Advances in Water Resources 34 (2011) 1124–1139

Contents lists available at ScienceDirect

Advances in Water Resources

journal homepage: www.elsevier.com/locate/advwatres

Parallel distributed computing using Python

Lisandro D. Dalcin [*], Rodrigo R. Paz, Pablo A. Kler, Alejandro Cosimo

Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC), Instituto de Desarrollo Tecnológico para la Industria Química (INTEC), Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Universidad Nacional del Litoral (UNL), S3000GLN Santa Fe, Argentina

A R T I C L E I N F O

Article history:
Available online 22 April 2011

Keywords:
Python
MPI
PETSc

A B S T R A C T

This work presents two software components aimed to relieve the costs of accessing high-performance parallel computing resources within a Python programming environment: MPI for Python and PETSc for Python.

MPI for Python is a general-purpose Python package that provides bindings for the *Message Passing Interface* (MPI) standard using any back-end MPI implementation. Its facilities allow parallel Python programs to easily exploit multiple processors using the message passing paradigm. PETSc for Python provides access to the *Portable, Extensible Toolkit for Scientific Computation* (PETSc) libraries. Its facilities allow sequential and parallel Python applications to exploit state of the art algorithms and data structures readily available in PETSc for the solution of large-scale problems in science and engineering.

MPI for Python and PETSc for Python are fully integrated to PETSc-FEM, an MPI and PETSc based parallel, multiphysics, finite elements code developed at CIMEC laboratory. This software infrastructure supports research activities related to simulation of fluid flows with applications ranging from the design of microfluidic devices for biochemical analysis to modeling of large-scale stream/aquifer interactions.

1. Introduction

The popularity of high-level, general purpose scientific computing environments – such as MATLAB and IDL in the commercial side or Octave and Scilab in the open source side – has increased considerably during the last decade. Users simply feel much more productive in such interactive environments providing tight integration of simulation and visualization. They are alleviated of low-level details associated to compilation and linking steps, memory management and input/output of more traditional scientific programming languages like Fortran, C, and C++. The Python programming language is a distinguished member among these modern computing environments. Since it was born in the early 1990s, Python has steadily grown in popularity among the scientific community to arguably become *de facto* standard for computation-driven scientific research.

This paper reports on two open source tools that facilitate the access to high-performance parallel computing resources within a Python programming environment, MPI for Python [1] (known in short as *mpi4py*) and PETSc for Python [2] (known in short as *petsc4py*). They target hardware platforms ranging from desktop computers with multiple-processor and/or multiple-core architectures to clusters of workstations or dedicated computing nodes (with standard or special network interconnects), or even high-performance shared memory machines.

The rest of this section presents a brief description about the Python programming language and some of the more important tools available for scientific computing, as well as a some comments about MPI and PETSc. Sections 2 and 3 provide details about the design and capabilities of MPI for Python and PETSc for Python. Section 4 presents some performance tests aimed to measure the overhead introduced by the Python layer in comparison to pure C code. Finally, Section 5 presents two different application examples.

1.1. Python

Python [3,4] is a modern, powerful programming language. It has efficient high-level data structures, a simple but effective approach to object-oriented programming, it is easy to learn and highly extensible. It supports modules and packages, which encourages program modularity and code reuse.

Python's elegant syntax, together with its dynamic nature, makes an excellent language for scripting and rapid application development. Sophisticated but easy to use and well integrated solutions are available for interactive command-line work, efficient multi-dimensional array processing, linear algebra, 2D and 3D visualization, and other scientific computing tasks.

* Corresponding author. Tel./fax: +54 342 4511594.
E-mail addresses: dalcinl@gmail.com (L.D. Dalcin), rodrigo.r.paz@gmail.com (R.R. Paz), pabloakler@gmail.com (P.A. Kler), alecosimo@gmail.com (A. Cosimo).

0309-1708/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advwatres.2011.04.013

Python is easily extended with new functions and data structures implemented in other languages. This feature 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. Such capabilities prove to be an advantage for modern scientific computing: users have a high-level and productive environment at hand, yet they can reuse existing library code and optimize performance critical bottlenecks.

The Python programming language, augmented with a set open source packages that have been developed over the last decade by scientists and engineers, provides a "computational ecosystem" that is quite capable of supporting a wide range of applications – from casual scripting and lightweight tools to full-fledged systems. For a throughout discussion about the role of Python in scientific computing and additional information about selected Python packages, see [5–9].

1.1.1. NumPy

The *NumPy* project [10] started in the mid-90s as a collaborative effort of an international team of volunteers aimed to develop a data-structure for efficient array computation in Python. Since then, the *NumPy* package has found wide-spread adoption in academia and industry. Today, *NumPy* is one of the core packages for numerical computation in Python.

NumPy provides a powerful multi-dimensional array object with advanced and efficient general-purpose array operations. Additionally, *NumPy* contains three sub-libraries with numerical routines providing basic linear algebra operations, basic Fourier transforms and sophisticated capabilities for random number generation. It also provides facilities in order to support interoperability with C, C++, and Fortran.

Besides its obvious scientific applications, *NumPy* can also be used as an efficient multi-dimensional container of generic data. New structured data-types with fixed storage layout can be defined by combining fundamental data-types like integers and floats. This allows *NumPy* to seamlessly and speedily integrate with a wide variety of database formats.

1.1.2. F2PY

Although *NumPy* provides similar and higher-level capabilities, there are situations where selected, numerically intensive parts of Python applications still require the efficiency of a compiled code for processing huge amounts of data in deeply-nested loops. Fortran (especially Fortran 90 and above) is a language for efficiently implementing lengthy computations involving multi-dimensional arrays. State of the art implementations of many commonly used algorithms are readily available and implemented in Fortran.

F2PY [11] is a development tool that provides a connection between the Python and Fortran programming languages. It works by creating Python extension modules from special signature files or directly from annotated Fortran source files. These files with additional annotations included as comments, contain all the information (function names, arguments and their types, etc.) that is needed to construct convenient Python bindings to Fortran functions. *F2PY*-generated Python extension modules enable Python codes to call those Fortran 77/90/95 routines. In addition, *F2PY* provides the required support for transparently accessing Fortran 77 *common blocks* or Fortran 90/95 *module data*.

In a Python programming environment, *F2PY* is then the tool of choice for taking advantage of the speed-up of compiled Fortran code and integrating existing Fortran libraries.

1.1.3. Cython

Cython [12] is a recent development that provides access to low-level C data types and functionalities in a Python programming environment.

The *Cython* language is similar to Python, supporting most Python language constructs and libraries while adding syntax for declaring types, calling C functions, and manipulating C values. *Cython* code is compiled via C and the result runs within the Python runtime environment. When static type declarations are used in *Cython* source, it typically executes many times faster than Python and sometimes approaches the speed of C.

Using *Cython*, code which manipulates Python values and C values can be freely intermixed, with conversions occurring automatically wherever possible. Error checking of Python operations is also automatic, and the full power of Python exception handling facilities is available even in the midst of manipulating C data.

1.1.4. SWIG

SWIG [13], the *Simplified Wrapper and Interface Generator*, is an interface compiler that connects programs written in C and C++ with a variety of scripting languages.

Originally developed in 1995, *SWIG* was first used by scientists (in the Theoretical Physics Division at Los Alamos National Laboratory, USA) for building user interfaces to molecular dynamic simulation codes running on the *Connection Machine 5* supercomputer. In this environment, scientists needed to work with huge amounts of simulation data, complex hardware, and a constantly changing code base. The use of a Python scripting language interface provided a simple yet highly flexible foundation for solving these types of problems [14,15].

SWIG works by parsing the declarations found in C/C++ header files and using them to generate the wrapper code needed by scripting languages, in particular Python, to access the underlying C/C++ code. In addition, *SWIG* provides many customization features that let developers tailor the wrapping process to suit specific application needs.

Although *SWIG* was originally developed for scientific applications, it has since evolved into a general purpose tool that is used in a wide variety of applications – almost everything C/C++ and scripting programming is involved.

1.2. MPI

MPI, the *Message Passing Interface* [16,17], is a standardized, portable message-passing system designed to work on a wide variety of parallel computers. The standard defines a set 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++).

The paradigm of message-passing is especially suited for (but not limited to) distributed memory architectures and is used in today's most demanding scientific and engineering applications related to modeling, simulation, design, and signal processing.

MPI defines a high-level abstraction for fast and portable interprocess communication [18,19]. Applications can run in clusters of (possibly heterogeneous) workstations or dedicated compute 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.

Implementations are available from vendors of high-performance computers to well known open source projects like MPICH [20,21] and Open MPI [22,23].

1.3. PETSc

PETSc [24,25], the *Portable Extensible Toolkit for Scientific Computation*, is a suite of algorithms and data structures for the solution of problems arising on scientific and engineering applications, especially those modeled by partial differential equations, of large-scale nature, and targeted for high performance parallel computing environments [26].

PETSc is written in C (thus making it usable from C++); a Fortran interface (very similar to the C one) is also available. PETSc's complete functionality is only exercised by parallel applications, but serial applications are fully supported.

PETSc employs the MPI standard for inter-process communication, thus it is based on the message-passing model for parallel computing. Despite that, PETSc provides high-level interfaces with *collective* semantics so that typical users rarely have to make message-passing calls themselves.

PETSc is designed with an object-oriented style. Almost all user-visible types are abstract interfaces with implementations that may be chosen at runtime. Those objects are managed through handles to opaque data structures which are created, accessed and destroyed by calling appropriate library routines.

PETSc consists of a variety of components. Each component manipulates a particular family of objects and the operations one would like to perform on these objects. Some of the PETSc modules deal with:

- Index sets, including permutations, indexing into vectors, renumbering, etc.
- Vectors.
- Matrices (generally sparse).
- Distributed arrays for parallelizing regular grid-based problems.
- Krylov subspace methods.
- Preconditioners, including multigrid and sparse direct solvers.
- Nonlinear solvers.
- Timesteppers for solving time-dependent, nonlinear partial differential equations.

PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping. The libraries enable easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility.

Finally, PETSc is designed to be highly modular, enabling the interoperability with several specialized parallel libraries like *Hypre* [27], *Trilinos/ML* [28], *MUMPS* [29], and others through a unified interface.

2. MPI for Python

This section is devoted to describing MPI for Python, an open source software project that provides bindings of the MPI standard for the Python programming language.

MPI for Python is a general-purpose and full-featured package targeting the development of parallel applications in Python. It provides core facilities that allow parallel Python programs to exploit multiple processors. Sequential Python applications can also take advantages of MPI for Python by communicating through the MPI layer with external, independent parallel modules, possibly written in other languages like C, C++, or Fortran. MPI for Python employs a back-end MPI implementation, thus being usable on any parallel environment supporting MPI.

