Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 1108–1115
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

MPI for Python

Lisandro Dalcín*, Rodrigo Paz, Mario Storti

Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC), Instituto de Desarrollo Tecnológico para la Industria Química (INTEC), Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICET), Universidad Nacional del Litoral (UNL) (3000) Santa Fe, Argentina

Received 7 January 2005; accepted 23 March 2005
Available online 14 June 2005

Abstract

MPI for Python provides bindings of the Message Passing Interface (MPI) standard for the Python programming language and allows any Python program to exploit multiple processors. This package is constructed on top of the MPI-1 specification and defines 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 general Python objects. Efficiency has been tested in a Beowulf class cluster and satisfying results were obtained. MPI for Python is open source and available for download on the web (`http://www.cimec.org.ar/python`).

Keywords: Message passing; MPI; High level languages; Parallel Python

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 [5] 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 [18] 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.

MATLAB is the dominant interpreted programming language for implementing general numerical computations. In the open source side, Octave and Scilab are well known, freely distributed software packages providing compatibility with MATLAB language. Even in a specialized application domain like multi-physics simulations by finite element methods, the developers of *OOFELIE* [24,7] toolkit had early realized the importance of providing end-users with a high-level, interpreted, interactive language in order to simplify the access to an object oriented library written in C++.

In this work, we describe our experiences using Python, a well-established interpreted programming language, in parallel environments. We also present MPI for Python, a new package enabling general applications to exploit multiple

* Corresponding author. Fax: +54 342 4511594.
E-mail addresses: dalcinl@intec.unl.edu.ar (L. Dalcín), rodrigop@intec.unl.edu.ar (R. Paz), mstorti@intec.unl.edu.ar (M. Storti).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.03.010

processors using standard MPI "look and feel" in Python scripts.

The next section presents a brief overview of MPI, Python and related work. Section 3 describes design, implementation and provided functionality of MPI for Python. Section 4 presents some efficiency comparisons between MPI for Python and C codes communicating numeric arrays. Finally, Section 5 presents our conclusions and plans for future work.

2. Background

2.1. What is MPI?

MPI [20,21], 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 (MPI is not a programming language extension) and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++).

Since its release, the MPI specification has become the leading standard for message-passing libraries in the world of parallel computers. Implementations are available from vendors of high-performance computers as a component of the system software, and also from well known open source projects like MPICH [22,10] and LAM [17,6].

MPI follows an object oriented design. Among the different abstractions introduced, *communicators* play the most important role. Basically, communicators specify a communication domain between an ordered set of processes or *group*. This abstraction enables division of processes, avoids message conflicts between different modules, and permits extensibility by users.

2.2. What is Python?

Python [31,32] is a modern but mature, easy to learn, powerful programming language with a constantly growing community of users. It has efficient high-level data structures and a simple but effective approach to object-oriented programming with dynamic typing and dynamic binding. 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 its extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. It can be easily extended with new functions and data types implemented in C or C++ [33] and is also suitable as an extension language for customizable applications that require a programmable interface.

Python is an ideal candidate for writing higher-level parts of large-scale scientific applications and driving simulations in parallel architectures [30,4,12] like clusters of PCs or SMPs. Python codes are quickly developed, easily maintained, and can achieve a high degree of integration with other libraries written in compiled languages.

2.3. Related works

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

OOMPI [27] is an excellent C++ class library specification layered on top of the C bindings encapsulating MPI into a functional class hierarchy. This library provides a flexible and intuitive interface by adding some abstractions, like *Ports* and *Messages*, which enrich and simplify the syntax.

Pypar [23] is a rather minimal Python interface to MPI. 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 Python objects of any type can be communicated. There is also good support for communicating numeric arrays and practically full MPI bandwidth can be achieved.

pyMPI [19] rebuilds the Python interpreter and adds a built-in module for message passing. It permits interactive parallel runs, which are useful for learning and debugging, and provides an interface suitable for basic parallel programming. There is no full support for defining new communicators, process topologies or intracommunicators. General Python objects can be messaged between processors, but there is no support for direct communication of numeric arrays.

Scientific Python [13] provides a collection of Python modules that are useful for scientific computing. Among them, there are interfaces to MPI and BSP (*Bulk Synchronous Parallel programming*). The MPI interface is simple but incomplete and does not resemble the MPI specification. However, there is support for communicating numeric arrays.

