Parallel Distributed Computing using Python

Lisandro Dalcin
dalcinl@gmail.com

Joint work with
Pablo Kler **Rodrigo Paz**
Mario Storti **Jorge D'Elía**

Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Instituto de Desarrollo Tecnológico para la Industria Química (INTEC)
Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC)
http://www.cimec.org.ar

HPCLatAm 2011
Córdoba, Argentina
September 1, 2011

Outline

Motivation

- Apply numerical methods in complex problems of medium to large scale in science and engineering
  - multiphysic nature
  - strong/loose coupling
  - multiple interacting scales
- Ease the access to computing resources in distributed memory architectures (clusters, supercomputers)
  - beginners
  - scientists and engineers
  - experienced software developers

Objectives

- Develop extension modules for the Python programming language providing access to MPI and PETSc libraries
  - message passing
  - parallel linear algebra
  - linear and nonlinear solvers
- Perform computer-based simulations
  - problems modeled by partial differential equations (PDEs)
  - problems related to computational fluid mechanics (CFD)

Why Python?

- very clear, readable syntax
- natural expression of procedural code
- very high level dynamic data types
- intuitive object orientation
- exception-based error handling
- full modularity, hierarchical packages
- comprehensive standard library
- extensible with C and C++
- embeddable within applications

Python for Scientific Computing

- Scientific computing (and particularly HPC) has been traditionally dominated by C, C++, y Fortran
- High level and general purpouse computing environments (*Maple*, *Mathematica*, *MATLAB*) got popular since the 90's
- Python is becoming increasingly popular in the scientific community since the 2000's
- Key feature: easy to extend with C, C++, Fortran
  - NumPy
  - SciPy
  - SymPy
  - Cython
  - SWIG
  - F2Py

What is MPI?

Message Passing Interface

`http://www.mpi-forum.org`

- Standardized message passing system
  - platform-independent (POSIX, Windows)
  - many implementations and vendors
    - MPICH2, Open MPI
    - HP, Intel, Oracle, Microsoft
- Specifies semantics of a set of library routines
  - No special compiler support
  - Language-neutral (C, C++, Fortran 90)
- Standard versions (backward-compatible)
  - MPI-1 (1994, 2008)
  - MPI-2 (1996, 2009)
  - MPI-3 (under development)

What is PETSc?

Portable, Extensible Toolkit for Scientific Computation

`http://www.mcs.anl.gov/petsc`

- PETSc is a suite of algorithms and data structures for the numerical solution of
  - problems in science and engineering
  - based on partial differential equations models
  - discretized with finite differences/volumes/elements
  - leading to large scale applications
- PETSc employs the MPI standard for parallelism
- PETSc has an OO design, it is implemented in C, can be used from C++ , provides a Fortran 90 interface

MPI for Python (**mpi4py**)

- Python bindings for **MPI**
- API based on the standard MPI-2 C++ bindings
- Supports all MPI features
  - targeted to MPI-2 implementations
  - also works with MPI-1 implementations

[mpi4py] Implementation

Implemented with **Cython**

- Code base far easier to write, maintain, and extend
- Faster than other solutions (mixed Python and C codes)
- A *pythonic* API that runs at C speed !

[mpi4py] Implementation – Cython [1]

```
cdef import from "mpi.h":
    ctypedef void* MPI_Comm
    MPI_Comm MPI_COMM_NULL
    MPI_Comm MPI_COMM_SELF
    MPI_Comm MPI_COMM_WORLD
    int MPI_Comm_size(MPI_Comm,int*)
    int MPI_Comm_rank(MPI_Comm,int*)

cdef inline int CHKERR(int ierr) except -1:
    if ierr != 0:
        raise RuntimeError("MPI error code %d" % ierr)
    return 0
```

[mpi4py] Implementation – Cython [2]

```
cdef class Comm:
    cdef MPI_Comm ob_mpi
    ...
    def Get_size(self):
        cdef int size
        CHKERR( MPI_Comm_size(self.ob_mpi, &size) )
        return size
    def Get_rank(self):
        cdef int rank
        CHKERR( MPI_Comm_rank(self.ob_mpi, &rank) )
        return rank
    ...

cdef inline Comm NewComm(MPI_Comm comm_c):
    cdef Comm comm_py = Comm()
    comm_py.ob_mpi = comm_c
    return comm_py

COMM_NULL  = NewComm(MPI_COMM_NULL)
COMM_SELF  = NewComm(MPI_COMM_SELF)
COMM_WORLD = NewComm(MPI_COMM_WORLD)
```

[mpi4py] Features – MPI-1

- Process groups and communication domains
  - intracommunicators
  - intercommunicators
- Point to point communication
  - blocking (send/recv)
  - nonblocking (isend/irecv + test/wait)
- Collective operations
  - Synchronization (barrier)
  - Communication (broadcast, scatter/gather)
  - Global reductions (reduce, scan)

[mpi4py] Features – MPI-2

- Extended collective operations.
- Dynamic process management (spawn, connect/accept)
- Parallel I/O (read/write)
- One sided operations, aka RMA (put/get/accumulate)

[mpi4py] Features – Communicating of Python objects

- High level and very convenient, based in `pickle` serialization
- Can be slow for large data (CPU and memory consuming)

At the sending side . . .

`comm.send(object)` $\longrightarrow$ `pickle.dump()` $\longrightarrow$ `MPI_Send()`

At the receiving side . . .