MPI for Python is implemented with *Cython*. The rationale for this choice is twofold. A high-level, object oriented interface with Python *look and feel* can be easily developed and maintained in terms of lower-level MPI types and calls handled in C. Additionally, MPI for Python can expose its internals to other C codes in such a way that MPI handles can be recovered from Python objects. Third-party tools aimed to bridge C and Python (e.g., *SWIG*) can take advantage of this and couple MPI for Python with other Python wrappers to MPI-based libraries.

MPI for Python implements the entire specifications from the MPI-2 standard revision 2.2[1] [30]. Naming conventions of the MPI-2 C++ bindings[2] are adopted, so users familiar with the C++ bindings can use MPI for Python without 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* can also work with raw memory buffers. The *pickle* module provides user-extensible facilities to serialize general Python objects using ASCII or binary formats. MPI for Python can communicate any built-in or user-defined Python object implementing the *pickle* protocol. These facilities are transparently 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*) imposes important overheads in memory as well as processor usage, especially when objects with large memory footprints are being communicated. Pickling general Python objects, ranging from primitive or container built-in types to user-defined classes, necessarily requires some processing for dispatching the appropriate serialization method (that depends on the type of the object) and processor usage to perform the actual packing. Additional memory is always needed, and if its total amount is not known *a priori*, many memory 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 also supports direct communication of any object implementing the Python buffer interface. This interface is a standard Python mechanism provided by some types (e.g., byte 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 space or time overhead compared to compiled C, C++ or Fortran codes.

2.2. Using MPI for Python

Here, a general overview of MPI concepts and functionalities readily available in MPI for Python are presented. Discussed features range from classical MPI-1 message-passing communication operations to more advanced MPI-2 operations like dynamic process management, one-sided communication, and parallel input/output.

2.2.1. Communicators

In MPI for Python, `Comm` is the base class of communicators. Communicator size and calling process rank can be respectively obtained with methods `Get_size()` and `Get_rank()`.

The `Intracomm` and `Intercomm` classes are derived from 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.

[1] At the time of this writing, MPI for Python does not support `MPI_Alltoallw` (see MPI standard document [30], Section 5.8, pp. 160). Support for this functionality is under development.

[2] The C++ bindings for MPI were deprecated in the revision 2.2 of the MPI standard; they are scheduled for removal in the upcoming MPI-3 standard.

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

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 Union(), Intersect() and Difference() are fully supported, as well as the creation of new communicators from these groups.

2.2.2. Blocking point-to-point communications

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 either general Python objects or raw memory buffers. The buffered, ready, and synchronous (Bsend(), Rsend() and Ssend()) modes are also supported.

2.2.3. Nonblocking point-to-point 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. 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. 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.

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 dynamic languages like 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 is started with the Start() method.

2.2.4. Collective communications

The Bcast(), Scatter(), Gather(), Allgather() and Alltoall() methods of Intracomm instances provide support for collective communications. Those methods can communicate either general Python objects or raw memory buffers. The "vector" variants (which can communicate varying amount of data at each process) Scatterv(), Gatherv(), Allgatherv() and Alltoallv() are also supported, they can only communicate objects exposing raw memory buffers. All these collective operations are supported on both intracommunicators and intercommunicators.

Global reduction operations are accessible through the Reduce(), Allreduce(), Scan() and Exscan() methods. All the predefined (i.e., SUM, PROD, MAX, etc.) and user-defined reduction operations can be applied to general Python objects (however, the actual required computations are performed sequentially at some process). User-defined reduction operations on memory buffers are also supported. Reductions are supported on both intracommunicators and intercommunicators (except inclusive and exclusive scan operations, MPI-2 defines them only for intracommunicators).

2.2.5. Dynamic process management

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() 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 application 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.

2.2.6. One-sided operations

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 every process in a communicator and specifying a memory buffer (i.e., a base address and length). 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 to the base address of the remote memory block being accessed.

2.2.7. Parallel input/output operations

In MPI for Python, all MPI input/output operations are performed through instances of the File class. File handles are obtained by calling method Open() at every process in 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 supports around 30 different 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.

2.3. Related projects

pyMPI [31] is a pioneering project bringing general-purpose MPI support to Python. It is implemented in C with modified Python interpreter (as opposed to employ the stock Python interpreter readily available at the computing platform) and a companion MPI module exposing core MPI functionalities and other facilities that are beyond the MPI standard. It permits basic interactive parallel runs, which are useful for learning and debugging, and provides an interface suitable for basic parallel programing. General Python objects supporting *pickle* protocol as well as *NumPy* arrays can be communicated. However, there is partial support for MPI-1 features like nonblocking communication, communication domains, and process topologies. Support for user-defined MPI datatypes is absent. Advanced MPI-2 features like dynamic process management, one-sided communication, and parallel input/output are not available.

Pypar [32] is a minimalistic and intuitive Python interface to MPI. It is a lightweight wrapper implemented with a mixture of high-level Python code and low-level extension module written in C, the Python interpreter does not require modification. General Python objects can be communicated using the *pickle* protocol. There is good support for communicating *NumPy* arrays and practically full MPI bandwidth can be achieved. However, there is no support for basic MPI-1 features of common use like user-defined MPI datatypes, nonblocking communication, communication domains, and process topologies. Advanced MPI-2 features like dynamic process management, one-sided communication, and parallel input/output are not available.

pyMPI and *Pypar* provide many of the features available in MPI for Python. However, differences in design and additional capabilities distinguish MPI for Python from these packages. These differences are summarized below.

- MPI for Python does not require a modified Python interpreter. The stock Python interpreter readily available at the computing platform is employed.
- MPI for Python is implemented with *Cython* (as opposed to low-level C code), facilitating development and maintenance.
- MPI for Python features support for well-known wrappers generator tools like *SWIG* and *F2PY*, lowering the barrier to reuse existing C, C++, and Fortran MPI-based libraries.
- MPI for Python Application Program Interface (API) follows closely the MPI standard specification.
- MPI for Python can communicate general Python object and *NumPy* arrays. However, fast communication of array data is not limited to *NumPy* – any Python type implementing the Python buffer interface can participate.
- MPI for Python supports the complete MPI-1 specification. All blocking/nonblocking point-to-point and collective operations are available, as well as user-defined MPI datatypes, multiple communication domains and Cartesian/graph process topologies.
- MPI for Python supports for the complete MPI-2 specification, providing full coverage of advanced features like dynamic process management, one-sided communications, and parallel input/output.

3. PETSc for Python

This section describes PETSc for Python, an open-source software project that provides bindings to PETSc libraries for the Python programming language.

PETSc for Python is a general-purpose and full-featured package. Its facilities allow sequential and parallel Python applications to exploit state of the art algorithms and data structures readily implemented in PETSc and targeted to large-scale numerical simulations arising in many problems of science and engineering.

PETSc for Python is implemented with *Cython*. The rationale for this choice is the same as the already commented in Section 2. PETSc presents a rather large API accessible from C and Fortran. Although PETSc is designed with an object-oriented style, its API is limited to the procedural style of the C and Fortran – these programming languages do not support natively some more advanced concepts such as classes, inheritance and polymorphism, or exception-based error handling. PETSc for Python was designed from the ground to present to users an easy-to-use, high-level, *pythonic* interface. This interface is certainly easier and more pleasant to use than the native ones available in PETSc for C and Fortran.

PETSc for Python has coverage for the most important PETSc features. Among them, we can mention assembling distributed vector and sparse matrices in parallel, solving systems of linear equations with Krylov-based iterative methods and direct methods, and solving systems of nonlinear equations with Newton-based iterative methods including matrix-free techniques. It is not feasible to provide here an detailed listing of all the supported features, the complete API reference is available at the project documentation.

The rest of this section presents a general overview and some examples of the many PETSc concepts and functionalities readily available in PETSc for Python. The examples are simple, self-contained, and implemented in a few lines of Python code. Nevertheless, they show general usage patterns of PETSc for Python for implementing linear algebra algorithms, assembling sparse matrices, and solving systems of linear and nonlinear equations within a Python programming environment.

3.1. Vectors

PETSc for Python provides access to PETSc vectors, index sets and general vector scatter/gather operations through the `Vec`, `IS`, and `Scatter` classes respectively. By using them, the management of distributed field data is highly simplified in parallel applications.

Besides the use as containers for field data, PETSc vectors also represent algebraic entities of finite-dimensional vector spaces. For this case, the `Vec` class provides many methods for performing common linear algebra operations, like computing vector updates (`axpy()`, `aypx()`, `scale()`), inner products (`dot()`) and different kinds of norms (`norm()`).

Fig. 1 shows a basic implementation of a simple Krylov-based iterative linear solver, the (unpreconditioned) conjugate gradient method.

3.2. Matrices

PETSc for Python provides access to PETSc matrices through the `Mat` class. New `Mat` instances are obtained by calling the `create()` method. Next, the user specifies the row and column sizes by calling the `setSizes()` method. Finally, a call to the `setType()` method selects a particular matrix implementation.

Matrix entries can be set (or added to existing entries) by calling the `setValues()` method. PETSc simplifies the assembling of parallel matrices. Any process can contribute to any entry. However, off-process entries are internally cached. Because of this, a final call to the `assemblyBegin()` and `assemblyEnd()` methods is required in order to communicate off-process entries to the actual owning process. Additionally, those calls prepare some internal data structures for performing efficient parallel operations like matrix–vector product. The latter operation is available by calling the `mult()` method.

Fig. 2 shows the basic steps for creating and assembling a sparse matrix in parallel. The assembled matrix is a discrete representation

```
def cg(A, b, x, imax=50, eps=1e-6):
    """
    A, b, x  : matrix, rhs, solution
    imax     : maximum iterations
    eps      : relative tolerance
    """
    # allocate work vectors
    r = b.duplicate()
    d = b.duplicate()
    q = b.duplicate()
    # initialization
    i = 0
    A.mult(x, r)
    r.aypx(-1, b)
    r.copy(d)
    delta_0 = r.dot(r)
    delta = delta_0
    # enter iteration loop
    while (i < imax and
           delta > delta_0 * eps**2):
        A.mult(d, q)
        alpha = delta / d.dot(q)
        x.axpy(+alpha, d)
        r.axpy(-alpha, q)
        delta_old = delta
        delta = r.dot(r)
        beta = delta / delta_old
        d.aypx(beta, r)
        i = i + 1
    return i, delta**0.5
```

Fig. 1. Basic implementation of conjugate gradients method.

```
from petsc4py import PETSc

# grid size and spacing
m, n  = 32, 32
hx = 1.0/(m-1)
hy = 1.0/(n-1)

# create sparse matrix
A = PETSc.Mat()
A.create(PETSc.COMM_WORLD)
A.setSizes([m*n, m*n])
A.setType('aij') # sparse

# precompute values for setting
# diagonal and non-diagonal entries
diagv = 2.0/hx**2 + 2.0/hy**2
offdx = -1.0/hx**2
offdy = -1.0/hy**2

# loop over owned block of rows on this
# processor and insert entry values
Istart, Iend = A.getOwnershipRange()
for I in range(Istart, Iend) :
    A[I,I] = diagv
    i = I//n    # map row number to
    j = I - i*n # grid coordinates
    if i> 0  : J = I-n; A[I,J] = offdx
    if i< m-1: J = I+n; A[I,J] = offdx
    if j> 0  : J = I-1; A[I,J] = offdy
    if j< n-1: J = I+1; A[I,J] = offdy

# communicate off-processor values
# and setup internal data structures
# for performing parallel operations
A.assemblyBegin()
A.assemblyEnd()
```

Fig. 2. Assembling a sparse matrix in parallel.

of the two-dimensional Laplace operator on the unit square equipped with homogeneous boundary conditions after a 5-points finite differences discretization. The grid supporting the discretization scheme is structured and regularly spaced. Furthermore, the grid nodes have a simple contiguous block-distribution by rows on a group of processes.

