MAX-PLANCK-INSTITUT
FÜR PLASMAPHYSIK

Psydac: a Python IGA library for large-scale simulations

Ease of use and high performance in the open-source Python ecosystem

SFSCON 2023, Bozen, 10-11 November 2023

Yaman Güçlü[1]

[1]Numerical Methods in Plasma Physics Division, Max Planck Institute for Plasma Physics, Garching, Germany

The impossible task of Scientific Computing

Prototyping environment:

- User-friendly and interactive
- Extensive numerical libraries
- Visualization tools
- Data analysis

High performance computing (HPC):

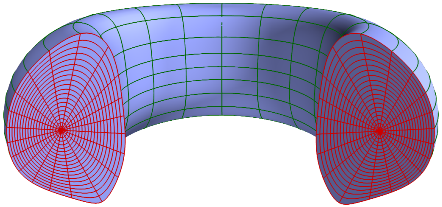
- Single-core optimization
- Shared-memory and MPI parallelization
- Use heterogeneous computing systems
- Strict quality control

How to bridge the gap between prototyping and production?

Strategy:

- Code is written in Python (some parts may be generated automatically)
- MPI parallelization is based on mpi4py library
- Bottlenecks are translated to Fortran/C using the Pyccel transpiler
- If needed, HPC specialists can further optimize the generated code

Genesis of PSYDAC



- Spline FEM widely used in big codes at IPP: SeLaLib, JOREK, ORB5, EUTERPE, GEMPIC, GVEC, ...
  - Can accurately represent complex geometries (IGA = Iso-Geometric Analysis)
  - Stable smooth solutions with high order of accuracy
  - Structure-preserving methods based on de Rham sequence (FEEC = finite element exterior calculus)
- In 2018 we created PSYDAC, an open source IGA library in Python:
  - Fast prototyping environment with high model flexibility
  - Experiment with new distributed data structures (MPI)
  - Achieve performance through C/Fortran code generation

Genesis of PSYDAC

Could we have used or extended another open-source FEM library? Not easily:

	Fenics	Tigar	Firedrake	GeoPDEs	PetIGA	FreeFem++	G+Smo	MFEM
Complex geometries	✓	✓	✓	✓	✗	✓	✓	✓
Spline spaces	✗	✓	✗	✓	✓	✗	✓	–
Support FEEC	✗	✗	✗	✗	–	✗	✗	✗
Python API	✓	✓	✓	✗	✗	✗	✗	✓
UFL (or equivalent)	✓	✓	✓	✗	✗	–	✗	✗
Model flexibility	✓	✓	✓	–	✓	✓	✓	–
HPC capabilities	✗	✗	✗	✗	✓	✗	✗	✓

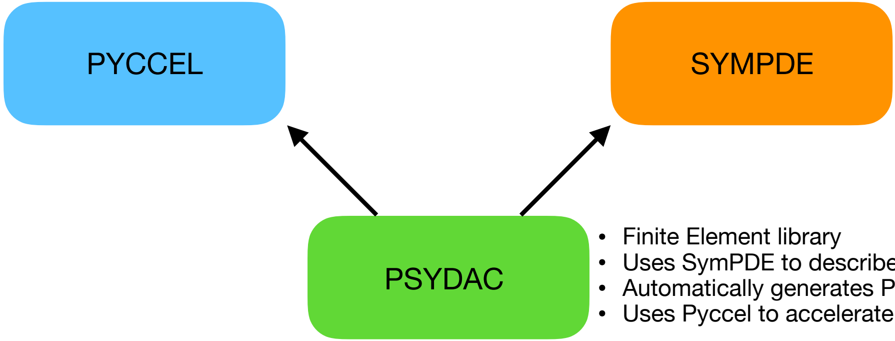
Glossary:

- FEEC = finite element exterior calculus
- UFL = unified form language – a domain specific language for the finite element method
- HPC = high performance computing
- "Model flexibility": ability to define any variational form without changing the library source code

