MPI for Python: Performance Improvements and MPI–2 Extensions

Lisandro Dalcín *, Rodrigo Paz, Mario Storti and Jorge D'Elía

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)
(S3000GLN) Santa Fe, Argentina. Phone/Fax: +54(0)342-4511594

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.

In its first release, MPI for Python was constructed on top of the MPI-1 specification defining an object oriented interface that closely followed the MPI-2 C++ bindings, and provided support for communications of general Python objects. In the latest release, this package is improved to enable direct blocking/nonblocking communication of numeric arrays, and to support almost all MPI-2 features.

Improvements in communication performance have been tested in a Beowulf class cluster. Results showed a negligible overhead in comparison to compiled C code.

MPI for Python is open source and available for download on the web (`http://www.cimec.org.ar/python`)

Key words: Message passing, MPI, High level languages, Parallel Python

1 Introduction

During the last decade, high performance computing has become an affordable resource to many more scientists and engineers than ever before. The conjunction of quality open source software and commodity hardware strongly influenced the now widespread popularity of dedicated Beowulf [1] class clusters and cluster of workstations. Message-passing has proven to be an effective computational model, specially suited for (but not limited to) distributed memory architectures. Although portable message-passing parallel programming used to be a nightmare in the past because of the many incompatible options developers were faced with, this situation definitely changed after the MPI Forum [2] released its standard specification, which rapidly gained widespread acceptance.

At the same time, the popularity of scientific computing environments such as MATLAB, and IDL has increased considerably. Users simply feel more productive in such interactive environments with tight integration of simulation and visualization. They are alleviated of low-level details associated to compilation/linking steps, memory management and input/output of traditional programming languages like Fortran, C, and C++. However, native support for parallel processing is absent and motivated different approaches to over-

∗ Corresponding author.

Email addresses: `dalcinl@intec.unl.edu.ar` (Lisandro Dalcín), `rodrigop@intec.unl.edu.ar` (Rodrigo Paz), `mstorti@intec.unl.edu.ar` (Mario Storti), `jdelia@intec.unl.edu.ar` (Jorge D'Elía).

come it [3,4].

Recently, the Python programming language has attracted the attention of many users and developers in the scientific community. Python offers a clean and simple syntax, is a very powerful language, and allows skilled users to build their own computing environment, tailored to their specific needs and based on their favorite high-performance Fortran, C, or C++ codes [5]. Sophisticated but easy to use and well integrated packages are available for interactive work [6,7], visualization [8,9], efficient multidimensional array processing [10], and scientific computing [11].

Following the aforementioned trends, some researchers have taken advantage of Python for writing the high-level parts of large-scale, massively parallel scientific applications and driving simulations in parallel architectures [12,13], while others have tried to make available the benefits of parallel computing to general Python codes using MPI [14,15].

In this work, the latest advances in the development of MPI for Python [16] are reported. MPI for Python is a package for the Python programming language enabling general applications to exploit multiple processors by using any available MPI implementation as a back-end.

The next section presents a brief overview of MPI, Python and MPI for Python. Section 3 describes the most relevant features added to MPI for Python. Section 4 presents some efficiency comparisons between MPI for Python and compiled C code communicating numeric arrays. Finally, section 5 presents some conclusions and plans for future work.

2 Background

2.1 What is MPI?

MPI [17,18], 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, and 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 and well known open source projects like MPICH [19,20] and Open MPI [21,22].

MPI follows an object oriented design defining a high-level abstraction for fast and portable interprocess communication[23,24]. Applications can run in clusters of (possibly heterogeneous) workstations or dedicated nodes, (symmetric) multiprocessors machines, or even a mixture of both. MPI hides all the low-level details, like networking or shared memory management, simplifying development and maintaining portability, without sacrificing performance.

2.2 What is Python?

Python [25] 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, makes 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++ and is also suitable as an extension language for customizable applications that require a programmable interface.

Python is an ideal candidate for writing the higher-level parts of large-scale scientific applications and driving simulations in parallel architectures. Python codes are quickly developed, easily maintained, and can achieve a high degree of integration with other libraries written in compiled languages.

2.3 What is MPI for Python?

MPI for Python [26] is a Python package providing bindings of the MPI standard, allowing any Python program to exploit multiple processors. This package is constructed on top of the MPI-1/MPI-2 specification and defines an object oriented interface that closely follows MPI-2 C++ bindings.