`object = comm.recv()` $\longleftarrow$ `pickle.load()` $\longleftarrow$ `MPI_Recv()`

```
from mpi4py import MPI

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

if rank == 0:
    msg1 = [77, 3.14, 2+3j, "abc", (1,2,3,4)]
elif rank == 1:
    msg1 = {"A": [2,"x",3], "B": (2.17,1+3j)}

wt = MPI.Wtime()
if rank == 0:
    comm.send(msg1, 1, tag=0)
    msg2 = comm.recv(None, 1, tag=7)
elif rank == 1:
    msg2 = comm.recv(None, 0, tag=0)
    comm.send(msg1, 0, tag=7)
wt = MPI.Wtime() - wt
```

[mpi4py] Features – Communicating array data

- Lower level, slightly more verbose
- Very fast, almost C speed (for messages above 5-10 KB)

At the sending side . . .

```
message = [object, (count, displ), datatype]
```

```
comm.Send(message) ⟶ MPI_Send()
```

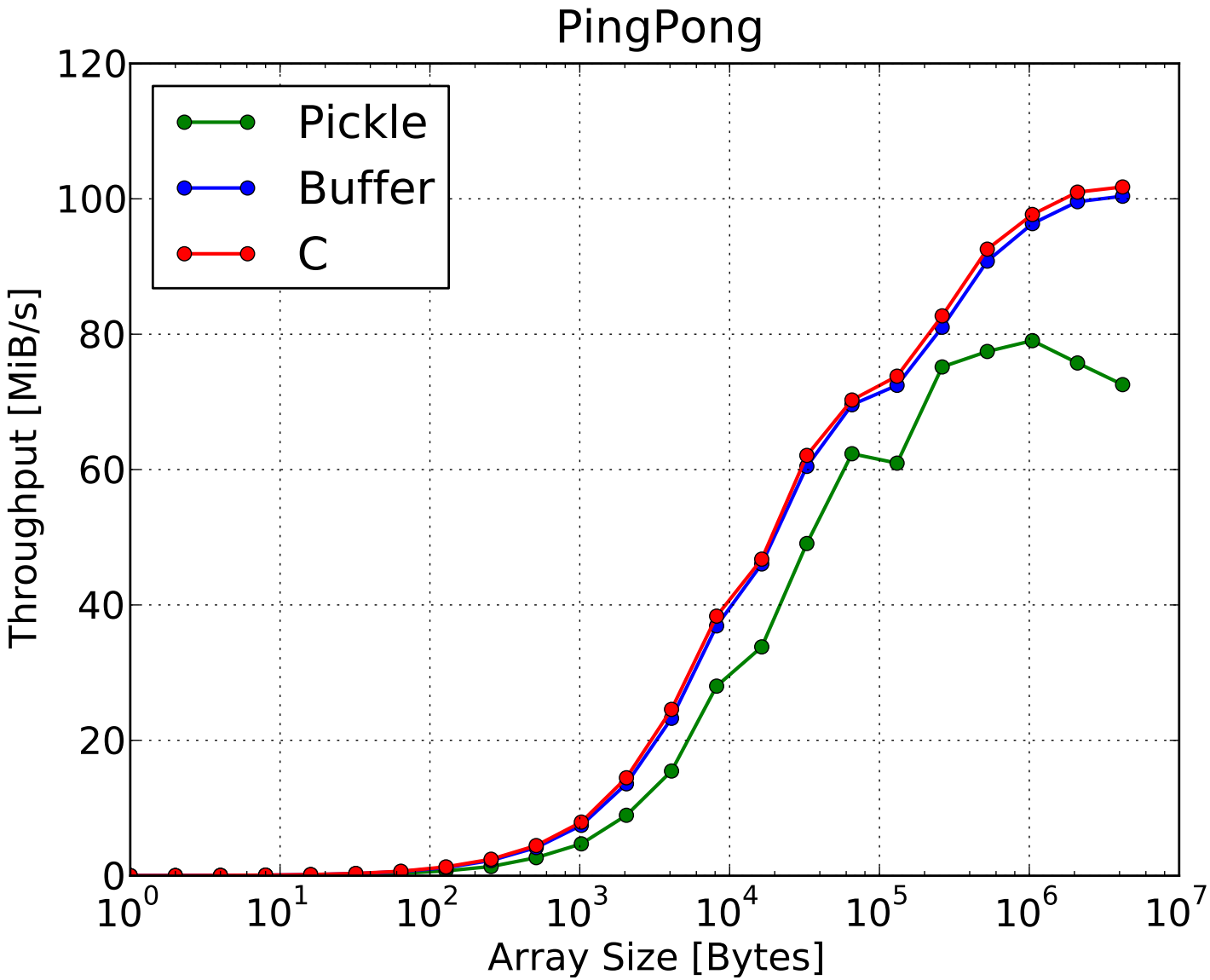
At the receiving side . . .

```
message = [object, (count, displ), datatype]
```