The PSYDAC project

- Converts Python to Fortran
- Compiles Fortran with compiler of choice

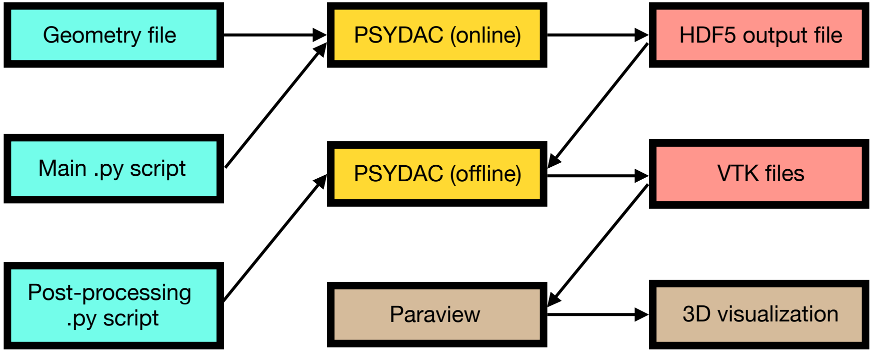
- Extension of SymPy library
- Symbolic description of PDEs
- Provides differential operators, integrals, etc



- Finite Element library
- Uses SymPDE to describe weak formulation
- Automatically generates Python code
- Uses Pyccel to accelerate Python

See `https://github.com/pyccel`: Pyccel + SymPDE + PSYDAC + . . .

Simulation workflow



- Psydac is MPI-parallel both in the online and offline phases
- Paraview installation on cluster is also MPI-parallel

Library architecture

CODE GENERATION

HIGH-LEVEL API

- Discrete Bilinear Forms
- Discrete Linear Forms
- Discrete Scalar Functionals

FEEC API

- De Rham sequences
- Grad, Curl, Div operators
- Projection operators

OUTPUT MANAGER

- Write function spaces to YML file
- Parallel write vectors to HDF5 file
- Generate VTK files

FINITE ELEMENT METHOD

- Tensor-product scalar spaces
- Vector spaces
- Multi-patch (broken) spaces
- Interpolation operators
- Histopolation operators
- Quadrature formulas

LINEAR ALGEBRA

- MPI-distributed vectors & matrices
- Block vectors & matrices
- Abstract linear operators
- Iterative linear solvers

BASIS

- Evaluation of B-splines and NURBS
- Derivatives evaluation
- Knot insertion
- Degree elevation

GEOMETRY

- NURBS mappings
- Inter-patch connectivity
- Boundaries and interfaces
- Load multi-patch geometry files

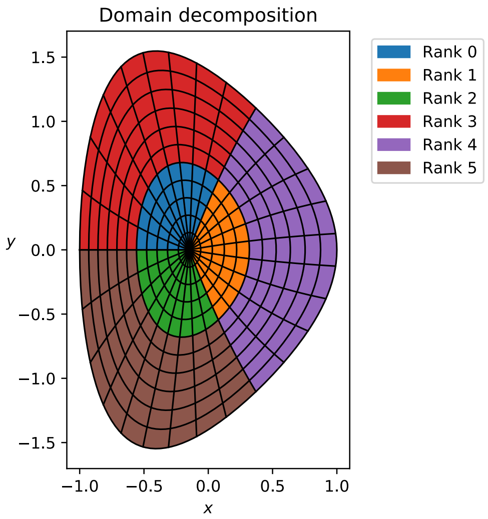
MPI parallelization

Work distribution across MPI processes:

- Decompose logical domain
- Decompose stiffness matrix
- Decompose coefficients vector
- Use iterative linear solvers

Distributed matrix-vector product:

- Each process needs a few vector values from adjacent domains (ghost regions)
- Use MPI communication to update ghost regions



Static compilation of computational kernels

Python kernels translated to Fortran (or C) using Pyccel:

- Evaluation of B-splines, NURBS, and their derivatives
- Matrix-vector (dot) product
- Vector-vector (inner) product
- Matrix transposition
- Matrix conversion to SciPy or PETSc sparse formats
- Assembly of matrices, vectors, and scalar functionals