In its first release, MPI for Python provided support for blocking point-to-point and collective communications of general Python objects, as well as many facilities for managing process groups and defining new communication domains. Its API was designed with a focus on translating syntax and semantics of standard MPI-2 bindings from C++ to Python. Users with only a basic knowledge of standard C/C++ MPI bindings were able to use this package without having to learn a new interface.

3 New Features in MPI for Python

This section presents a survey of MPI capabilities and the new available features in MPI for Python to enhance communication performance and better support classic MPI-1 operations [17] in a Python programming environment. The recent availability of free, high quality, open-source MPI-2 implementations strongly motivated the inclusion of another set of features, in order to provide support full for almost all MPI-2 extensions [18].

3.1 Object Serialization

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

MPI for Python can communicate any general or built-in Python object taking advantage of the features provided by `cPickle` and `marshal` modules. Their functionalities are wrapped in two classes, `Pickle` and `Marshal`, defining `dump()` and `load()` methods carefully optimized for serialization of Python objects on memory streams. This approach is also fully extensible; that is, users are allowed to define new, custom serializers implementing the generic `dump()`/`load()` interface.

Any provided or user-defined serializer can be attached to communicator instances. They will be routinely used to build binary representations of objects to communicate (at sending processes), and restoring them back (at receiving processes).

3.2 Direct Communication of Memory Buffers

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. 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 was improved to support direct communication of any object exporting single-segment buffer interface. This interface is a standard Python mechanism provided by some type of objects (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 new 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.

3.3 Nonblocking and Persistent Communications

On many systems, performance can be significantly increased by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent, dedicated communication controller. Nonblocking communication is a mechanism provided by MPI in order to support such overlap.

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.

Often a communication with the same argument list is repeatedly executed within an inner loop. In such a case, 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. This kind of optimization can also alleviate the extra overheads associated to interpreted, dynamic languages like Python, especially for fine-grained tasks.

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.

The inherently asynchronous nature of nonblocking communications currently imposes some restrictions in what can be communicated using MPI for Python. Communication of memory buffers, as described in section 3.2 is fully supported. However, communication of general Python objects using serialization, as described in section 3.1, is possible but not transparent since objects must be explicitly serialized at sending processes, while receiving processes must first provide a memory buffer large enough to hold the incoming message and next recover the original object.

3.4 MPI-2 Extensions

3.4.1 Dynamic Process Management

An MPI-1 application is static; that is, no processes can be added to or deleted from an application after it has been started. 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 extension can be really useful, especially for serial 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 a mechanism to establish communication between two existing MPI applications, even when one did not “start” the other.

In MPI for Python, new processes 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), which 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 in a client/server approach. Server applications 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. 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 can release resources by calling the `Unpublish_name()` and `Close_port()` functions.

3.4.2 One-Sided Communications

One-sided communications (also called *Remote Memory Access*, *RMA*) supplements the traditional two-sided MPI communication model with a one-sided interface that can take advantage of the capabilities of RMA network hardware. This extension lowers latency and software overhead in applications written using a shared-memory-like paradigm. The semantics of one-sided communication are fairly complex. The MPI RMA API revolves around the use of objects called *windows*, which intuitively specify regions of a process's memory that have been made available for remote operations.

Windows are created by calling the `Create()` method of the `Win` class at all processes within a communicator and specifying a memory buffer (i.e., a base address and length). Three one-sided operations for remote write, read and reduction are available using the `Put()`, `Get()`, and `Accumulate()` methods respectively within a `Win` instance. These methods need an offset into the window and an integer rank identifying the remote target. This one-sided operations are implicitly nonblocking, and must be synchronized.

Windows are synchronized by using two primary modes. Active target synchronization requires the origin process to call the `Start()`/`Complete()` methods at the origin process, and target process cooperates by calling the `Post()`/`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()`/`Unlock()` methods.

3.4.3 Extended Collective Operations

In the MPI-1 specification, collective communications were only defined for intracommunicators. The MPI-2 specification introduces extensions generalizing many of the collective routines to intercommunicators. They can be really useful for collective interaction between disjoint group of processes created or connected as described in section 3.4.1.