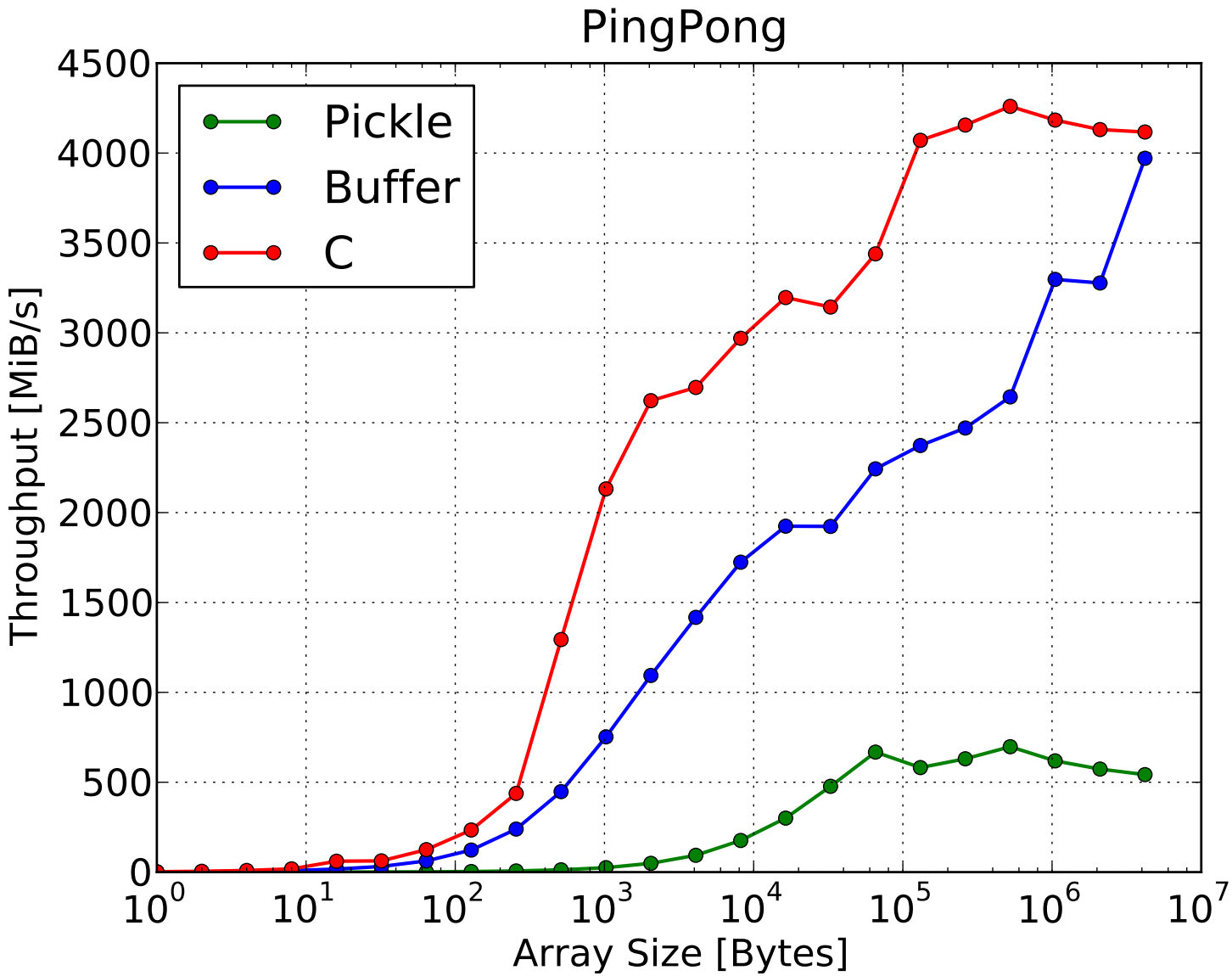
```
comm.Recv(message) ⟵ MPI_Recv()
```

```
from mpi4py import MPI
import numpy as np

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

array1 = np.arange(2**16, dtype=np.float64)
array2 = np.empty(2**16, dtype=np.float64)

wt = MPI.Wtime()
if rank == 0:
    comm.Send([array1, MPI.DOUBLE], 1, tag=0)
    comm.Recv([array2, MPI.DOUBLE], 1, tag=7)
elif rank == 1:
    comm.Recv([array2, MPI.DOUBLE], 0, tag=0)
    comm.Send([array1, MPI.DOUBLE], 0, tag=7)
wt = MPI.Wtime() - wt
```

