[my_mpi]` (see the examples provided in `mpi.cfg`):

  ```
  [mpi]

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

  [my_mpi]

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

```
...
```

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

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

1.4.3 Installing

After building, the distribution is ready for install (perhaps you will need root privileges):

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

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

1.4.4 Testing

Issuing at the command line:

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

or:

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

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

You can also try *unittest* scripts located at `tests/unittest`.

1.5 Tutorial (dated)

1.5.1 Invoking the interpreter

In the following examples, we assume that a parallelized version of the Python interpreter was launched using the implementation-provided MPI startup mechanism. With the MPICH distribution on a cluster of five PC's running GNU/Linux and listed in file `nodes.dat`, this can be accomplished by typing:

```
$ mpirun -np 5 -machinefile nodes.dat <prefix>/bin/bwpython -i
```

from a command-line shell environment like bash. After that, the `MPI` module can be imported by typing:

```
>>> from mpi4py import MPI
```

1.5.2 Hello World!

By typing the following sentences in the Python prompt, output from all processes should be obtained.

```
>>> rank, size = MPI.COMM_WORLD.rank, MPI.COMM_WORLD.size
>>> print 'Hello World! I am process', rank, 'of', size
Hello World! I am process 0 of 5
Hello World! I am process 4 of 5
Hello World! I am process 2 of 5
Hello World! I am process 1 of 5
Hello World! I am process 3 of 5
```

1.5.3 Point-to-Point communications

First, we prepare some different data in each process.

```
>>> rank, size = MPI.COMM_WORLD.rank, MPI.COMM_WORLD.size
>>> sendbuf = 10 * size + rank
>>> print '[%d]' % rank, sendbuf
[0] 50
[4] 54
[2] 52
[1] 51
[3] 53
```

Next, we can try some point-to-point communications.

- using standard form ...

```
>>> MPI.COMM_WORLD.Send(sendbuf, dest=0, tag=7)
>>> recvbuf = []
>>> if MPI.COMM_WORLD.rank == 0:
...     for i in xrange(MPI.COMM_WORLD.size):
...         data = MPI.COMM_WORLD.Recv(source=i, tag=7)
...         recvbuf.append(data)
...
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, recvbuf)
[0] [50, 51, 52, 53, 54]
[4] []
[2] []
[1] []
[3] []
```

- using ports ...

```
>>> MPI.COMM_WORLD[0].Send(sendbuf) # get port 0, send with default tag
>>> recvbuf = []
>>> if MPI.COMM_WORLD.rank == 0:
...     for p in MPI.COMM_WORLD:          # iterate over comm
...         recvbuf += [ p.Recv() ] # recv with default tag
...
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, recvbuf)
[0] [50, 51, 52, 53, 54]
[4] []
[2] []
[1] []
[3] []
```

- using ports with stream syntax ...

```
>>> MPI.COMM_WORLD[0] << [ sendbuf ] # input stream must be list !!!
>>>
>>> recvbuf = [] # output stream must be list, too !!!
>>> if MPI.COMM_WORLD.rank == 0:
...     for p in MPI.COMM_WORLD:
...         p >> recvbuf
...
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, recvbuf)
[0] [50, 51, 52, 53, 54]
[4] []
[2] []
[1] []
[3] []
```

1.5.4 Collective communications

- broadcast using standard form ...

```
>>> sendbuf = MPI.COMM_WORLD.rank**2 # square of process rank
>>>
>>> root = MPI.COMM_WORLD.size-1 # last process
>>> recvbuf = MPI.COMM_WORLD.Bcast(sendbuf, root)
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, recvbuf)
[0] 16
[1] 16
[2] 16
[3] 16
[4] 16
```

- gather using ports ...

```
>>> rank, size = MPI.COMM_WORLD.rank, MPI.COMM_WORLD.size
>>> sendbuf = [ rank**2 , rank%2!=0 ]
>>> print '[%d] %s' % (rank, sendbuf)
[0] [0, False]
[1] [1, True]
[2] [4, False]
[3] [9, True]
[4] [16, False]
>>>
>>> root = size//2 # middle process
>>> recvbuf = MPI.COMM_WORLD[root].Gather(sendbuf)
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, recvbuf)
[0] None
[1] None
[2] [[0, False], [1, True], [4, False], [9, True], [16, False]]
[3] None
[4] None
```

- scatter using ports ...

```
>>> rank, size = MPI.COMM_WORLD.rank, MPI.COMM_WORLD.size
>>> root = size//2 # middle process
>>>
>>> sendbuf = None
>>> if rank == root: # set data in root
...     sendbuf = [ (i, i**2, i**3) for i in range(size) ]
>>> print '[%d] %s' % (rank, sendbuf)
[0] None
[1] None
[2] [(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
[3] None
[4] None
>>>
>>> p = MPI.COMM_WORLD[root]
>>> recvbuf = p.Scatter(sendbuf)
>>> print '[%d] %s' % (rank, recvbuf)
[0] (0, 0, 0)
[1] (1, 1, 1)
[2] (2, 4, 8)
[3] (3, 9, 27)
[4] (4, 16, 64)
```

1.5.5 Communicators

- duplication and comparison ...

```
>>> MPI.rprint([MPI.IDENT, MPI.CONGRUENT, MPI.SIMILAR, MPI.UNEQUAL])
[1, 2, 3, 4]
>>>
>>> comm = MPI.COMM_WORLD.Dup();
>>> rslt = MPI.Comm.Compare(comm, MPI.COMM_WORLD)
>>> congruent = (rslt==MPI.CONGRUENT)
>>> print '[%d] congruent: %s' % (MPI.COMM_WORLD.rank, congruent)
[0] congruent: True
[1] congruent: True
[2] congruent: True
[3] congruent: True
[4] congruent: True
>>>
>>> flag = ( comm == MPI.COMM_WORLD )
>>> print '[%d] %s' % (MPI.COMM_WORLD.rank, flag)
[0] True
[1] True
[2] True
[3] True
[4] True
```

- splitting ...

```
>>> rank, size = MPI.COMM_WORLD.rank, MPI.COMM_WORLD.size
>>> color = int(rank < size//2)
>>> key   = size-rank
>>> fmt = '[%d] color: %s - key: %s'
>>> print fmt % (MPI.COMM_WORLD.rank, color, key)
```

```
[0] color: 1 - key: 5
[1] color: 1 - key: 4
[2] color: 0 - key: 3
[3] color: 0 - key: 2
[4] color: 0 - key: 1
>>>
>>> comm = MPI.COMM_WORLD.Split(color, key)
>>> rk = comm.Get_rank()
>>> sz = comm.Get_size()
>>>
>>> fmt = '[%d] rank: %d - size: %d'
>>> pprint fmt % (rank, rk, sz)
[0] rank: 1 - size: 2
[1] rank: 0 - size: 2
[2] rank: 2 - size: 3
[3] rank: 1 - size: 3
[4] rank: 0 - size: 3
```

CHAPTER

TWO

Indices and tables

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

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

[SPaSM] Parallel Molecular Dynamics Code, http://bifrost.lanl.gov/MD/MD.html, 1994-2001.

[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.