MPI for Python was enhanced in order to support these extensions. The `Barrier()`, `Bcast()`, `Gather()`, `Scatter()`, `Allgather()`, `Alltoall()`, `Reduce()`, and `Allreduce()` methods are defined for both `Intracomm` and `Intercomm` classes. They are able to collectively communicate general Python objects, as discussed in section 3.1, or memory buffers, as discussed in section 3.2. Notably, scan and exclusive scan operations as defined in MPI do not apply to intercommunicators; that is, the `Scan()` and `Exscan()` methods are only available for `Intracomm` instances.

3.4.4 Parallel I/O

POSIX provides a model of a widely portable file system. However, the optimization needed for parallel I/O cannot be achieved with this interface, and can only be if the parallel I/O system provides 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 I/O, strided accesses, and control over physical file layout on storage devices.

The common patterns for accessing a shared file (broadcast, reduction, scatter, gather) is expressed using user-defined MPI datatypes. Compared to 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).

All these features are available in MPI for Python by using instances of the `File` class. Parallel files are created by calling method `Open()` at all processes within a communicator; they can be closed or even destroyed by calling `Close()` and `Delete()` methods respectively. The data layout in the file can be set and queried with the `Set_view()` and `Get_view()` methods respectively. Data access is provided by many methods related to read and write operations, but with different behavior regarding positioning, coordination, and synchronism.

4 Testing

Some efficiency tests were run on the Beowulf class cluster *Aquiles* [27] at CIMEC, Argentina. Its hardware consists of sixteen disk-less uniprocessor computing nodes with Intel Pentium 4 Prescott 3.0GHz 2MB cache processors, Intel Desktop Board D915PGN motherboards, Kingston Value RAM 2GB DDR 400MHz memory, and 3Com 2000ct Gigabit LAN network cards, interconnected with a 3Com SuperStack 3 Switch 3870 48-ports Gigabit Ethernet.

MPI for Python was compiled on a *Linux 2.6.17* box using *GCC 3.4.4* with *Python 2.4.4*. The chosen MPI implementation was *MPICH2 1.0.4p1*. Communications between processes involved numeric arrays, they were provided by *NumPy 1.0*.

The first test consisted in blocking send and receive operations (`MPI_SEND` and `MPI_RECV`) between a pair of nodes. Messages were numeric arrays of double precision (64 bits) floating-point values. The two supported communications mechanisms, serialization and memory buffers, were compared against compiled C code. A basic implementation of this test using MPI for Python with direct communication of memory buffers (translation to C or C++ is straightforward) is shown below.

```
from mpi4py import MPI
from numpy import empty, float64

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
array1 = empty(2**16, dtype=float64)
array2 = empty(2**16, dtype=float64)
sendbuf = [array1, 2**16, MPI.DOUBLE]
recvbuf = [array2, 2**16, MPI.DOUBLE]

wt = MPI.Wtime()
if rank == 0:
    comm.Send(sendbuf, 1, tag=0)
    comm.Recv(recvbuf, 1, tag=0)
elif rank == 1:
    comm.Recv(recvbuf, 0, tag=0)
    comm.Send(sendbuf, 0, tag=0)
wt = MPI.Wtime() - wt
```

Results are shown in figures 1 and 2. Throughput is computed as $2S/\Delta t$, where S is the basic message size (in megabytes), and Δt is the measured wall-clock time. Clearly, the overhead introduced by object serialization degrades overall efficiency; the maximum throughput in Python is about 60% of the one in C. However, the direct communication of memory buffers introduces a negligible overhead for medium-sized to long arrays.

The second test consisted in a small variation of the first one. The interchange of messages consisted in a bidirectional send/receive operation (`MPI_SENDRECV`). Results are shown in figures 3 and 4. In comparison to the previous test, the overhead introduced by object serialization is lower (the maximum throughput in Python is about 75% of the one in C) and the overhead communicating