3.3. Linear solvers

PETSc for Python provides access to PETSc linear solvers and preconditioners through the `KSP` and `PC` classes.

New `KSP` instances are obtained by calling the `create()` method. This call automatically creates a companion inner preconditioner (i.e., a `PC` instance) that can be retrieved with the `getPC()` method for further manipulations. The `KSP` and `PC` classes provide the `setType()` methods for the selection of a specific iterative method and preconditioning strategy. The `setTolerances()` method enables the specification of the different tolerances for declaring convergence; other algorithmic parameters can also be set. Additionally, PETSc for Python supports attaching a user-defined Python function for monitoring the iterative process (by calling the `setMonitor()` method) and defining a custom convergence criteria (by calling the `setConvergenceTest()` method).

`KSP` objects have to be associated with a matrix (i.e., a `Mat` instance) representing the operator of the linear problem and a (possibly different) matrix for defining the preconditioner. This is done by calling the `setOperators()` method. Additional options set from command line arguments, configuration files, and environment variables can be specified by calling the `setFromOptions()` method. In order to actually solve a system of linear equations, the `solve()` method have to be called with appropriate vector arguments (i.e., `Vec` instances) specifying the right hand side and the location where to build the solution.

Fig. 3 presents an example showing the basic steps required for creating and configuring a linear solver and its inner preconditioner in PETSc for Python. This linear solver and preconditioner

```
# create linear solver,
ksp = PETSc.KSP()
ksp.create(PETSc.COMM_WORLD)
# use conjugate gradients
ksp.setType('cg')
# and incomplete Cholesky
ksp.getPC().setType('icc')
# obtain sol & rhs vectors
x, b = A.getVecs()
x.set(0)
b.set(1)
# and next solve
ksp.setOperators(A)
ksp.setFromOptions()
ksp.solve(b, x)
```

Fig. 3. Solving a linear problem.

combination is employed for solving a linear system involving a previously assembled parallel sparse matrix (see Fig. 2).

3.4. Nonlinear solvers

PETSc for Python provides access to PETSc nonlinear solvers through the `SNES` class.

`SNES` objects have to be associated with a user-defined Python function in charge of evaluating the nonlinear residual vector and, optionally, a function for the Jacobian matrix evaluation, at each nonlinear iteration step. Those user routines can be set with the methods `setFunction()` and `setJacobian()`.

New `SNES` instances are obtained by calling the `create()` method. This call automatically creates a companion inner linear solver (i.e., a `KSP` instance) that can be retrieved with the `getKSP()` method for further manipulations. The `setTolerances()` method enables the specification of the different tolerances for declaring convergence; other algorithmic parameters can also be set. Additionally, PETSc for Python supports attaching user-defined Python functions for monitoring the iterative process (by calling the `setMonitor()` method) and defining a custom convergence criteria (by calling the `setConvergenceTest()` method).

In order to actually solve a system of nonlinear equations, the `solve()` method has to be called with appropriate vector arguments (i.e., `Vec` instances) specifying an optional right hand side (usually not provided as it is the zero vector) and the location where to build the solution (which additionally can specify an initial guess for starting the nonlinear loop).

Consider the following boundary value problem in two dimensions:

$$-\Delta U(x) = \alpha \exp[U(x)], \quad x \in \Omega,$$
$$U(x) = 0, \quad x \in \partial\Omega;$$

where Ω is the unit square $(0,1)^2$ and $\partial\Omega$ is the boundary, Δ is the two-dimensional Laplace operator, U is a scalar field defined on Ω, and α is a constant. The equation is nonlinear and usually called the Bratu problem. The nonlinear system has a bifurcation (turning point) at $\alpha_{\max} \approx 6.80812$, there is no solution for $\alpha > \alpha_{\max}$. The standard 5-point finite differences stencil is employed for performing a spatial discretization on a structured, regularly spaced grid. As result of the discretization process, a system of nonlinear equation is obtained. Fig. 4 shows the Python implementation of the nonlinear residual function for the Bratu problem and the basic steps required for creating and configuring a nonlinear solver. The inner Krylov linear solver is configured to use conjugate gradient method. Additionally, the nonlinear solver is configured to use a matrix-free method (i.e., the Jacobian is not explicitly computed).

```
from petsc4py import PETSc
from numpy import exp

m, n = 32, 32 # grid sizes
alpha = 6.8  # parameter

# Nonlinear function
def Bratu2D(snes, X, F, alpha, m, n):
    # NumPy array <- Vec
    x = X[...].reshape(m, n)
    f = F[...].reshape(m, n)
    # setup 5-points stencil
    u  = x[1:-1, 1:-1] # center
    uN = x[1:-1,  :-2] # north
    uS = x[1:-1, 2:  ] # south
    uW = x[ :-2, 1:-1] # west
    uE = x[2:,   1:-1] # east
    # compute nonlinear function
    hx = 1.0/(m-1) # x grid spacing
    hy = 1.0/(n-1) # y grid spacing
    f[:,:] = x
    f[1:-1, 1:-1] = \
         (2*u - uE - uW) * (hy/hx) \
       + (2*u - uN - uS) * (hx/hy) \
       - alpha * exp(u)  * (hx*hy)

# create  nonlinear solver
snes = PETSc.SNES().create()
# register the function in charge of
# computing the nonlinear residual
f = PETSc.Vec().createSeq(m*n)
snes.setFunction(Bratu2D, f,
                 args=(alpha, m, n))
# configure the nonlinear solver
# to use a matrix-free Jacobian
snes.setUseMF(True)
snes.getKSP().setType('cg')
snes.setFromOptions()

# solve the nonlinear problem
b, x = None, f.duplicate()
x.set(0) # zero inital guess
snes.solve(b, x)
```

Fig. 4. Solving a nonlinear problem.

4. Performance tests

This section presents some performance test aimed to measure the overhead introduced by the Python layer in comparison to pure C codes. First, wall-clock time[3] of some selected point-to-point and collective MPI communication calls using MPI for Python are compared to the ones obtained with an equivalent pure C implementation; both Ethernet and shared memory communication channels were exercised. Next, PETSc for Python and an equivalent pure C code are employed for driving the solution of a model transient, nonlinear, partial differential equation problem using matrix-free methods; the heavy computations at grid-level loops are implemented in Fortran 90.

The hardware employed to perform these tests was

- a small research cluster consisting of a server and 6 compute nodes with two quad-core Intel Xeon E5420 2.5 GHz processors, 1333 MHz front-side bus, 8 GB DDR2–667 memory (10.6 GB/s per-socket theoretical peak memory bandwidth), interconnected via a switched Gigabit Ethernet network;
- a high-end desktop with single quad-core Intel i7 950 3.07 GHz processor, QuickPath interconnect, 12 GB DDR3-1066 memory (25.6 GB/s per-socket theoretical peak memory bandwidth)

and the software stack consisted of

- Linux 2.6.32 and GCC 4.4.4;
- MPICH2 1.2.1p1 and PETSc 3.1;
- Python 2.6.2 and NumPy 1.3.0.

4.1. MPI for Python

[3] *Wall-clock time* or *wall time* is the total amount of time (as determined by a chronometer) that a task takes to complete. It includes the time required for computations, input/output and communications. *Wall-clock time* should not be confused with *CPU time* or *processor time*, which measures only the time a processor was assigned to actively work on a certain task.

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 (via *pickle*) and direct communication of 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. The actual implementation took into account memory preallocation (in order to avoid paging effects) and parallel synchronization (in order to avoid asynchronous skew in the start-up phase).

```
   from mpi4py import MPI
   import numpy as np
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   msglen = 2**16
   array1 = np.empty(msglen,dtype = 'd')
array2 = np.empty(msglen,dtype = 'd')
sendbuf = [array1,msglen,MPI.DOUBLE]
   recvbuf = [array2,msglen,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
```

For increasing message sizes, the wall clock time required for communication is measured many times and then averaged. Throughput is computed as the ratio of message size (in bytes) and wall clock time (in seconds) required to accomplish the communication. Python overhead is computed from wall-clock times as $(T_{\text{Python}} - T_{\text{C}})/T_{\text{C}}$.

Results obtained on the switched Gigabit Ethernet network are shown in Fig. 5. As expected, the overhead introduced by object serialization degrades overall efficiency. Comparing to communication performed in C, the overhead of *pickle* communication is around 80% for small messages and around 30% for large messages. However, fast communication of array data is quite efficient. The overhead of *buffer* communication is below 10% for small messages and below 5% for large messages.

Results obtained on shared memory are shown in Fig. 6. In this case, the overhead introduced by the Python layer is quite more noticeable. Comparing to communication performed in C, *pickle* communication overhead is 150$\times$ for small messages and around 5$\times$ for large messages, while the overhead of *buffer* communication is around 2.5$\times$ for small messages and around 0.5$\times$ for large messages.

The second test consisted of wall-clock time measurements of *Broadcast* and *All-to-All* collective operations on four processes. Messages were again numeric arrays of double precision floating-point values.

Results obtained on the switched Gigabit Ethernet network are shown in Fig. 7. For small messages, the overhead of *pickle* communication is significant – 12$\times$ for *Broadcast* and 3.5$\times$ for *All-to-All* – for large messages, it is less noticeable. The overhead of *buffer* communication is small, particularly for *All-to-All* – less than 10% for all message sizes. Results obtained on shared memory are shown in Fig. 8, they follow the trends of previous results. However, the overhead of *pickle* communication is quite more noticeable for all message sizes.

Finally, it is worth to remark that the all the previous tests involved communication of contiguous *NumPy* arrays. For these objects, the *pickle* protocol is implemented quite efficiently – the total amount of memory required for serialization is known in advance and the array items have a common data type corresponding to a C primitive type. For more general, user-defined Python objects containing deeply nested data structures, *pickle* communication is expected to achieve lower performance than the reported here.