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

Numeric (discontinued) and *Numarray* [9] (a reimplementation) are extension modules that provide array manipulation and computational capabilities similar to those found in IDL, MATLAB, or Octave. Using Numarray, it is possible to write many efficient numerical data processing applications directly in Python without using any C, C++ or Fortran code.

Pyfort [8] is a tool for connecting Fortran routines (and "Fortran-like" C) to Python and Numeric. Pyfort translates a module file that describes the routines to access from Python into a C language source file defining a Python module.

SciPy [14] 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 [3] 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.

3. Implementation

Python has enough networking capabilities as to successfully develop an implementation of MPI in "pure Python", i.e., without using compiled languages or third-party MPI distributions. The main advantage of such kind of implementation is surely portability. As an example of this approach, MatlabMPI [16] is a MPI toolbox for MATLAB written in "pure MATLAB" and based in built-in file I/O capabilities, without relying on any foreign language or library. However, Python is really easy to extend and connect with external software components developed in compiled languages. Additionally, there are many useful and high-quality MPI-based parallel libraries to justify some extra complexities. The development of an MPI package based in calls to a third-party MPI implementation will sensibly ease the integration of other parallel tools in Python.

3.1. Parallelization

Python and MPI distributions can be optionally built as shared libraries in recent operating systems supporting dynamic linking. After that, the Python interpreter can be easily enabled to run scripts in parallel and support extension modules calling MPI functions.

The following C code shows a basic (it should be improved with some error checking) Python parallelization. After compiling this source and linking it with MPI and Python libraries, the resulting executable will be a fully functional Python interpreter providing MPI initialization before the Python main program is called. [1]