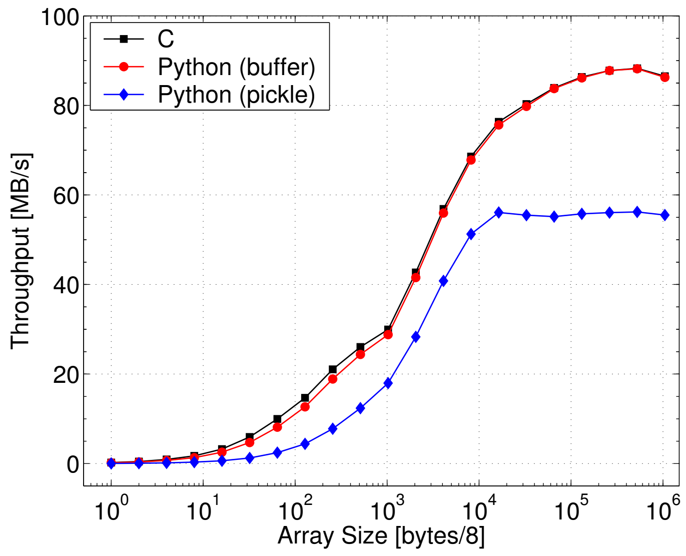

Figure 1. Throughput in blocking Send and Receive

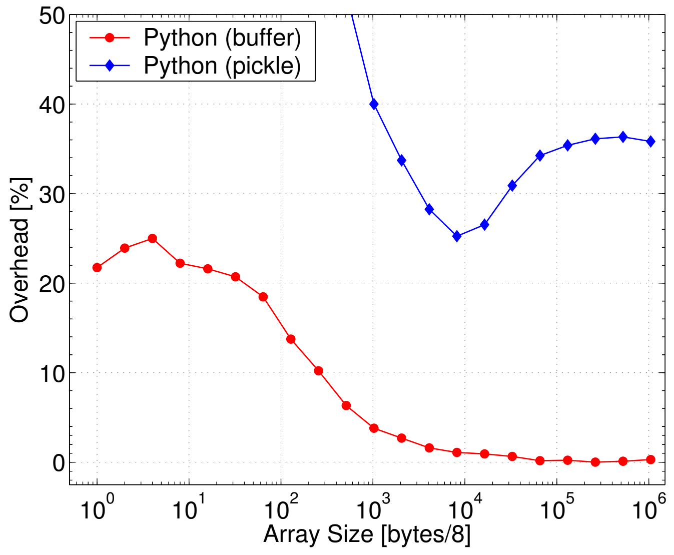

Figure 2. Relative overhead in blocking Send and Receive

memory buffers is similar (and again, it is negligible for medium-sized to long arrays).

The third test consisted in an all-to-all collective operation (`MPI_ALLTOALL`) on sixteen nodes. As in previous tests, messages were numeric arrays of double precision floating-point values. Results are shown in figures 5 and 6. Throughput is computed as $2(N-1)S/\Delta t$, where N is the number of nodes, S is the