Point to Point Throughput – Gigabit Ethernet



Point to Point Throughput – Shared Memory

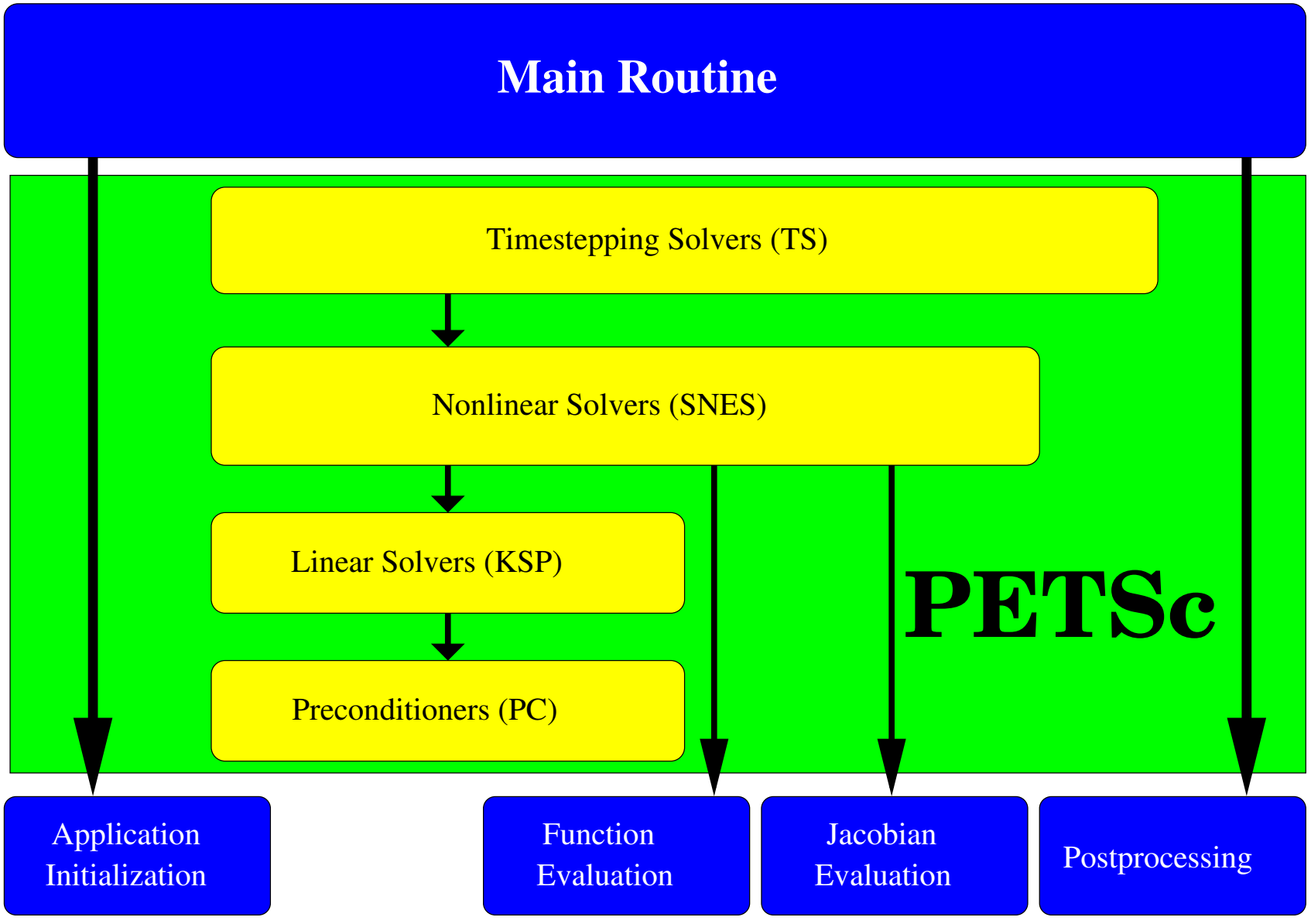


PETSc for Python (**petsc4py**)

- Python bindings for **PETSc**
- Implemented with **Cython**
- Supports most important PETSc features
- Pythonic API that better match PETSc's OO design
  - class hierarchies, methods, properties
  - automatic object lifetime management
  - exception-based error handling

[petsc4py] Features – PETSc components

- **Index Sets**: permutations, indexing, renumbering
- **Vectors**: sequential and distributed
- **Matrices**: sequential and distributed, sparse and dense
- **Linear Solvers**: Krylov subspace methods
- **Preconditioners**: sparse direct solvers, multigrid
- **Nonlinear Solvers**: line search, trust region, matrix-free
- **Timesteppers**: time-dependent, linear and nonlinear PDE's


Main Routine
Timestepping Solvers (TS)
Nonlinear Solvers (SNES)
Linear Solvers (KSP)
Preconditioners (PC)
PETSc
Application Initialization
Function Evaluation
Jacobian Evaluation
Postprocessing

[petsc4py] Vectors (Vec) – CG Method

$cg(A, x, b, i_{max}, \epsilon)$:

$i \Leftarrow 0$

$r \Leftarrow b - Ax$

$d \Leftarrow r$

$\delta_0 \Leftarrow r^T r$

$\delta \Leftarrow \delta_0$

while $i < i_{max}$ and

$\delta > \delta_0 \epsilon^2$ do :

$q \Leftarrow Ad$

$\alpha \Leftarrow \dfrac{\delta}{d^T q}$

$x \Leftarrow x + \alpha d$

$r \Leftarrow r - \alpha q$

$\delta_{old} \Leftarrow \delta$

$\delta \Leftarrow r^T r$

$\beta \Leftarrow \dfrac{\delta}{\delta_{old}}$

$d \Leftarrow r + \beta d$

$i \Leftarrow i + 1$

```
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
```

[petsc4py] Matrices (Mat) [1]

```
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
```

[petsc4py] Matrices (Mat) [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()
```

[petsc4py] Linear Solvers (KSP+PC)

```
# create linear solver,
ksp = PETSc.KSP()
ksp.create(PETSc.COMM_WORLD)