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

int main(int argc, char **argv)
{
  int status;
  MPI_Init(&argc, &argv);
  status = Py_Main(argc, argv);
  MPI_Finalize();
  return status;
}
```

In order to build a Python interpreter capable of providing interactive sessions in parallel, the mechanism for obtaining input from the user should be modified. There are different alternatives, but the simplest one is overriding the function called by the parser to get its input in interactive sessions. Basically, the new function should call the original one in the root process (any one with input/output capabilities) and broadcast the input obtained from the user (actually, string data) while other processes different from root are waiting for the arriving of broadcast-ed data.

[1] Early MPI initialization is a requirement of some MPI-1 implementations, e.g. MPICH, which need to access original command line arguments in order to successfully start parallel jobs. MPI-2 suggests to implementors not to follow this approach to assure code portability.

3.2. Design

In Section 2.3 some previous attempts of integrating MPI and Python were mentioned. However, all of them lack from completeness or interface conformance with the standard specification. MPI for Python provides an interface designed with focus on translating MPI syntax and semantics from standard MPI-2 [21,25] C++ bindings to Python. As syntax translation from C++ to Python is generally straightforward, any user with some knowledge of those C++ bindings should be able to use this package without need of learning a new interface.

MPI for Python is implemented with Python code defining MPI constants, class hierarchies and functionalities. This code calls wrapper functions from an extension module written in C, which provides access to MPI-1 [20,11] handles, constants and functions in the native C implementation.

Following previous approaches already mentioned in Section 2.3, any Python object to be transmitted is first serialized using the module `cPickle` from Python standard library. After that, string data is communicated (using `MPI_CHAR` datatype). Finally, received strings are unpacked and the original object is restored. Serialization process introduces some overheads: dynamic memory allocations, heavier messages and extra processing. However, this methodology is easily implemented and quite general. Direct communication, i.e., without serialization, of consecutive numeric arrays is feasible but not currently supported. This issue will be addressed in the near future.

3.3. Interface overview

3.3.1. Communicators

`Comm` is the base class of communicators. Communicator size and calling process rank can be, respectively, obtained with methods `Get_size()` and `Get_rank()`. Communicator comparisons can be done with (static) method `Compare()`, which returns one of the integer values `IDENT`, `CONGRUENT`, `SIMILAR`, or `UNEQUAL`.

`Intracomm` and `Intercomm` classes are derived from `Comm` class. Method `Is_inter()` (and `Is_intra()`, 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). In fact, they are ob-

tained by duplication of MPI_COMM_WORLD and MPI_COMM_SELF. Nevertheless, the original predefined MPI communication domains can be accessed via __COMM_WORLD__ and __COMM_SELF__ instances, but this not recommended as it may cause message conflicts with other modules calling MPI functions.

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

Virtual topologies (Cartcomm and Graphcomm classes, both being a specialization 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.

3.3.2. Point-to-point and collective communications

Methods Send(), Recv() and Sendrecv() of communicator objects provide support for blocking point-to-point communications within Intracomm and Intercomm instances. Non-blocking communications are not currently supported. Nevertheless, Request class and some of its methods are already implemented.

Methods Bcast(), Scatter(), Gather(), Allgather() and Alltoall() of Intracomm instances provide support for collective communications.

Global reduction operations Reduce(), Allreduce() and Scan() are supported but naively implemented.

3.3.3. Environmental management

Functions Init() and Finalize(), respectively, provide MPI initialization and finalization. Functions Is_initialized() and Is_finalized() provide the respective tests for initialization and finalization.

MPI version number supported for the underlying MPI implementation can be retrieved from function Get_version(). Standard attributes of MPI_COMM_WORLD can be accessed from constants TAG_UB, HOST, IO and WTIME_IS_GLOBAL. Function Get_processor_name() returns the calling processor name. Timer functionalities are available through functions Wtime() and Wtick(), constant WTIME_IS_GLOBAL indicates whether clocks at all processes in COMM_WORLD communicator are synchronized.

Error handling functionality is almost completely supported. Errors originated in native MPI calls will throw an instance of the exception class Exception, which derives from standard exception RuntimeError. In order to facilitate the sharing of communicators with other modules interfacing MPI-based parallel libraries, default MPI error handlers ERRORS_RETURN and ERRORS_ARE_FATAL can be assigned to and retrieved from communicators with methods Set_errhandler() and Get_errhandler().

3.3.4. Extensions

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

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 the process rank in COMM_WORLD.

3.3.5. Documentation

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

4. Efficiency

Some efficiency tests were run on the Beowulf class cluster *Geronimo* [28] at CIMEC. Hardware consisted of ten computing nodes with Intel P4 2.4 Ghz processors, 512 KB cache size, 1024 MB RAM DDR 333 MHz and 3COM 3c509 (Vortex) Nic cards interconnected with an Encore ENH924-AUT+ 100 Mbps Fast Ethernet switch. MPI for Python was compiled with *MPICH 1.2.6* and *Python 2.3*, *Numarray 1.1* was also used.

The first test was a bi-directional blocking send and receive between pairs of processors. Messages were numeric arrays (*NumArray* objects) of double precision (64 bits) floating-point values. A basic implementation of this test using MPI for Python (translation to C or C++ is straightforward) is shown below.

```
from mpi4py import mpi
import numarray as na

sbuff  = na.array(shape=2**20,
                  type= na.Float64)
wt = mpi.Wtime()
if mpi.even:
```

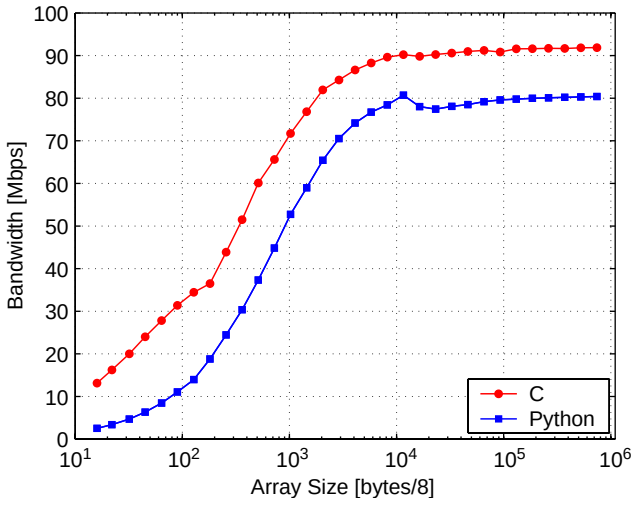

Fig. 1. Bandwidth in blocking send/receive.

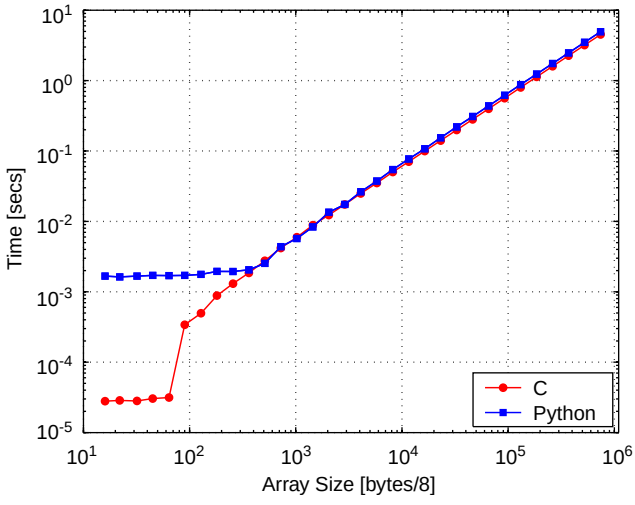

Fig. 2. Timing in broadcast.