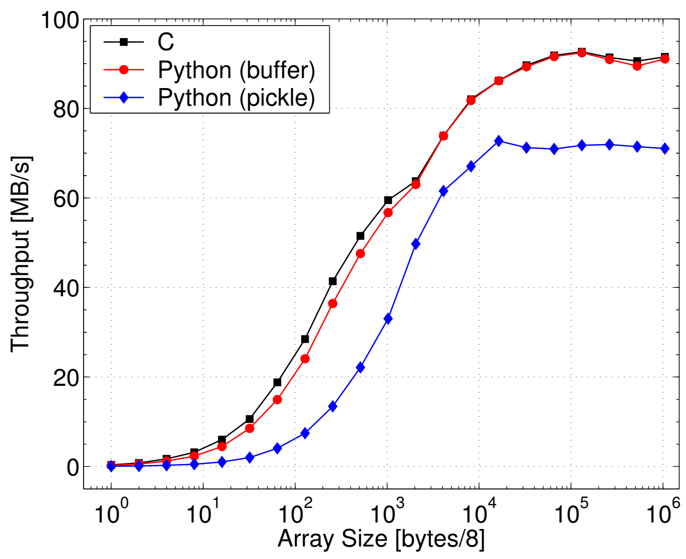

Figure 3. Throughput in bidirectional Send/Receive

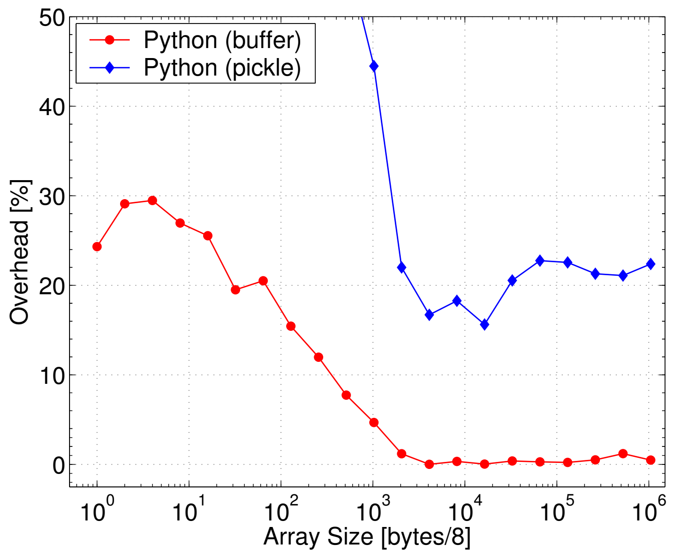

Figure 4. Relative overhead in bidirectional Send/Receive

basic message size (in megabytes), and Δt is the measured wall-clock time. The overhead introduced by object serialization is notably more significant than in previous tests; the maximum throughput in Python is about 40% of the one in C. However, the overhead communicating memory buffers is always below 1.5%.

Interested readers should review previous results from a similar set of tests [16],

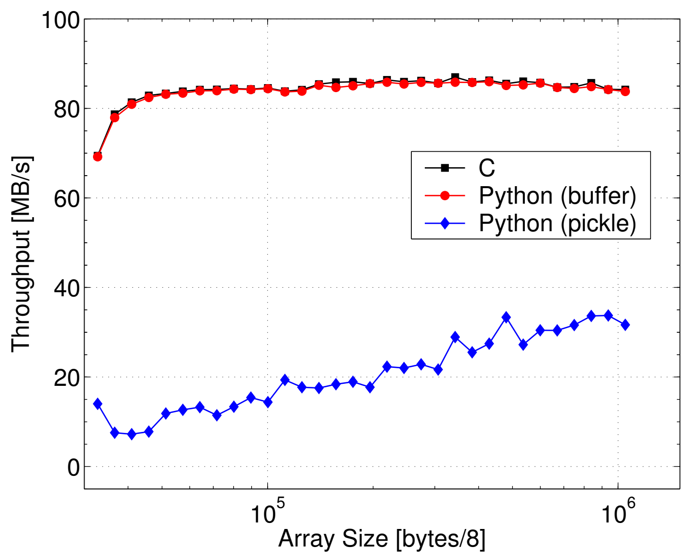

Figure 5. Timing in All-To-All

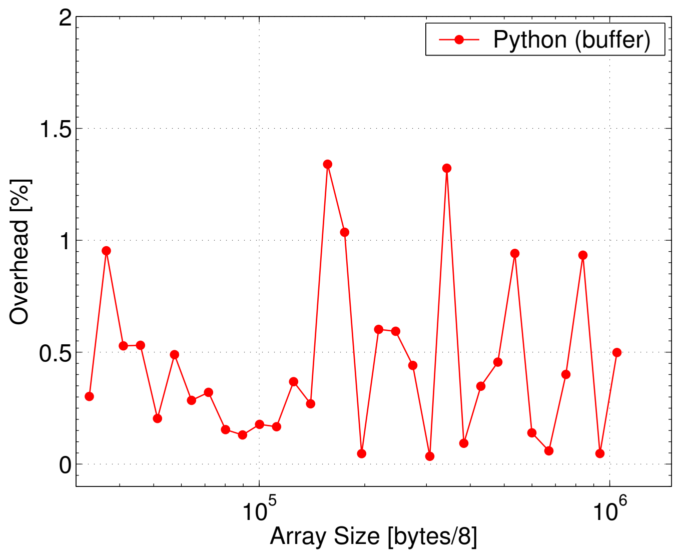

Figure 6. Timing in All-To-All

but obtained with older hardware components and software distributions.