# use conjugate gradients method
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)
```

[petsc4py] Interoperability

Support for wrapping other PETSc-based C/C++/F90 codes

- using **Cython** (`cimport` statement)
- using **SWIG** (*typemaps* provided)
- using **F2Py** (`fortran` attribute)

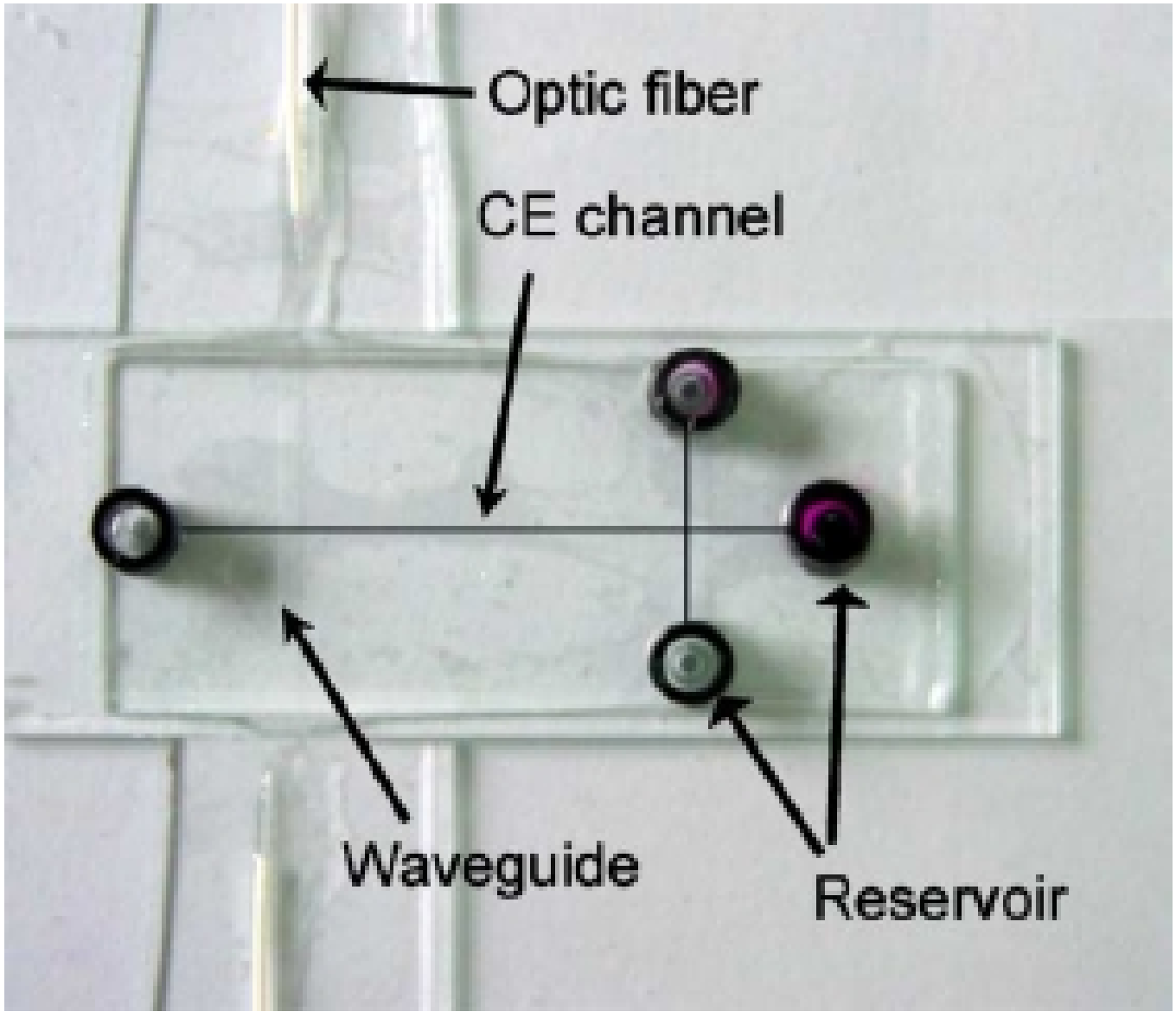
[petsc4py] Interoperability – SWIG