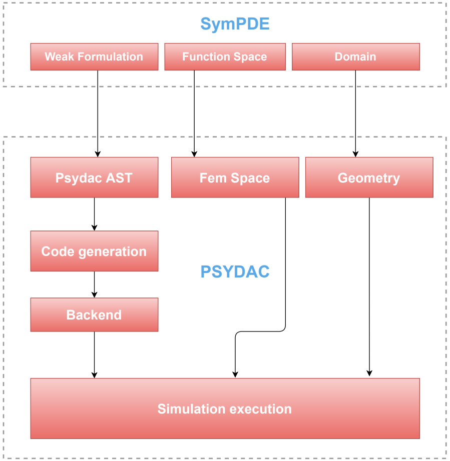
Pyccel supports OpenMP pragmas:

- Corresponding OpenMP pragmas are printed in the Fortran/C code
- Bypass Python's global interpreter lock (GIL) to achieve parallel multi-threading

Automatic code generation

Workflow:

- User describes model equations w/ SymPDE
- User selects analytical geometry or loads geometry file
- PSYDAC creates FEM spaces and MPI decomposition
- PSYDAC generates computational kernels in Python:
  - Assembly of matrices (bilinear forms)
  - Assembly of vectors (linear forms)
  - Assembly of scalar functionals
- Python kernels translated to Fortran using Pyccel
- Linear system solved with iterative method (e.g. PCG)



Usage example: 2D Poisson's equation (1/3)

Continuum model in strong form:

Given $f(x, y)$, find $u(x, y)$ such that

$$\begin{cases} -\nabla^2 u = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega. \end{cases}$$

Continuum model in weak form:

Given $f \in L^2(\Omega)$, find $u \in H_0^1(\Omega)$ s.t.

$$\underbrace{\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega}_{a(u,v)} = \underbrace{\int_\Omega f \, v \, \mathrm{d}\Omega}_{l(v)}$$

$$\forall v \in H_0^1(\Omega)$$

```python
from sympy  import cos, pi
from sympde import *

# Problem definition
OmegaP  = Square('Omega')                          # parametric domain
F       = CollelaMapping2D('F', eps=0.1, k1=1, k2=1)  # mapping
Omega   = F(OmegaP)                                 # physical domain
x, y    = Omega.coordinates                     # physical coordinates
u_ex    = cos(pi/2 * x) * cos(pi/2 * y) # manufactured solution
f       = -laplace(u_ex)                            # right-hand side

# Function space for trial and test functions
V    = ScalarFunctionSpace('V', Omega)
u, v = elements_of(V, 'u, v')

# Declare bilinear and linear forms for variational formulation
a = BilinearForm((u, v), integral(Omega, dot(grad(u), grad(v))))
l =   LinearForm(    v , integral(Omega, f * v))
bc = EssentialBC( u, 0 , Omega.boundary) # boundary conditions

# Variational formulation of Poisson's equation
equation = find(u, forall=v, lhs=a(u, v), rhs=l(v), bc=bc)
```

Intermezzo: SymPDE (1/2)

SymPDE is a symbolic algebra system for weak formulations of partial differential equations (PDEs):

- It extends the famous Python library SymPy
- It is a domain specific language (DSL)
- It provides the tools for processing the expressions in another library (e.g. Psydac)

What can SymPDE do for you?

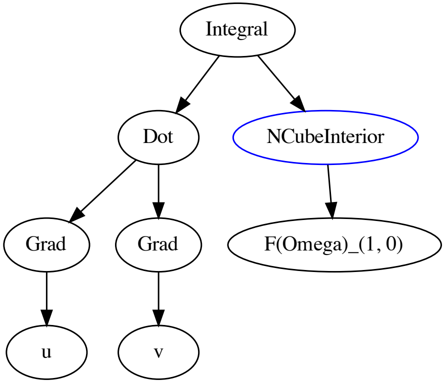
- Provide partial differential operators (gradient, divergence, curl, etc..)
- Satisfy operator identities (e.g. $\nabla \cdot (\nabla \times \mathbf{A}) \equiv 0$)
- Provide integral operators (volume integrals, boundary integrals, etc...)
- Check correctness of data types in a mathematical expression
- Verify linearity of $l(u)$ and bilinearity of $a(u, v)$