```
   mpi.WORLD.Send(buffer,mpi.rank+1)
   rbuff = mpi.WORLD.Recv(mpi.rank+1)
else:
   rbuff = mpi.WORLD.Recv(mpi.rank-1)
   mpi.WORLD.Send(buffer,mpi.rank-1)
wt = mpi.Wtime() - wt

tp = mpi.WORLD.Gather(wt, root=0)
if mpi.zero: print tp
```

Results are shown in Fig. 1. Maximum bandwidth in Python is about 85% of maximum bandwidth in C. Clearly, the overhead introduced by object serialization degrades overall efficiency.

The second test consisted in wall-clock time measurements of some collective operations on 10 uniprocessor nodes. Messages were again numeric arrays of double precision floating-point values. Results are shown in Figs. 2–6. For array sizes greater than 10^3 (8 KB), timings in Python are between 5% (for *Bcast*) to 20% (for *Alltoall*) greater than timings in C.

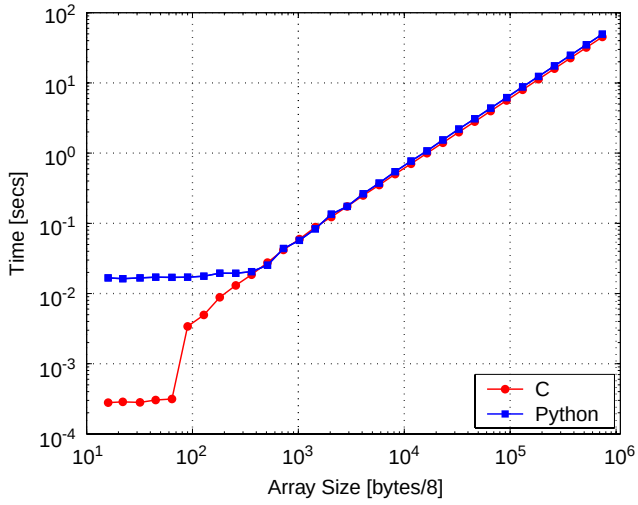

Fig. 3. Timing in scatter.

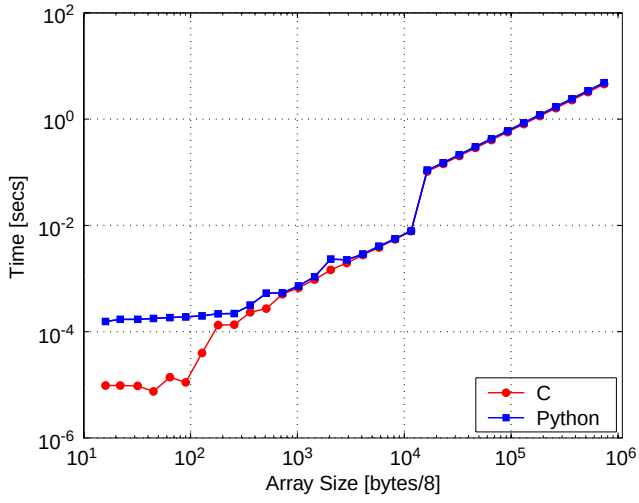

Fig. 4. Timing in gather.

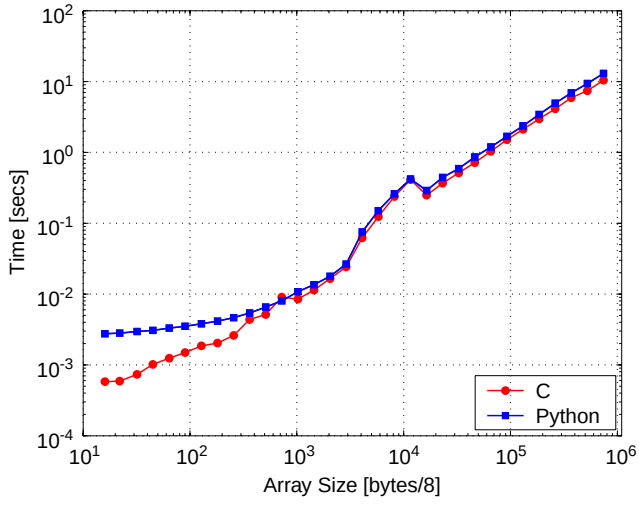

Fig. 5. Timing in gather to all.

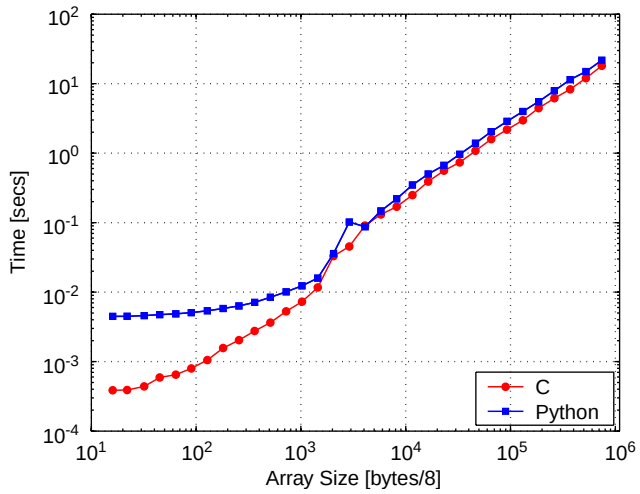

Fig. 6. Timing in all to all scatter/gather.

5. Conclusions and future work

Python is a very attractive language for rapid development of small scripts and code prototypes as well as large applications and highly portable and reusable modules and libraries. Unfortunately, like any scripting language, Python is not as efficient as compiled languages. However, it was conceived and carefully developed to be extensible in C (and consequently in C++). This exceptional characteristic enables Python to achieve performance in the time-critical parts of demanding applications. Moreover, Python can be used as a glue language capable of connecting existing software components in a high-level, interactive, and productive environment.

Running Python on parallel computers is a good starting point for decreasing the large software costs of using HPC systems. MPI for Python provides the base layer for applying the message-passing paradigm. This package is just another example of successful interconnection of Python with an external library, in this case MPI; its interface was designed to be as conforming as possible with standard MPI for C++, saving users from learning a new interface