```
%module MyApp

%include petsc4py/petsc4py.i
%{
#include "MyApp.h"
%}

class Nonlinear {
  Nonlinear(MPI_Comm comm,
            const char datafile[]);
  Vec  createVec();
  Vec  createMat();
  void evalFunction(SNES snes, Vec X, Vec F);
  bool evalJacobian(SNES snes, Vec X, Mat J, Mat P);
};
```

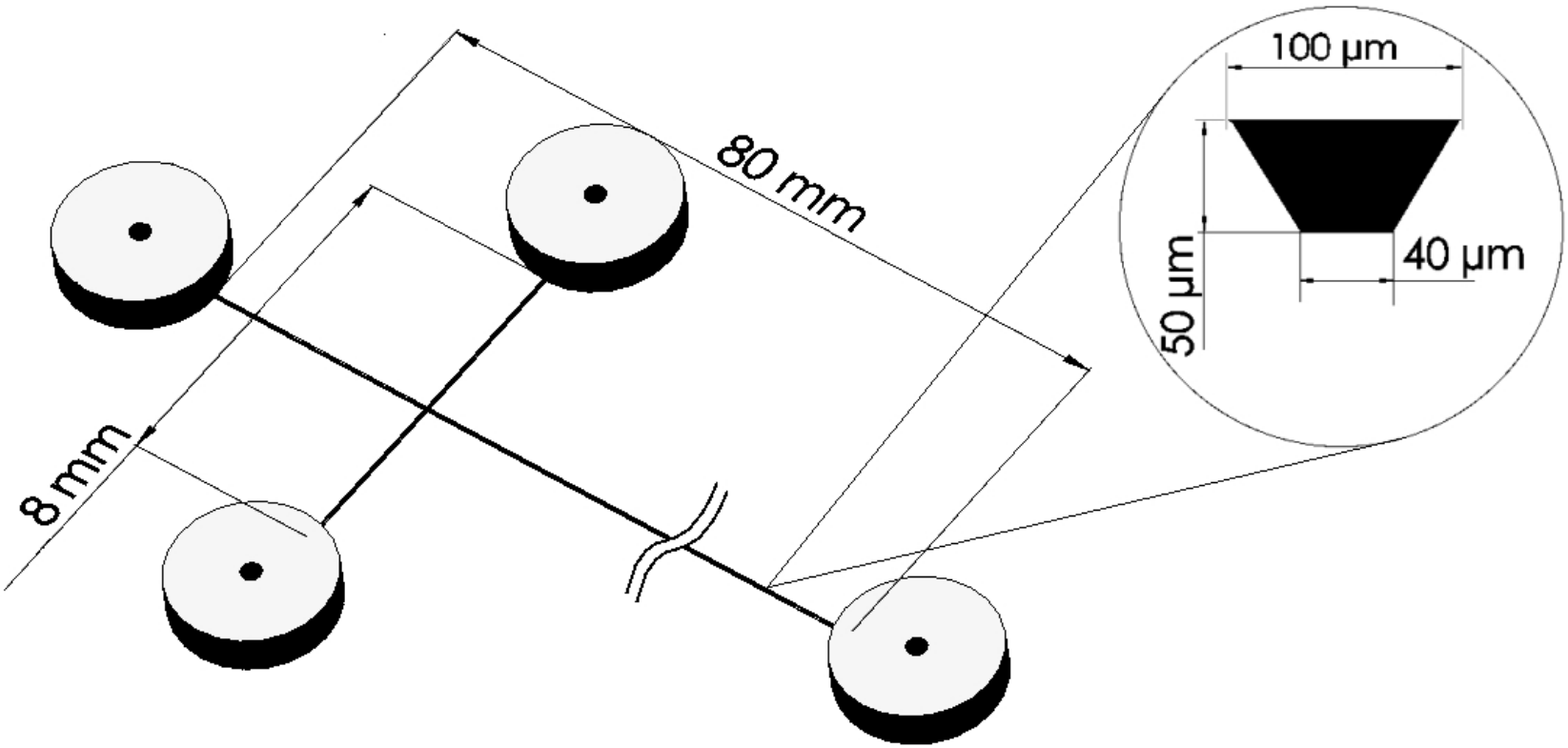
[petsc4py] Interoperability – SWIG

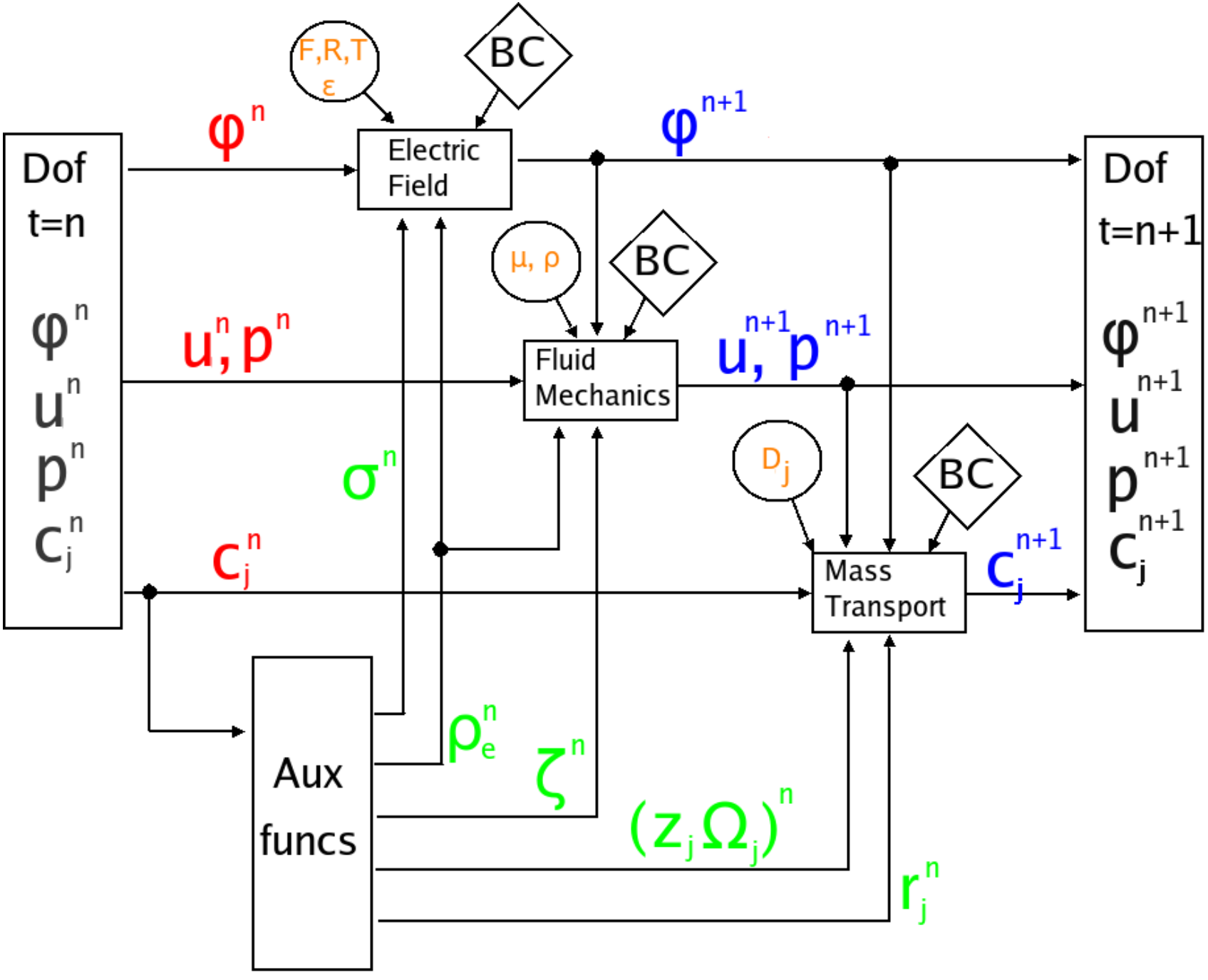
```
from petsc4py import PETSc
import MyApp