5 Conclusions

MPI for Python provides a base layer for applying the message-passing paradigm in parallel applications written in Python. It makes use of any available MPI implementation retaining the syntax and semantics of the standard MPI specification. It can communicate general Python objects as well as any Python object exposing a memory buffer. In the later case, efficiency tests have shown that performance degradation is negligible, even for medium sized numeric array objects. In fact, the introduced overhead 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 MPI functionalities like datatype decoding, attribute catching and interoperability with Fortran libraries. Additionally, an automatic mapping between MPI datatypes and NumPy datatypes will be provided in order to simplify the parallelization of demanding applications involving multidimensional array processing.

Acknowledgements

This work has received financial support from *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina, grants PIP 02552/00, PIP 5271/05), *Universidad Nacional del Litoral* (UNL, Argentina, grant CAI+D 2005-10-64), and *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT, Argentina, grants PICT 12-14573/2003, PME 209/2003).

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

References

[1] Beowulf.org, The Beowulf cluster site, `http://www.beowulf.org/` (2006).

[2] Message Passing Interface Forum, Message Passing Interface (MPI) Forum Home Page, `http://www.mpi-forum.org/` (1994).

[3] J. Kepner, S. Ahalt, MatlabMPI, Journal of Parallel and Distributed Computing 64 (8) (2004) 997–1005.

[4] J. Kepner, A multi-threaded fast convolver for dynamically parallel image filtering, Journal of Parallel and Distributed Computing 63 (3) (2003) 360–372.

[5] H. P. Langtangen, Python Scripting for Computational Science, Simula Research Lab, Department of Informatics, University of Oslo, 2006.

[6] F. Pérez, B. Granger, IPython: a System for Interactive Scientific Computing, Comput. Sci. Eng. 9 (3), to appear. University of Colorado APPM Preprint #549.

[7] F. Pérez, IPython: an Enhanced Python Shell, `http://ipython.scipy.org/` (2001–2006).

[8] P. Barrett, J. Hunter, P. Greenfield, Matplotlib - A portable Python plotting package, in: Astronomical data analysis software & systems XIV., 2004.

[9] J. D. Hunter, Matplotlib, `http://matplotlib.sourceforge.net/` (2003–2006).

[10] T. Oliphant, NumPy: Numerical Python, `http://numpy.scipy.org/` (2005–2006).

[11] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, `http://www.scipy.org/` (2001–2006).

[12] D. M. Beazley, P. S. Lomdahl, Feeding a large scale physics application to Python, in: Proceedings of 6th. International Python Conference, San Jose, California, 1997, pp. 21–29.

[13] K. Hinsen, The Molecular Modelling Toolkit: A new approach to molecular simulations, Journal of Computational Chemistry 21 (2) (2000) 79–85.

[14] P. Miller, pyMPI: Putting the py in MPI, `http://pympi.sourceforge.net/` (2000–2006).

[15] O. Nielsen, Pypar Home page, `http://datamining.anu.edu.au/~ole/pypar/` (2002–2006).

[16] L. Dalcín, R. Paz, M. Storti, MPI for Python, Journal of Parallel and Distributed Computing 65 (9) (2005) 1108–1115.

[17] MPI Forum, MPI: A message passing interface standard, International Journal of Supercomputer Applications 8 (3/4) (1994) 159–416.

[18] MPI Forum, MPI2: A message passing interface standard, High Performance Computing Applications 12 (1–2) (1998) 1–299.

[19] MPICH Team, MPICH: A portable implementation of MPI, `http://www-unix.mcs.anl.gov/mpi/mpich/` (2006).

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

[21] Open MPI Team, Open MPI: Open source high performance computing, `http://www.open-mpi.org/` (2006).

[22] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. 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, 2004, pp. 97–104.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI - The Complete Reference: Volume 1, The MPI Core, 2nd Edition, Vol. 1, The MPI Core, MIT Press, Cambridge, MA, USA, 1998.

[24] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, MPI - The Complete Reference: Volume 2, The MPI-2 Extensions, 2nd Edition, Vol. 2, The MPI-2 Extensions, MIT Press, Cambridge, MA, USA, 1998.

[25] G. van Rossum, Python programming language, `http://www.python.org/` (1990–2006).

[26] L. Dalcín, MPI for Python, `http://www.mpi4py.scipy.org/` (2006).

[27] M. A. Storti, Aquiles cluster at CIMEC, `http://www.cimec.org.ar/aquiles` (2005-2006).