specification. Additionally, MPI for Python can be used for learning and teaching message-passing parallel programming taking advantage of Python's interactive nature. Efficiency tests have shown that performance degradation is not prohibitive, even for moderately sized objects. In fact, the overhead introduced by MPI for Python is far smaller than the normal one associated to the use of interpreted versus compiled languages.

Future work will be directed towards the improvement of MPI for Python by adding some currently unsupported functionalities like non-blocking communications and direct communication of numeric arrays. We will also develop Python packages providing access to very well known and widely used MPI-based parallel libraries like PETSc [1,2] and ParMETIS [15]. Furthermore, the higher-level portions of the parallel multi-physics finite elements code PETSc-FEM [29,26] developed at CIMEC are planned to be implemented in Python in the next major rewrite. This work has already started and preliminary results are promising.

Acknowledgments

This work received financial support from *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina), *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT) and *Universidad Nacional del Litoral* (UNL) through grants CONICET-PIP-198 *Germen-CFD*, ANPCyT-PID-99/74 *FLAGS*, ANPCyT-FONCyT-PICT-6973 *PROA* and CAI+D-UNL-PIP-02552-2000.

The authors make extensive use of freely available software such as GNU/Linux operating system, GCC compilers, Python, Perl, MPICH and LAM/MPI implementations, PETSc libraries, METIS/ParMETIS libraries, Octave, OpenDX and others. Many thanks to open source community for those excellent products.

Appendix A. Examples

A.1. Startup

In the following examples, we assume that a parallelized, interactive version of the Python interpreter was launched using the startup mechanism provided by the MPI implementation. As an example, using LAM on a cluster of PCs running GNU/Linux and listed in file `nodes.dat`, this step involves:

```
$ lamboot nodes.dat
$ mpirun -np 3 ppython
>>> from mpi4py import mpi
```

As an initial test, by typing the following sentences in the Python prompt, output from all processes should be obtained.