comm = PETSc.COMM_WORLD
app = MyApp.Nonlinear(comm, "example.dat")

X = app.crateVec()
F = app.crateVec()
J = app.crateMat()

snes = PETSc.SNES().create(comm)
snes.setFunction(app.evalFunction, F)
snes.setFunction(app.evalJacobian, J)

snes.setFromOptions()
snes.solve(None, X)
```

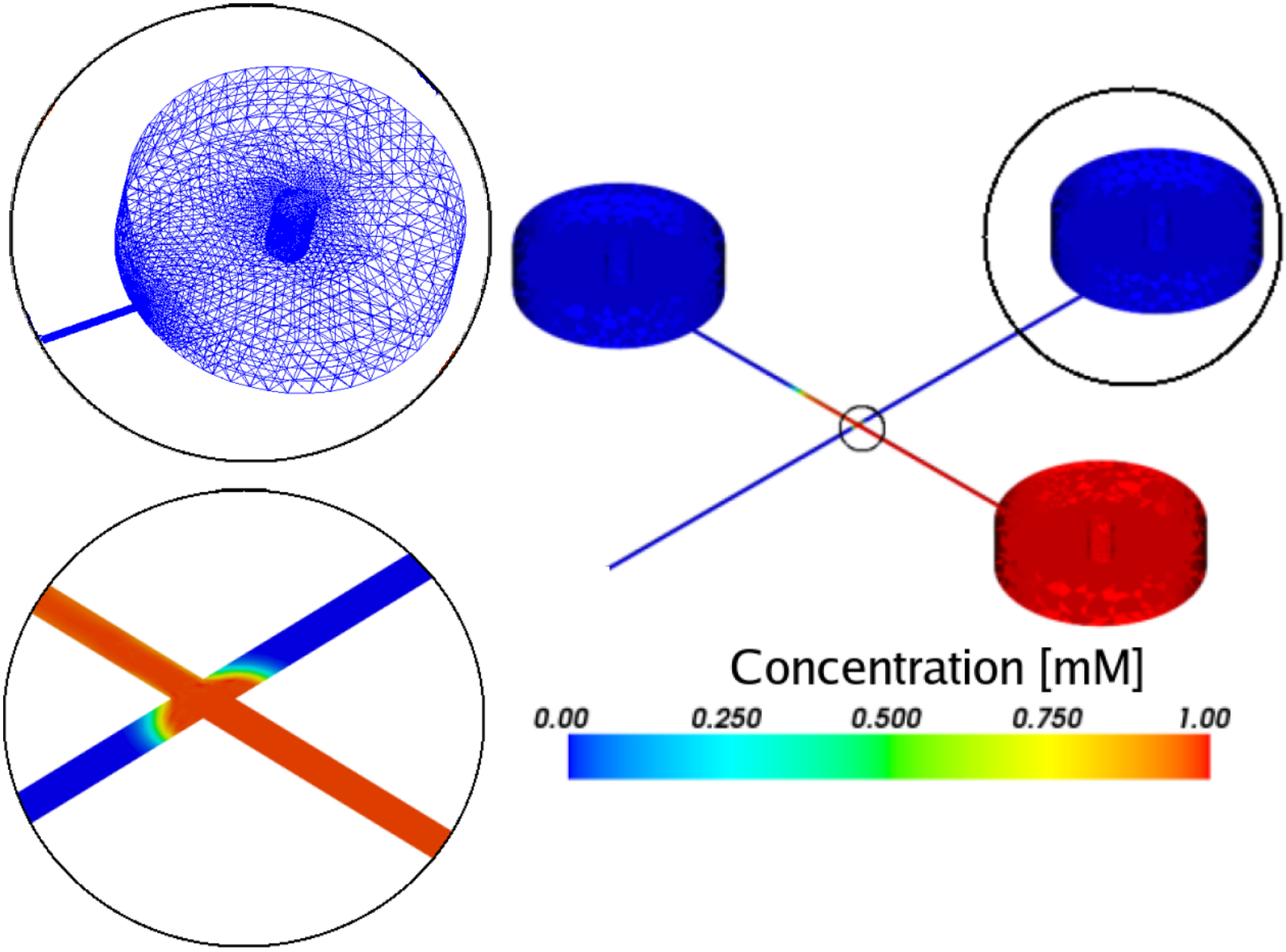
Microfluidics (μ-TAS)

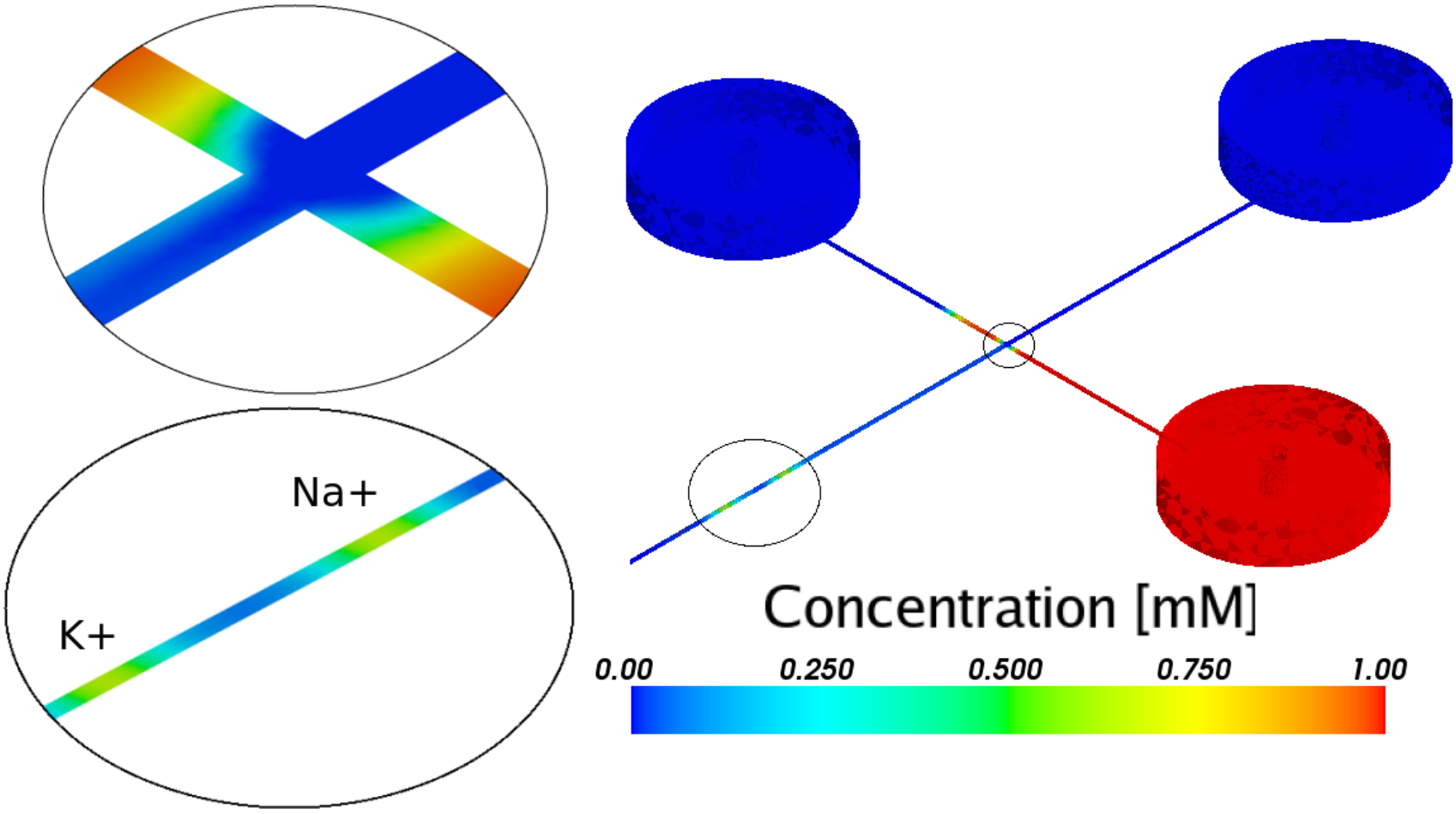


Microfluidics (μ-TAS)




F,R,T
ε
BC
Dof
t=n
φ^n
Electric
Field
φ^{n+1}
Dof
t=n+1
μ, ρ
BC
u^n, p^n
Fluid
Mechanics
u^{n+1}, p^{n+1}
φ^n
u^n
p^n
c_j^n
σ^n
D_j
BC
c_j^n
Mass
Transport
c_j^{n+1}
φ^{n+1}
u^{n+1}
p^{n+1}
c_j^{n+1}
Aux
funcs
ρ_e^n
ζ^n
$(z_j \Omega_j)^n$
r_j^n


Concentration [mM]
0.00
0.250
0.500
0.750
1.00


Na+
K+
Concentration [mM]
0.00
0.250
0.500
0.750
1.00

mpi4py

- Development: `http://mpi4py.googlecode.com`
- Mailing List: `mpi4py@googlegroups.com`
- Chat: `dalcinl@gmail.com`

petsc4py

- Development: `http://petsc4py.googlecode.com`
- Mailing List: `petsc-users@mcs.anl.gov`
- Chat: `dalcinl@gmail.com`

Thanks!