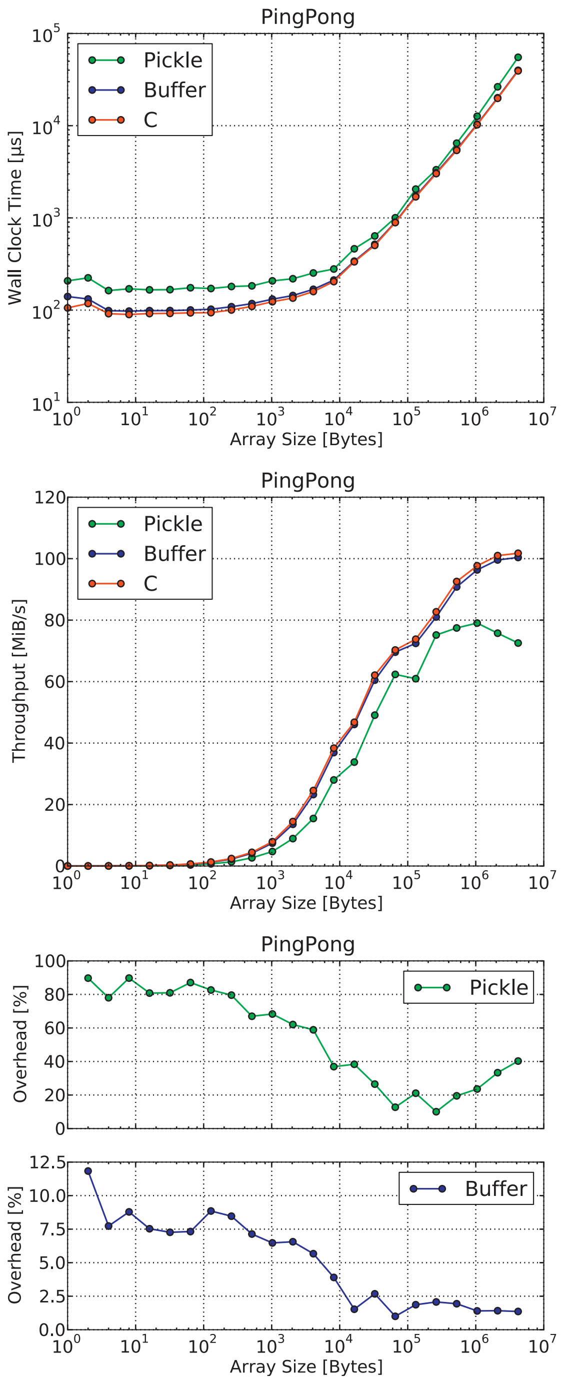


Fig. 5. PingPong – Gigabit Ethernet.

4.2. PETSc for Python

Consider the following diffusive, unsteady, non-linear, scalar problem in the unit cube $\Omega = (x_1, x_2, x_3) = (0,1)^3$

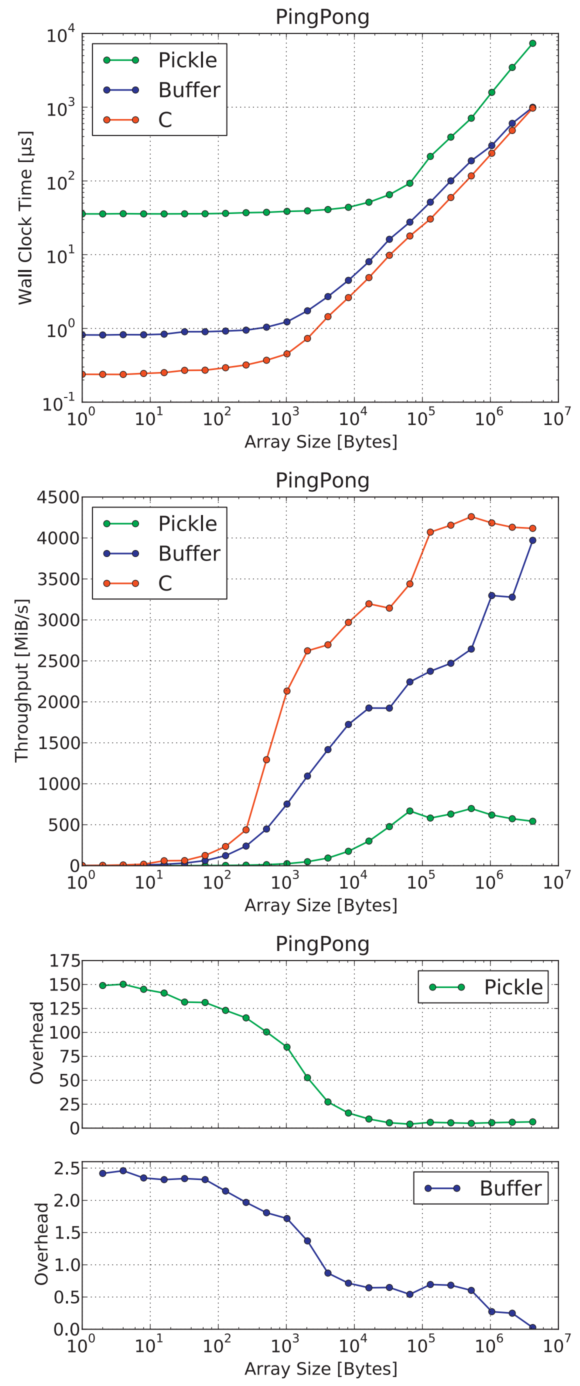


Fig. 6. PingPong – shared memory.

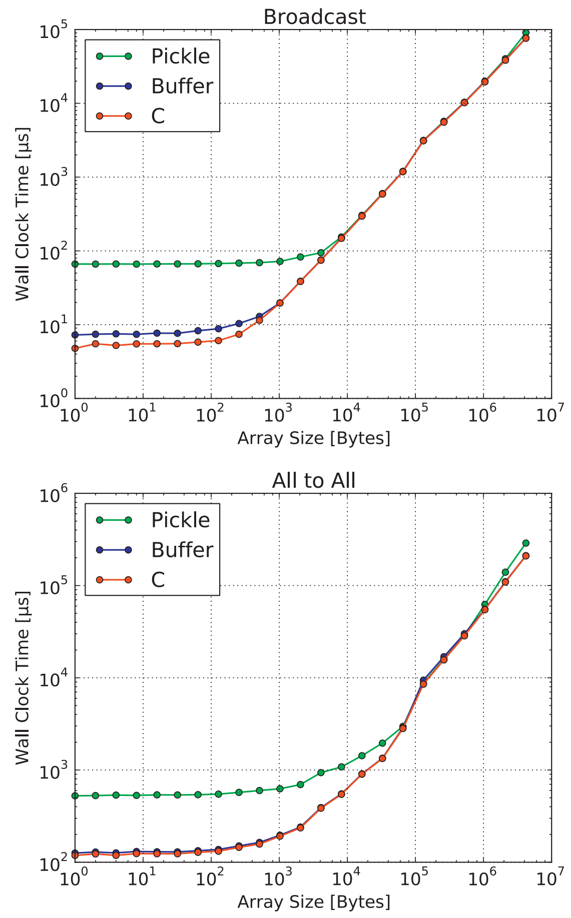


Fig. 7. Broadcast and All-to-All – Gigabit Ethernet.

$$\frac{\partial \phi}{\partial t} - \nabla \cdot (\kappa(\phi)\nabla\phi) = G, \quad \text{on } \Omega \times (0,T] \tag{1}$$

with homogeneous Neumann conditions at the boundary $\Gamma = \partial\Omega$ and given initial conditions,

$$\frac{\partial \phi}{\partial \mathbf{n}} = 0 \quad \text{at } \Gamma \times [0,T], \qquad \phi = \phi^0 \quad \text{at } t = 0, \tag{2}$$

where $\mathbf{n}$ is the outer normal to the boundary.

The diffusion coefficient κ depends on ϕ in the following way:

$$\kappa(\phi) = \begin{cases} 1 & \text{if } \phi \geqslant 0, \\ \frac{1}{1+\phi^2} & \text{if } \phi < 0, \end{cases} \tag{3}$$

and the source term G is the line source

$$G\left(x_1 = \frac{1}{4}, x_2 = \frac{1}{4}, \frac{1}{2} \leqslant x_3 \leqslant 1\right) = -300. \tag{4}$$

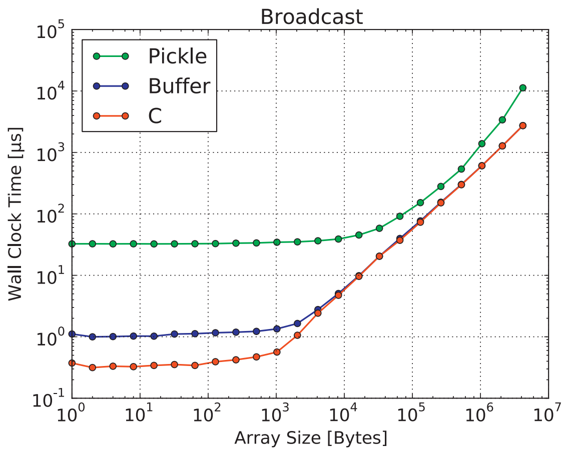
After time discretization using backward-Euler scheme with time step Δt and space discretization with centered finite differences on a structured grid of $N_1 \times N_2 \times N_3$ points, the following system of equations is obtained

$$\frac{1}{\Delta t}\left(\phi^{n+1}_{i,j,k} - \phi^{n}_{i,j,k}\right) - L_{i,j,k}(\kappa, \phi^{n+1}) - G_{i,j,k} = 0, \quad i,j,k = 1,\ldots,N_1,N_2,N_3, \tag{5}$$

where $\phi^{n}_{i,j,k}$ and $\phi^{n+1}_{i,j,k}$ are ϕ at point $((i-1)\Delta x_1, (j-1)\Delta x_2, (k-1)\Delta x_3)$ and time levels t^n and $t^{n+1} = t^n + \Delta t$, respectively, and

$$\begin{aligned} L_{i,j,k}(\kappa,\phi) = {} & \frac{\kappa\left[i-\frac{1}{2}\right]}{(\Delta x_1)^2}\phi[i-1] - \frac{\kappa\left[i-\frac{1}{2}\right] + \kappa\left[i+\frac{1}{2}\right]}{(\Delta x_1)^2}\phi[0] \\ & + \frac{\kappa\left[i+\frac{1}{2}\right]}{(\Delta x_1)^2}\phi[i+1] + \frac{\kappa\left[j-\frac{1}{2}\right]}{(\Delta x_2)^2}\phi[j-1] \\ & - \frac{\kappa\left[j-\frac{1}{2}\right] + \kappa\left[j+\frac{1}{2}\right]}{(\Delta x_2)^2}\phi[0] + \frac{\kappa\left[j+\frac{1}{2}\right]}{(\Delta x_2)^2}\phi[j+1] \\ & + \frac{\kappa\left[k-\frac{1}{2}\right]}{(\Delta x_3)^2}\phi[k-1] - \frac{\kappa\left[k-\frac{1}{2}\right] + \kappa\left[k+\frac{1}{2}\right]}{(\Delta x_3)^2}\phi[0] \\ & + \frac{\kappa\left[k+\frac{1}{2}\right]}{(\Delta x_3)^2}\phi[k+1], \end{aligned} \tag{6}$$

where, in short hand notation, $[0] = (i,j,k)$, $\left[i \pm \frac{1}{2}\right] = \left(i \pm \frac{1}{2}, j, k\right)$ and so on, and ϕ at staggered points is obtained with the averaging scheme $\phi\left[i \pm \frac{1}{2}\right] = \frac{1}{2}(\phi[i \pm 1] + \phi[0])$ and so on.