Intermezzo: SymPDE (2/2)

SymPDE is a SymPy extension, and each expression tree can be inspected, e.g. with Graphviz:

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega$$

```python
from sympy.printing.dot import dotprint
from pygraphviz         import AGraph

# Inspect bilinear expression a(u,v)
g = AGraph(dotprint(a(u, v)))
g.layout(prog='dot')
g.draw('graph.pdf')
```



Usage example: 2D Poisson's equation (2/3)

Galerkin method:

- Let $u_h \approx u$ in finite dim. subspace $V_h \subset H_0^1$
- Choose basis for $V_h = span(\phi_1, \phi_2, \dots \phi_N)$
- Search for $u_h(x, y) = \sum_k w_k \, \phi_k(x, y)$
- Choose test functions $v = \phi_k$
- Linear PDE yields linear system $A \, w = b$

Given $f \in L^2(\Omega)$, find $w \in \mathbb{R}^N$ s.t.

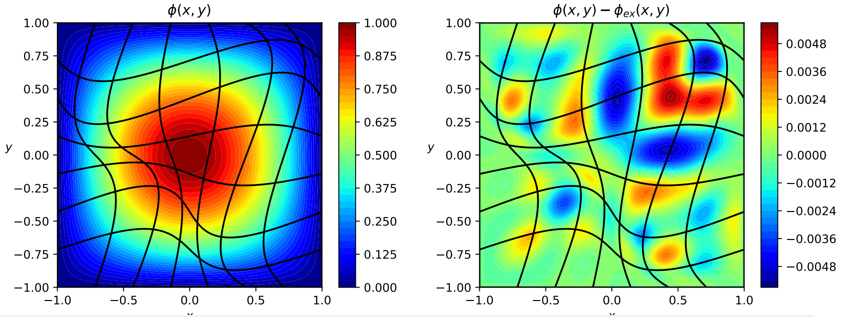
$$\sum_{j=1}^{N} \underbrace{\left(\int_{\Omega_h} \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}\Omega \right)}_{A_{ij}} w_j = \underbrace{\int_{\Omega_h} f \, \phi_i \, \mathrm{d}\Omega}_{b_i},$$

for $i = 1, \dots, N$.

```python
from mpi4py import MPI
from psydac import discretize

# Select MPI communicator (set to None for serial code)
comm = MPI.COMM_WORLD

# Create computational domain
Omega_h = discretize(Omega, ncells=(7, 7), comm=comm)

# Create finite element space (B-splines)
V_h = discretize(V, Omega_h, degree=(3, 3))

# Set up linear system: code generation
equation_h = discretize(equation, Omega_h, [Vh, Vh])

# Compute numerical solution:
#   1. assemble (distributed) sparse matrix A
#   2. assemble (distributed) dense  vector b
#   3. solve linear system A w = b
#   4. create callable field u_h(x,y)
u_h = equation_h.solve()
```

Usage example: 2D Poisson's equation (3/3)

Domain: 7×7 cells
Spline degree: $p = 3$
- Left: numerical solution
- Right: numerical error



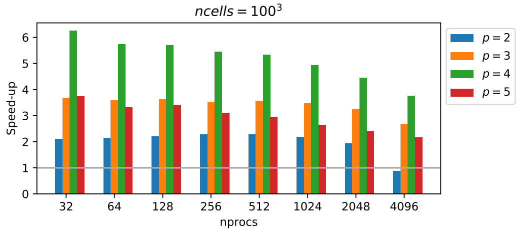
```
# Scalar error estimate (L2 norm)
l2norm   = Norm(u - u_ex, Omega, kind='l2')  # symbolic definition - SymPDE
l2norm_h = discretize(h2_norm, Omega_h, Vh)  # code generation - PSYDAC
l2_error = l2norm_h.assemble(u=u_h)          # computation
```