```
>>> print 'Hello World! I am process', \
... mpi.rank, 'of', mpi.size
Hello World! I am process 0 of 3
Hello World! I am process 1 of 3
Hello World! I am process 2 of 3
```

A.2. Point-to-point communications

First, we prepare some different integer data in each process send it to process with rank 0 in `COMM_WORLD`.

```
>>> sendbuf = 10*mpi.size + mpi.rank
>>> print '[%d]' % mpi.rank, sendbuf
[0] 30
[1] 31
[2] 32
>>>mpi.WORLD.Send(sendbuf,dest=0,tag=7)
```

Next, process 0 receives data from other processes.

```
>>> recvbuf = []
>>> if mpi.WORLD.Get_rank() == 0:
...   for i in xrange(mpi.size):
...     data = mpi.WORLD.Recv(source=i,
tag=7)
...     recvbuf.append(data)
...
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] [30, 31, 32]
[1] []
[2] []
```

Similar results can be achieved by using "stream" syntax. In this case, input and output streams must be list instances.

```
>>>sendbuf=100*mpi.size+(mpi.rank+1)**2
>>> mpi.WORLD[0] << [ sendbuf ]
>>>
>>> recvbuf = []
>>> if mpi.rank == 0:
...   for p in mpi.WORLD:
...     p >> recvbuf
...
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] [101, 104, 109]
[1] []
[2] []
```

A.3. Collective communications

The following code generates a dictionary in process with rank 0 and broadcasts it to other processes in `COMM_WORLD`:

```
>>> if mpi.rank == 0:
...  sendbuf = { 'foo' : True, \
...              'bar' :
3.14, \
...              'spam' : 'yes' }
... else:
...  sendbuf = None
...
>>> print '[%d]' % mpi.rank, sendbuf
[0] {'foo': True, 'spam': 'yes', 'bar':
3.14}
[1] None
[2] None
>>>
>>> recvbuf = mpi.WORLD.Bcast(sendbuf,
root=0)
>>> print '[%d]' % mpi.rank, recvbuf
[0] {'foo': True, 'spam': 'yes', 'bar':
3.14}
[1] {'foo': True, 'spam': 'yes', 'bar':
3.14}
[2] {'foo': True, 'spam': 'yes', 'bar':
3.14}
```

In the next example, a list of tuples is generated in the "middle" process and data is scattered to other processes. The syntax is slightly different than in previous examples (communicator indexing is used in order to specify the root processor):

```
>>> root = mpi.size/2
>>>
>>> sendbuf = None
>>> if mpi.rank = root:
...   sendbuf = [ (i,i**2,i**3) \
...               for i in [2,3,4] ]
>>> print '[%d] %s' % (mpi.rank, sendbuf)
[0] None
[1] [(2, 4, 8), (3, 9, 27), (4, 16, 64)]
[2] None
>>>
>>> recvbuf = mpi.WORLD[root].Scatter
(sendbuf)
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] (2, 4, 8)
[1] (3, 9, 27)
[2] (4, 16, 64)
```

Finally, some lists with integer an boolean values are generated in each processor. Next, they are gathered to last process in `COMM_WORLD`:

```
>>>sendbuf = [mpi.rank**2, mpi.rank%2!=0]
>>> print '[%d] %s' % (mpi.rank, sendbuf)
[0] [0, False]
[1] [1, True]
[2] [4, False]
>>>
>>> root = mpi.size-1
>>> recvbuf = mpi.WORLD[root].Gather
(sendbuf)
>>> print '[%d] %s' % (mpi.rank, recvbuf)
[0] None
[1] None
[2] [[0, False], [1, True], [4, False]]
```

References

[1] S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, H. Zhang, PETSc: Portable, extensible toolkit for scientific computation, 2001, http://www.mcs.anl.gov/petsc

[2] S. Balay, V. Eijkhout, W.D. Gropp, L.C. McInnes, B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, Basel, 1997, pp. 163–202.

[3] D.M. Beazley, SWIG: Simplified wrapper and interface generator, 1996–2005, http://www.swig.org/

[4] D.M. Beazley, P.S. Lomdahl, Feeding a large scale physics application to Python, in: Proceedings of Sixth International Python Conference, San Jose, California, 1997, pp. 21–29.
[5] Beowulf.org, The Beowulf cluster site, 2005, http://www.beowulf.org/
[6] G. Burns, R. Daoud, J. Vaigl, LAM: an open cluster environment for MPI, in: Proceedings of Supercomputing Symposium, 1994, pp. 379–386, http://www.lam-mpi.org/download/files/lam-papers.tar.gz
[7] A. Cardona, I. Klapka, M. Gerardin, Design of a new finite element programming environment, Eng. Comput. 11 (4) (1994) 365–381.
[8] P.F. Dubois, Pyfort: The Python-Fortran connection tool, 2000–2005, http://pyfortran.sourceforge.net/
[9] P. Greenfield, T. Miller, R.L. White, J.C. Hsu, Numarray: a new scientific array package for Python, 2003–2005, http://www.stsci.edu/resources/software_hardware/numarray
[10] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.
[11] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, 1994.
[12] K. Hinsen, The molecular modelling toolkit: a new approach to molecular simulations, J. Comput. Chem. 21 (2) (2000) 79–85.
[13] K. Hinsen, ScientificPython: Home page, 2005, http://starship.python.net/~hinsen/ScientificPython/
[14] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, 2001–2005, http://www.scipy.org/
[15] G. Karypis, ParMETIS: Parallel graph partitioning and sparse matrix ordering, 1996–2005, http://www-users.cs.umn.edu/~karypis/metis/parmetis/
[16] J. Kepner, S. Ahalt, MatlabMPI, J. Parallel Distrib. Comput. 64 (8) (2004) 997–1005.
[17] LAM Team, LAM/MPI parallel computing, 2005, http://www.lam-mpi.org/
[18] Message Passing Interface Forum, Message Passing Interface (MPI) Forum Home Page, 1994, http://www.mpi-forum.org/
[19] P. Miller, pyMPI: Putting the py in MPI, 2000–2005, http://pympi.sourceforge.net/
[20] MPI Forum, MPI: a message passing interface standard, Internat. J. Supercomput. Appl. 8(3/4) (1994) 159–416.
[21] MPI Forum, MPI2: a message passing interface standard, High Perform. Comput. Appl. 12(1–2) (1998) 1–299.
[22] MPICH Team, MPICH: a portable implementation of MPI, 2005, http://www-unix.mcs.anl.gov/mpi/mpich/
[23] O. Nielsen, Pypar Home page, 2002–2005, http://datamining.anu.edu.au/~ole/pypar/
[24] Open Engineering, OOFELIE toolkit, 2005, http://www.open-engineering.com/
[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI—The Complete Reference: vol. 1, The MPI Core, second ed., vol. 1, The MPI Core, MIT Press, Cambridge, MA, USA, 1998.
[26] V.E. Sonzogni, A.M. Yommi, N.M. Nigro, M.A. Storti, A parallel finite element program on a Beowulf cluster, Adv. Eng. Software 33 (7–10) (2002) 427–443.
[27] J.M. Squyres, J. Willcock, B.C. McCandless, P.W. Rijks, A. Lumsdaine, OOMPI Home page, 1996, http://www.osl.iu.edu/research/oompi/
[28] M.A. Storti, Geronimo cluster at CIMEC, 2001–2005, http://www.cimec.org.ar/geronimo
[29] M.A. Storti, N. Nigro, R. Paz, PETSc-FEM: A general purpose, parallel, multi-physics FEM program, 1999–2005, http://www.cimec.org.ar/petscfem
[30] S. Team, SPaSM: Parallel molecular dynamics code, 1994–2001, http://bifrost.lanl.gov/MD/MD.html
[31] G. van Rossum, Python programming language, 1990–2005, http://www.python.org/
[32] G. van Rossum, Python documentation, May 2004, http://docs.python.org/index.html
[33] G. van Rossum, Extending and embedding the Python interpreter, May 2004, http://docs.python.org/ext/ext.html