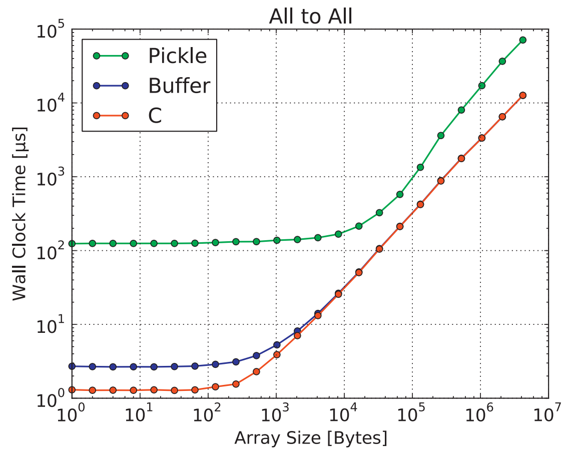


Fig. 8. Broadcast and All-to-All – shared memory.

The differences scheme detailed above is implemented in a Fortran 90 subroutine. This subroutine loops over i, j, k grid nodes computing nonlinear residuals according to Eq. (5) and taking into account boundary conditions (Eq. (2)).

The Fortran 90 subroutine in charge of computing residuals is employed in both Python and C driver codes employing PETSc data structures and algorithms. *F2PY* makes possible the access of the Fortran code from Python, while accessing the Fortran code from C is just a matter of accounting for the name mangling conventions of the Fortran compiler.

A `TS` timestepper is configured to run 10 time steps with $t^0 = 0$, $\Delta t = 0.01$ and initial conditions $\phi^0_{i,j,k} = 0$. At each time step, the `SNES` nonlinear solver iterates until the nonlinear residual norm is reduced to 10^{-6} of the initial one. At each nonlinear iteration, the inner `KSP` linear solver performs conjugate gradients iterations with no preconditioning until the linear residual norm is reduced by 10^{-6} respect to the initial one.

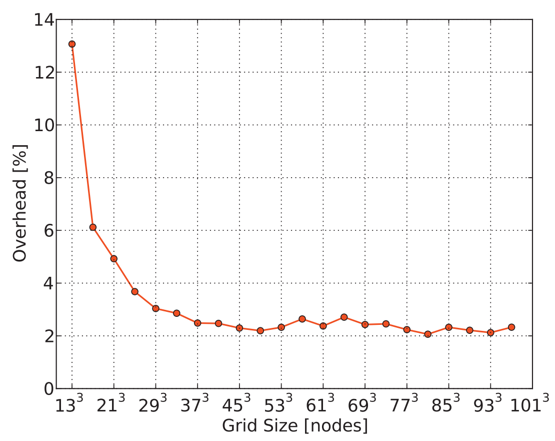


Fig. 9. Overhead of Python versus C codes for the solution of Eqs. (1) and (2).

Matrix entries of the Jacobian of Eq. (5) are not computed or stored, instead the action of the Jacobian is approximated by finite differencing the nonlinear residual.

The overhead of dispatch through Python, $(T_{\mathrm{Python}} - T_{\mathrm{C}})/T_{\mathrm{C}}$ using wall-clock times, is shown in Fig. 9. The horizontal axis indicates the number of grid points, vertical axis indicates the Python overhead, determined as the quotient between Python and C wall-clock timings. For the smallest problem, the overhead in using Python is around 13%, then it decreases as the problem size grows. For medium sized to large problems the overhead is around 3%.

5. Application examples

PETSc-FEM [33,34] has been developed since 1999 at CIMEC laboratory and it is publicly available under the GPL license. *PETSc-FEM* is a parallel multiphysics code implemented in C++ and primarily targeted to 2D and 3D finite elements computations on general unstructured grids.

PETSc-FEM provides a core library to manage parallel data distribution and assembly of residual vectors and Jacobian matrices, as well as facilities for general tensor algebra computations at the level of problem-specific finite element routines. Driver programs use the core library and other utility data structures and routines to implement different applications tailored to the problem at hand. Input data in provided through text files with a specific structure describing physical and algorithmic simulation parameters, finite element meshes and boundary conditions. These input files are usually generated in a preprocessing step using external tools, typically by scripts written in Octave and Perl. Those tools are also employed for postprocessing, data analysis, and visualization of output data. Complex simulations involving the coupling of different models require the execution of separate processes cooperating through interprocess communication mechanisms (IPC) like POSIX pipes and sockets using *ad hoc* protocols. IPC is usually performed at each time step and in some cases at the inner nonlinear iterations. Setting up new applications requires hard-wiring customizations in the source code or employing techniques like dynamic loading to "hook" application-specific code implemented by end users. A series of shell scripts an makefiles control other aspects of this framework: compiling and linking, program execution, and data flow through the preprocessing/simulation/postprocessing chain.

Simulation frameworks like the previously described are common in home-grown scientific codes. However, the expertise required to handle these complexities is beyond the regular training of scientists and engineers. End-users – and particularly beginners – have to invest excessive time learning new skills to manage the plethora of different components.

A computing environment based on the Python programing language, as well as a companion "stack" of publicly available, flexible, well-though Python-based tools, is a viable alternative to more traditional frameworks. The following list summarizes the benefits we experienced after switching to a Python-centered computing environment.

- A single language takes the place of several shell scripts, makefiles, desktop computing and visualization environments, and scripting programming languages.
- Prototyping and testing of new model components as well as reuse and coupling of existing models become remarkably simpler.

- Performance critical components developed in traditional scientific programming languages are incorporated to the framework with the help of readily available tools.
- Preprocessing, postprocessing, data analysis and visualization are tightly integrated to simulation.
- The edit/compile/run cycle is considerably reduced and overall end-user productivity increases. Beginners are able to get started in a substantially shorter time frame.

Functionalities from the core *PETSc-FEM* library are made available to Python by using *SWIG*. These Python wrappers to *PETSc-FEM* interoperate with MPI for Python and PETSc for Python. MPI for Python is employed for communication and coordination of parallel runs. PETSc for Python provides the data structures and algorithms readily available in PETSc. *PETSc-FEM* primarily contributes the evaluation of residual vectors and Jacobian matrices. Python code drives the entire application, gluing the pieces together. In order to illustrate the capabilities of the tool chain consisting of MPI for Python, PETSc for Python and *PETSc-FEM*, two application examples are presented.

5.1. Hydrology: coupled subsurface/surface flow

This section summarizes some results obtained when modeling a large scale basin in Santa Fe, Argentina. A fully coupled model was developed to achieve a better understanding of the dynamics and interactions of stream/aquifer water flow and the impact of changing land and hydrology at regional levels. This model comprises the groundwater flow equation, coupled to the Saint–Venant equation for flow in open channels. The water exchange at the aquifer/river interface depends on the head differences between them and a resistivity coefficient that characterizes such interface.

5.1.1. Subsurface flow

The equation for the flow in a confined (phreatic) aquifer integrated in the vertical direction is [35]

$$S\frac{\partial\phi}{\partial t} = \nabla\cdot(K\nabla\phi) + G_{\mathrm{aq}}, \quad \text{on } \Omega_{\mathrm{aq}}\times(0,t]. \tag{7}$$

The corresponding unknown for each node is the piezometric height or the level of the phreatic surface at that point ϕ and Ω_{aq} is the aquifer domain, S the storativity, K the hydraulic conductivity and G_{aq} is the source term accounting for rain, losses from streams or other aquifers.

5.1.2. Surface flow

When velocity variations on the channel cross section are neglected, the flow can be treated as one dimensional. The equations of mass and momentum conservation on a variable cross sectional stream (in conservation form) are [36,37]

$$\begin{aligned}&\frac{\partial\mathbf{A}}{\partial t}+\frac{\partial\mathbf{Q}(\mathbf{A})}{\partial s}=G_{\mathrm{st}},\\&\frac{1}{\mathbf{A}}\frac{\partial\mathbf{Q}}{\partial t}+\frac{1}{\mathbf{A}}\frac{\partial}{\partial s}\left(\frac{\mathbf{Q}^2}{\mathbf{A}}\right)+g(S_0-S_f)+g\frac{\partial}{\partial s}(h+h_b)=0, \quad \text{on } \Omega_{st}\times(0,t],\end{aligned} \tag{8}$$

where $\mathbf{A} = \mathbf{A}(h)$ is the section of the channel occupied by water for a given water depth h and h_b is the channel bottom elevation. For instance, in rectangular channels $\mathbf{A}(h) = wh$, where w is channel width. $\mathbf{Q}$ is the discharge, G_{st} represents the gain or loss of the stream (i.e., the lateral inflow per unit length), s is the arc-length along the channel, $v = \mathbf{Q}/\mathbf{A}$ the average velocity in s-direction, v_t the velocity component in s-direction of lateral flow from tributaries. S_0 is the bottom slope, S_f is the slope friction and g is the acceleration due to gravity. The bottom shear stresses are approximated by using the Chèzy or Manning equations,

$$S_f = \begin{cases}\frac{v^2}{C_h^2}\frac{P(h)}{\mathbf{A}(h)} & \text{Chèzy model,}\\ \left(\frac{n}{a}\right)^2 v^2\frac{P^{4/3}(h)}{\mathbf{A}^{4/3}(h)} & \text{Manning model,}\end{cases} \tag{9}$$

where P is the wetted perimeter of the channel, n is Manning roughness coefficient, and a is a conversion factor ($a = 1$ for SI units).

5.1.3. River/aquifer coupling

The stream/aquifer interaction process occurs between a stream and its adjacent flood-plain aquifer. A typical discretization is shown in Fig. 10 where an element which represents the stream loss is connected to two nodes on the stream and two on the aquifer. If the stream level is over the phreatic aquifer level ($h_b + h > \phi$) then the stream losses water to the aquifer and *vice versa*. The stream gain (loss) at a point is

$$G_s = P/R_f(\phi - h_b - h), \tag{10}$$

where R_f is the resistivity factor per unit arc length of the perimeter. The corresponding loss (gain) to the aquifer is

$$G_a = -G_s\delta_{\Gamma_s}, \tag{11}$$

where Γ_s represents the planar curve of the stream and δ_{Γ_s} is a Dirac's delta distribution with a unit intensity per unit length, such that,

$$\int f(\mathbf{x})\delta_{\Gamma_s}\,d\Omega = \int_{\Gamma_s} f(\mathbf{x}(s))ds. \tag{12}$$

So that, in the context of a weighted formulation the coupling term can be computed as

$$\begin{aligned}\int W(\mathbf{x})G_a(\mathbf{x})d\Omega &= -\int W(\mathbf{x})G_s(\mathbf{x})\delta_{\Gamma_s}\,d\Omega\\&= -\int W(\mathbf{x}(s))G_s(\mathbf{x}(s))ds.\end{aligned} \tag{13}$$

where W are weighting functions.

Upon using the SUPG Galerkin finite element discretization procedure with linear triangles and/or bilinear rectangular elements and the trapezoidal rule for time integration, we obtain the system to be solved at each time step [38]

$$\mathbf{R} = \mathbf{K}(\mathbf{U}^{k+\theta})\mathbf{U}^{k+\theta} + \mathbf{B}(\mathbf{U}^{k+\theta})\frac{\mathbf{U}^{k+1}-\mathbf{U}^{k}}{\Delta t} - \mathbf{G}^{k+\theta} = 0, \tag{14}$$

where $\mathbf{U}^{k+\theta} = \theta\mathbf{U}^{k+1} + (1-\theta)\mathbf{U}^k$, $\mathbf{U}$ is the state for the coupled problem (i.e., phreatic level and Saint–Venant state variables), θ is the time-weighting factor satisfying $0 \leqslant \theta \leqslant 1$, Δt is the time increment and k denotes the number of time steps. $\mathbf{K}$ and $\mathbf{B}$ are the non-symmetric stiffness matrix and the symmetric mass matrix, respectively ($\mathbf{K}$ and $\mathbf{B}$ depend on $\mathbf{U}$), $\mathbf{G}$ is the source vector and $\mathbf{R}$ is the residual vector.

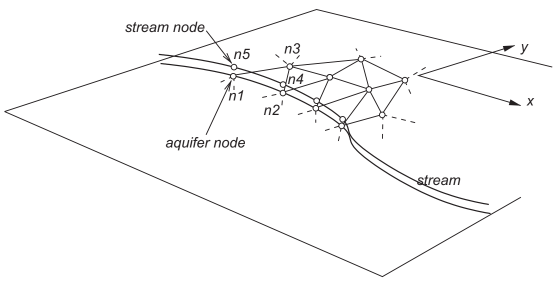


Fig. 10. Stream/aquifer coupling.

5.1.4. Numerical simulations

An example of surface and subsurface interaction flow is presented. The study area represents a third of the total area of Santa Fe province (Argentina), amounting to roughly 33,000 km^2 (see Fig. 11). A period of 12 months is simulated where the total precipitation is the annual average observed in recent years (1000 mm/year), but divided in two wet seasons with a rainfall rate of 2000 mm/year (March–April and September–October) and dry seasons of 500 mm/year (the rest of the year).

At time $t = 0$ s (January/1) the piezometric head in the phreatic aquifer is 30 m above the aquifer bottom, while the water depth in stream is 10 m above the stream bed. The hydraulic conductivity and storativity of phreatic aquifer are $K = 2 \cdot 10^{-3}$ m/s and $S = 2.5 \cdot 10^{-2}$ m/s, respectively. The Manning friction law is adopted for this case. The stream channel roughness is $n = 3 \cdot 10^{-3}$ and the river width is $w = 10$ m. The average value of resistivity river walls is $R_f = 10^5$ s. The computational mesh has 1.7 million triangles (see a detail in Fig. 12) and the drainage network has more than 150 branches discretized with 70 thousand elements, creating an average space of 100 m between river nodes. The time step adopted in simulations is $\Delta t = 1$ day.

Fig. 13 shows the phreatic elevation in four different days of the simulated period. The phreatic levels increases after the two first dry months (January–February) when dry period starts (March–April). At the same time, when consider the river vicinities region where the subsurface/surface flow interaction process takes place, we see an increment of the river water level due to the recharge from the elevated aquifer phreatic levels in wet seasons. The opposite process is observed in dry periods.

5.2. Microfluidics: lab-on-a-chip simulations

A *Lab-on-a-chip* (LOC) performs the functions of classical analytical devices in small units of a few square centimeters in size [39]. They are used in a variety of chemical, biological and medical applications. The benefits of LOC are the reduction of consumption

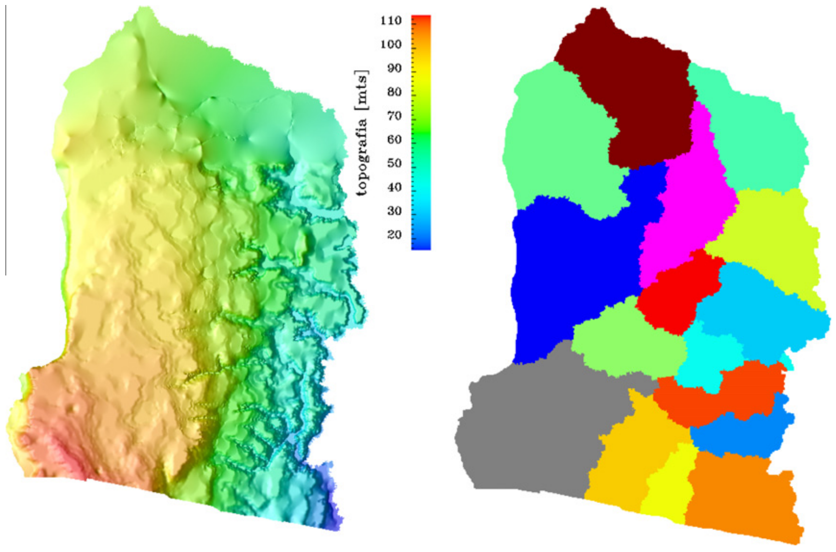


Fig. 11. Terrain elevation of the computational domain and its parallel partition.

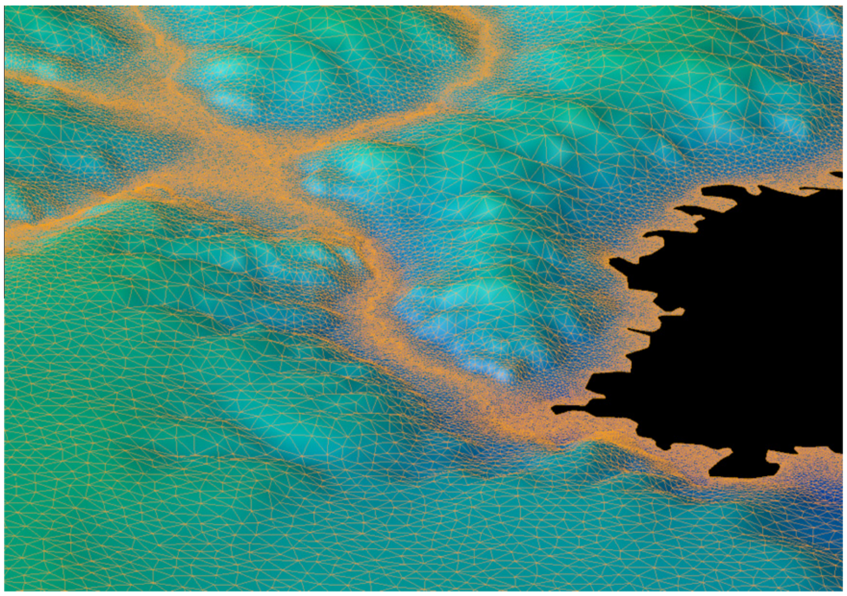


Fig. 12. Mesh over topography detail.

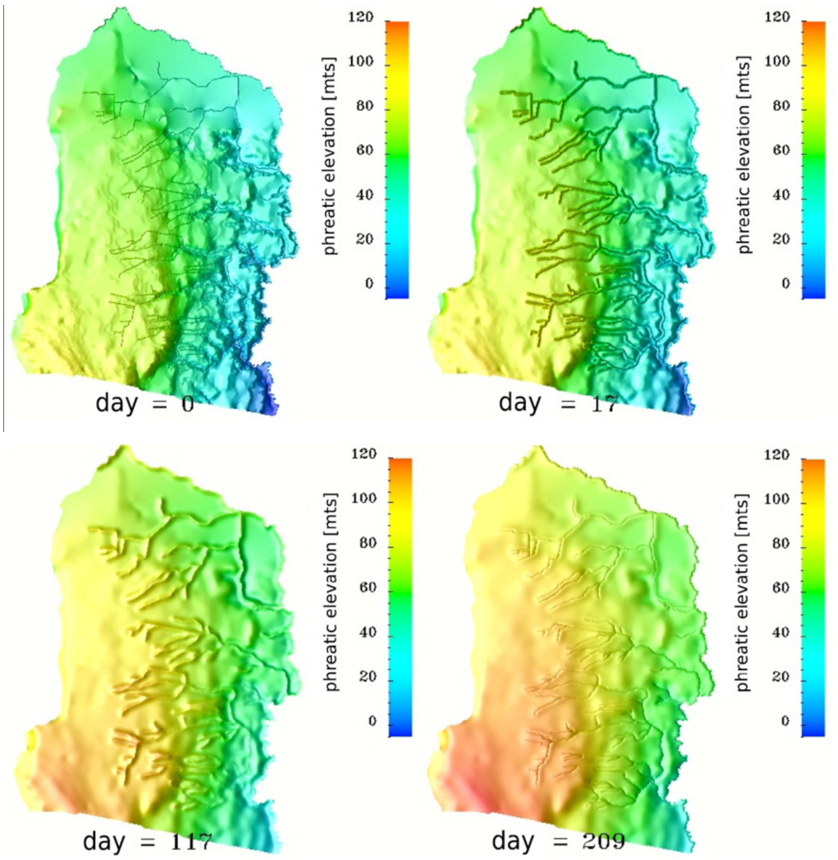


Fig. 13. Aquifer phreatic elevation during a periodic rainfall.

of samples and reagents, shorter analysis time, greater sensitivity, portability and disposability. There has been a huge interest in these devices in the past decade that led to a wide commercial range of products.

Historically, the most important techniques developed in LOC devices are electrophoretic separations [40,41]. They are based on the mobility of ions under the action of an external electric field. These techniques are widely used in chemical and biochemical analysis. As microchips for electrophoresis becomes increasingly complex, simulation tools are required to prototype numerically these devices, as well as to control and optimize handling [42].

Numerical simulation of a two dimensional electrophoresis (2DE) device is presented. Simulations were carried out by using a 3D time-dependent finite element model for electrophoretic processes in microfluidic chips. Two-dimensional electrophoretic separations consist of two independent mechanisms that are employed sequentially. The separation efficiency is estimated as the product of the independent efficiency of each method, provided the methods are uncoupled. Two such mechanisms, satisfying uncoupling, are free flow isoelectric focusing (FFIEF) and capillary zone electrophoresis (CZE). FFIEF is a technique in which an electric field and a pH gradient are established perpendicularly to a flowing sample solution, allowing components to focus at its stable isoelectric point (pI) [43–45]. CZE is based on the application of an electric potential difference along a capillary channel, then electric forces generate electrosmotically-driven fluid flow and induce species migration along the channel axis, yielding the separation according to their electrophoretic mobilities [46,47].

5.2.1. Modeling

Mathematical modeling of electrophoretic separations carried out on LOC involves fluid, electric and concentration fields, and the strong coupling between them. In this particular case, due to the high reaction rate, the fluid and the electric fields can be treated in a quasi-stationary form, reducing the complexity of the solving process [48]. The set of differential equations that were solved on the LOC geometry Ω_{loc}, can be summarized as:

$$\nabla\cdot\mathbf{u}=0,\quad \text{on } \Omega_{\text{loc}}, \tag{15}$$

$$\rho(\mathbf{u}\cdot\nabla\mathbf{u})=\nabla\cdot(-p\mathbf{I}+\mu(\nabla\mathbf{u}+\nabla\mathbf{u}^T)),\quad \text{on } \Omega_{\text{loc}}, \tag{16}$$

$$\nabla\cdot\left(-F^2\sum_{j=1}^{N}z_j^2\Omega_j c_j\nabla\phi-F\sum_{j=1}^{N}z_jD_j\nabla c_j+F\sum_{i=j}^{N}z_jc_j\mathbf{u}\right)=0,\quad \text{on } \Omega_{\text{loc}}, \tag{17}$$

$$\frac{\partial c_j}{\partial t}+\nabla\cdot\left(-z_j\Omega_j\nabla\phi c_j+\mathbf{u}c_j-D_j\nabla c_j\right)-r_j=0,\quad \text{on } \Omega_{\text{loc}}\times(0,t]. \tag{18}$$

Eqs. (15) and (16) are the Navier–Stokes equations for solving the fluid field, where $\mathbf{u}$ is the velocity, p is the pressure, and μ is the dynamic viscosity. In this simulation, in order to model the electroosmotic flow, a slip velocity is set as boundary condition. The magnitude for this velocity ($\mathbf{u_{eo}}$) is based on the Helmholtz–Smoluchowski approximation [46]:

$$\mathbf{u_{eo}}=-\frac{\epsilon\zeta_w(-\nabla\phi)}{\mu},\quad \text{on } \Gamma_{\text{eo}}, \tag{19}$$

where ϵ is the electric permittivity, ζ_m is the electrokinetic potential of the solid walls Γ_{eo}, and ϕ is the electric potential.

Eq. (17) expresses the electric charge conservation for the domain as a combination of migrative, diffusive and advective components for the motion of all charges present in the solution. In this case, for the j-species, z_j represents the valence, Ω_j is the mobility, c_j the concentration in mol m^{-3}, and D_j the diffusion coefficient.

Finally, Eq. (18) is the mass transport equation for a generic j-species, were r_j is the reaction term. Different electrolytes (acids, bases and ampholytes), analytes, and particularly the hydrogen ion have to be considered in order to determine the reaction terms. In electrolyte chemistry the processes of association and dissociation are much faster than the transport electrokinetic processes, hence, it is a good approximation to adopt chemical equilibrium constants to model the reactions of weak electrolytes [49], while strong electrolytes are considered as completely dissociated.

In solving 2DE amphoteric species are mainly involved. The reactions associated to a generic ampholyte AH can be summarized as [49]:

$$\mathrm{AH} \underset{k_{a2}}{\overset{k_{a1}}{\rightleftarrows}} \mathrm{A}^- + \mathrm{H}^+ \tag{20}$$

$$\mathrm{AH}_2^+ \underset{k_{b2}}{\overset{k_{b1}}{\rightleftarrows}} \mathrm{AH} + \mathrm{H}^+ \tag{21}$$

where k_{a1}, k_{b1} are the dissociation rates, and k_{a2}, k_{b2} the association rates for the ampholyte AH. Then the equilibrium state is characterized by,

$$\frac{k_{a2}}{k_{a1}} = \frac{[\mathrm{A}^-][\mathrm{H}^+]}{[\mathrm{AH}]} = K_a \tag{22}$$

$$\frac{k_{b2}}{k_{b1}} = \frac{[\mathrm{AH}][\mathrm{H}^+]}{[\mathrm{AH}_2^+]} = K_b \tag{23}$$

where K_a and K_b are the dissociation constants for the equilibrium state, and the square brackets represent concentration of the given species. The corresponding expressions of r_j are obtained as follows:

$$r_{\mathrm{A}^-} = -k_{a1}[\mathrm{A}^-][\mathrm{H}^+] + k_{a2}[\mathrm{AH}] \tag{24}$$

$$r_{\mathrm{AH}} = k_{a1}[\mathrm{A}^-][\mathrm{H}^+] - k_{a2}[\mathrm{AH}] - k_{b1}[\mathrm{AH}][\mathrm{H}^+] + k_{b2}[\mathrm{AH}_2^+] \tag{25}$$

$$r_{\mathrm{AH}_2^+} = k_{b1}[\mathrm{AH}][\mathrm{H}^+] - k_{b2}[\mathrm{AH}_2^+] \tag{26}$$

$$r_{\mathrm{H}^+} = -k_{a1}[\mathrm{A}^-][\mathrm{H}^+] + k_{a2}[\mathrm{AH}] - k_{b1}[\mathrm{AH}][\mathrm{H}^+] + k_{b2}[\mathrm{AH}_2^+] \tag{27}$$

In Eq. (27) the water dissociation term is not included due to the fact that this reaction is several orders of magnitude faster than reactions (20) and (21) [49], then $[\mathrm{OH}^-]$ can be calculated directly as

$$[\mathrm{OH}^-] = \frac{K_w}{[\mathrm{H}^+]} \tag{28}$$

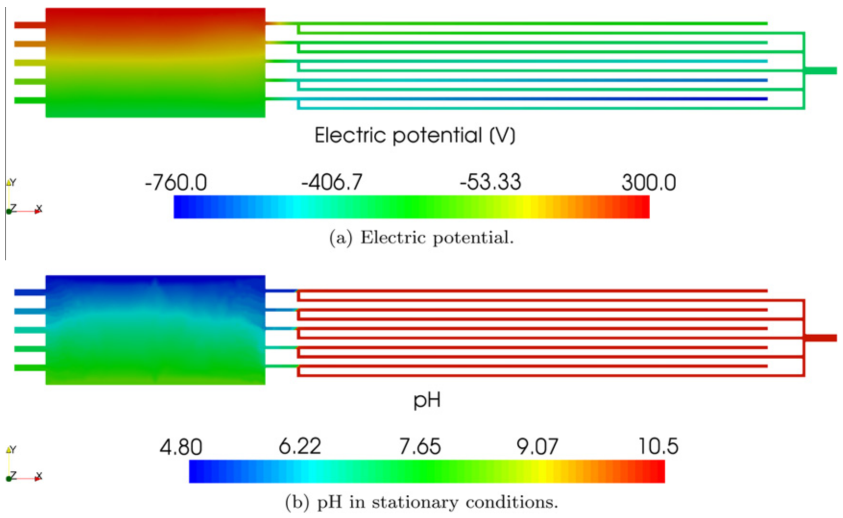


Fig. 14. Initial distributions of electric potential and pH.

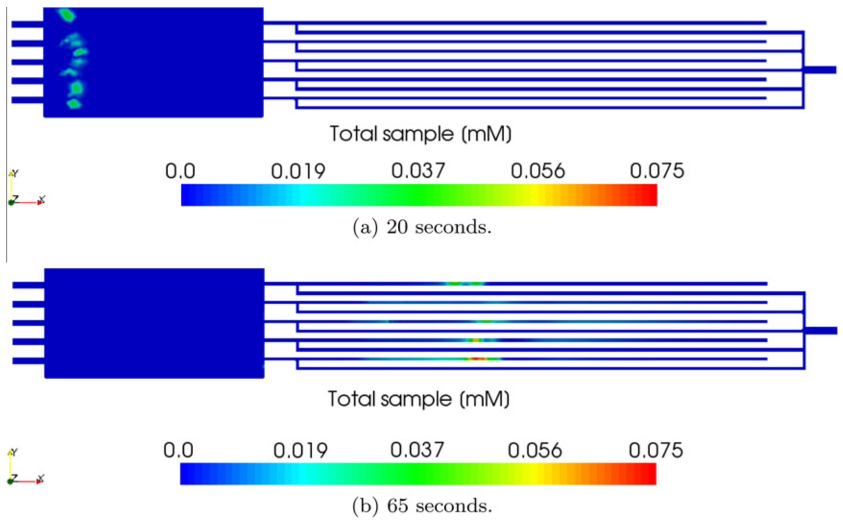


Fig. 15. Total sample distribution at 20 and 65 s.

where $K_w = 10^{-14}\,\text{mol}^2\,\text{m}^{-6}$ is the dissociation constant for pure water at 25 °C.

5.2.2. Simulation results

A 2DE separation involving FFIEF and CZE is simulated on a LOC prototype. FFIEF is carried out on the left part ($10 \times 3500 \times 7000\,\mu\text{m}^3$), then samples flow through five CZE channels ($10 \times 1000 \times 16{,}000\,\mu\text{m}^3$) on the right part. Mesh consist of 175 thousand linear triangles, with a total amount of 6 millions of degrees of freedom.

Boundary conditions for the fluid field are those stated by the Eq. (19), and the pressure is set to 0 Pa at the outlets. In the case of the electric field, Dirichlet boundary conditions are set where the electric potential is applied, and natural Neumann conditions on the other walls are set. Finally, for the concentration field, advective flux is set at the inlets and outlets, and natural Neumann conditions are set on the walls.

The applied electric potential differences (Fig. 14(a)) are fixed during the operation to provide the system with a transverse electric field in the FFIEF region and an axial electric field in the CZE channels. The pH gradient for FFIEF is established by focusing 20 ampholytes between two sheath flows of basic and acidic solutions. A concentrated basic buffer solution is continuously injected from the inlet at the right. When stationary conditions are reached, a near-linear pH gradient is developed (Fig. 14(b)).

The proposed numerical prototype is employed to separate a sample of 10 proteins. Proteins are injected from the central channel. After a few seconds, the different bands of isoelectric points are developed. In this particular case there are eight bands, consequently, there are three or four proteins that cannot be effectively separated by FFIEF, thus CZE is employed as an additional separation method. After leaving the FFIEF chamber, proteins separate electrophoretically completing the successful separation of the 10 sample compounds. Total sample distributions at two different instants of time during the separation process are shown in Fig. 15.

One of the aims of this tool is to provide information about the separative performance of LOC. In analytical and bioanalytical chemistry, separative performance of two-dimensional electrophoresis assays can be evaluated by using a customary representation in a two-dimensional map. This map contains information of the isoelectric points and the electrophoretic mobilities of the analytes present in sample. Results in this format are shown in Fig. 16.

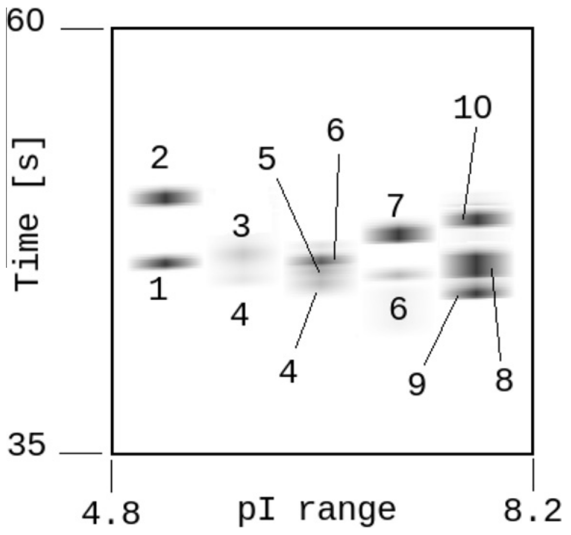


Fig. 16. Two dimensional map for the separation.

6. Conclusions

Python is an attractive language for rapid development of small scripts and code prototypes as well as large applications and highly portable and reusable modules and libraries. Running Python on parallel computers is a feasible alternative for decreasing the costs of software development targeted to HPC systems.

In this work, two software components facilitating the access to parallel distributed computing resources within a Python programming environment were presented: MPI for Python and PETSc for Python. These packages are able to support serious medium and large scale parallel applications.

Efficiency tests have shown that performance degradation is not prohibitive. In comparison to pure C codes, MPI for Python can communicate Python array data at nearly full speed over Gigabit Ethernet and around half speed over shared memory channels. PETSc for Python overhead is consistently less than 10%.

This software suite is supporting research activities in a variety of fields. Applications examples related to finite elements simulations of hydrology and microfluidic problems were presented.

Acknowledgments

The authors extend sincere thanks to Christopher Kees, Jed Brown, and the anonymous reviewer for their kind advice, insightful comments and constructive suggestions.

This work has received financial support from *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina, Grant PIP 5271/05), *Universidad Nacional del Litoral* (UNL, Argentina, Grant CAI+D 2009 65/334), *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT, Argentina, Grants PICT 01141/2007, PICT 0270/2008, PICT-1506/2006).

Appendix A. Project development and support

MPI for Python and PETSc for Python are active software projects. New features and enhancements are being added on regular basis in order to keep Python interfaces in accordance to the updates of the MPI standard and PETSc. The testing process is supported by automated unit testing based on the `unittest` package from the Python standard library. These tests are run regularly on a variety of platforms and computer architectures.

MPI for Python is hosted on Google Code project hosting service (http://mpi4py.googlecode.com). This service provides version control repository (http://mpi4py.googlecode.com/svn/), issue tracker (http://code.google.com/p/mpi4py/issues/list), and release downloads (http://code.google.com/p/mpi4py/downloads/). Google Groups service hosts an on-line discussion and support forum (http://groups.google.com/group/mpi4py) and a mailing list (mpi4py@googlegroups.com). PETSc for Python is hosted on Google Code project hosting service (http://petsc4py.googlecode.com). This service provides version control repository (http://petsc4py.googlecode.com/hg/), issue tracker (http://code.google.com/p/petsc4py/issues/list), and release downloads (http://code.google.com/p/petsc4py/downloads/). PETSc for Python uses the same support channels of PETSc (petsc-users@mcs.anl.gov, petsc-maint@mcs.anl.gov, and petsc-dev@mcs.anl.gov mailing lists).

References

[1] Dalcin L. MPI for Python; 2005–2010. <http://mpi4py.googlecode.com>.
[2] Dalcin L. PETSc for Python; 2005–2010. <http://petsc4py.googlecode.com/>.
[3] van Rossum G. Python programming language; 1990–2010. <http://www.python.org/>.
[4] van Rossum G. Python reference manual; 2010. <http://docs.python.org/ref/ref.html>.

[5] Millman KJ, Aivazis M. Python for scientists and engineers. Comput Sci Eng 2011;13(2):9–12. doi:10.1109/MCSE.2011.36.
[6] Pérez F, Granger B, Hunter J. Python: an ecosystem for scientific computing. Comput Sci Eng 2011;13(2):13–21. doi:10.1109/MCSE.2010.119.
[7] van der Walt S, Colbert S, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22–30. doi:10.1109/MCSE.2011.37.
[8] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn D, Smith K. Cython: the best of both worlds. Comput Sci Eng 2011;13(2):31–9. doi:10.1109/MCSE.2010.118.
[9] Ramachandran P, Varoquaux G. Mayavi: 3D visualization of scientific data. Comput Sci Eng 2011;13(2):40–51. doi:10.1109/MCSE.2011.35.
[10] Oliphant T. NumPy: numerical Python; 2005–2010. <http://numpy.scipy.org/>.
[11] Peterson P. F2PY: Fortran to Python interface generator; 2000–2010. <http://cens.ioc.ee/projects/f2py2e/>.
[12] Cython Team. Cython: C – extensions for Python; 2007–2010. <http://www.cython.org>.
[13] Beazley DM. SWIG: simplified wrapper and interface generator; 1996–2010. <http://www.swig.org/>.
[14] Beazley DM, Lomdahl PS. Feeding a large scale physics application to Python. In: Proceedings of 6th international Python conference, San Jose, California; 1997. p. 21–9.
[15] Kadau K, Germann TC, Lomdahl PS. Molecular dynamics comes of age: 320 billion atom simulation on BlueGene/L. Int J Modern Phys C 2006;17:1755–61.
[16] Forum MPI. MPI: a message passing interface standard. Int J Supercomput Appl 1994;8(3/4):159–416.
[17] MPI Forum. MPI-2: a message passing interface standard. High Perform Comput Appl 1998;12(1–2):1–299.
[18] Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI – the complete reference. The MPI core of scientific and engineering computation, vol. 1. Cambridge, MA, USA: MIT Press; 1998.
[19] Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Nitzberg B, Saphir W, et al. MPI – the complete reference. The MPI-2 extensions of scientific and engineering computation, vol. 2. Cambridge, MA, USA: MIT Press; 1998.
[20] MPICH2 Team, MPICH2: a portable implementation of MPI; 2003–2010. <http://www-unix.mcs.anl.gov/mpi/mpich2/>.
[21] Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput 1996;22(6):789–828.
[22] Open MPI Team, Open MPI: open source high performance computing; 2004–2010. <http://www.open-mpi.org/>.
[23] Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, et al. Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI users' group meeting, Budapest, Hungary; 2004. p. 97–104.
[24] Balay S, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, et al. PETSc web page; 2010. <http://www.mcs.anl.gov/petsc>.
[25] Balay S, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, et al. PETSc users manual, Tech. Rep. ANL-95/11 – Revision 3.1. Argonne National Laboratory; 2010.
[26] Balay S, Gropp WD, McInnes LC, Smith BF. Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP, editors. Modern software tools in scientific computing. Birkhäuser Press; 1997. p. 163–202.
[27] Falgout R, Jones J, Yang U. Numerical solution of partial differential equations on parallel computers, vol. 51. Springer-Verlag; 2006. p. 267–94 [chapter: the design and implementation of hypre, a library of parallel high performance preconditioners].
[28] Heroux M, Bartlett R, Hoekstra VHR, Hu J, Kolda T, Lehoucq R, et al. An overview of Trilinos. Tech. Rep. SAND2003-2927. Sandia National Laboratories; 2003.
[29] Amestoy PR, Duff IS, L'Excellent J-Y, Koster J. A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J Matrix Anal Appl 2001;23(1):15–41.
[30] MPI Forum. MPI: a message passing interface standard, version 2.2; 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.
[31] Miller P. pyMPI project page, 2000–2011. <http://pympi.sourceforge.net/>.
[32] Nielsen O. Pypar project page; 2002–2011. <http://code.google.com/p/pypar/>.
[33] Sonzogni VE, Yommi AM, Nigro NM, Storti MA. A parallel finite element program on a Beowulf cluster. Adv Eng Softw 2002;33(7–10):427–43.
[34] Storti MA, Nigro N, Paz R. PETSc-FEM: a general purpose, parallel, multi-physics FEM program; 1999–2010. <http://www.cimec.org.ar/petscfem>.
[35] Rodríguez L. Investigation of stream–aquifer interactions using a coupled surface-water and ground-water flow model. PhD thesis. University of Arizona; 1995.
[36] Whitham G. Linear and nonlinear waves, pure and applied mathematics. A Wiley-Interscience series of texts, monographs, and tracts John Wiley & Sons Inc.; 1974.
[37] Hirsch C. Numerical computation of internal and external flows. Wiley Series in numerical methods in engineering, vol. II John Wiley & Sons Inc.; 1990.
[38] Paz R, Storti M. An interface strip preconditioner for domain decomposition methods: application to hydrology. Int J Numer Methods Eng 2005;62(13):1873–94.
[39] Manz A, Graber N, Widmer H. Miniaturized total chemical analysis systems: a novel concept for chemical sensing. Sensor Actuator B 1990;1:244–8.
[40] Landers JP. Handbook of capillary and microchip electrophoresis and associated microtechniques. 3rd ed. CRC Press; 2007.
[41] Tian W-C, Finehout E. Microfluidics for biological applications. 1st ed. Springer; 2008.
[42] Erickson D. Towards numerical prototyping of labs-on-chip: modeling for integrated microfluidic devices. Microfluid Nanofluid 2005;1(4):301–18.
[43] Kohlheyer D, Eijkel JCT, van den Berg A, Schasfoort RBM. Miniaturizing free-flow electrophoresis — a critical review. Electrophoresis 2008;29(5):977–93.
[44] Turgeon RT, Bowser MT. Micro free-flow electrophoresis: theory and applications. Anal Bioanal Chem 2009;394(1):187–98.
[45] Sommer G, Hatch A. IEF in microfluidic devices. Electrophoresis 2009;30:742–57.
[46] Probstein R. Physicochemical hydrodynamics. An introduction. 2nd ed. Wiley-Interscience; 2003.
[47] Hunter R. Foundations of colloid science. 2nd ed. Oxford University Press; 2001.
[48] Kler PA, Berli CLA, Guarnieri FA. Modelling and high performance simulation of electrophoretic techniques in microfluidic chips. Microfluid Nanofluid 2010;10(1):187–98.
[49] Arnaud I, Josserand J, Rossier J, Girault H. Finite element simulation of off-gel buffering. Electrophoresis 2002;23:3253–61.