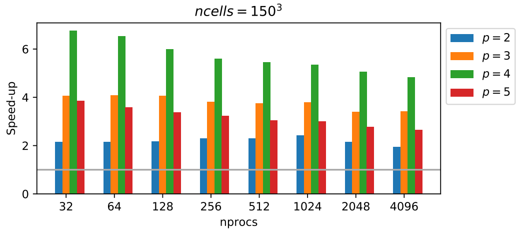
Running the whole thing in IPython:

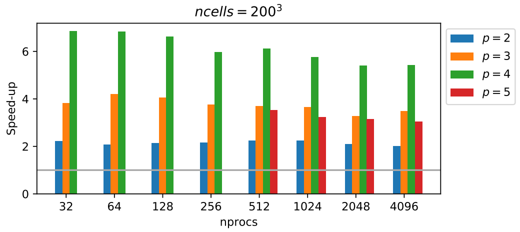
```
In [1]: run poisson_2d.py
In [2]: print(l2_error)
Out[2]: 5.17e-03
```

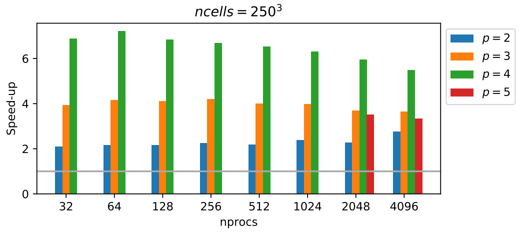
Parallel Performance

Assembly of stiffness matrix for 3D Poisson problem. Speedup compared to C library PetIGA (grey line):



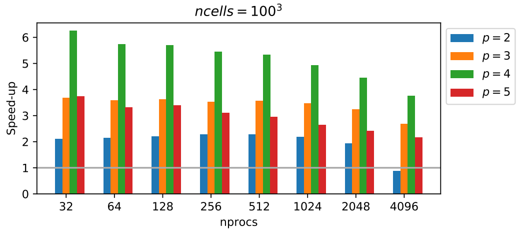


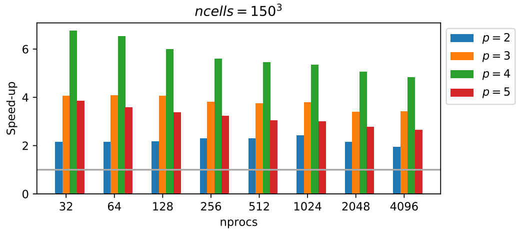


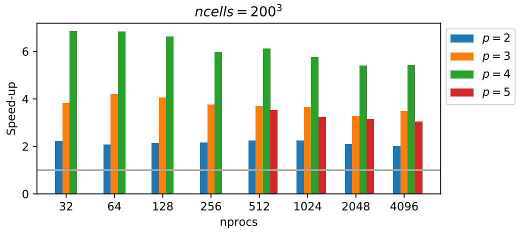


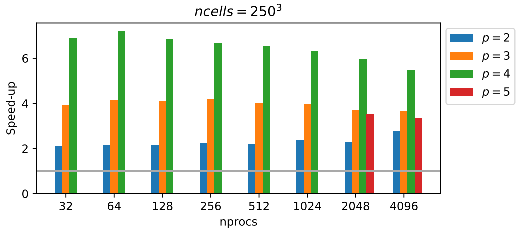
Parallel Performance

Matrix-vector product for 3D Poisson problem. Speedup compared to C library PetIGA (grey line):









Conclusions

We can bridge the gap from prototype to production in scientific computing:

- Use Python to experiment low-level algorithms and high-level library design
- Achieve flexibility through: domain specific languages, semantic parsers, code generation
- Use Python accelerators (e.g. Pyccel) to get single-process performance:
  - Statically compile the most computationally-intensive functions
  - Achieve parallel multithreading with OpenMP (circumventing Python's GIL)
  - Future work: GPU offloading (with CUDA/HIP or OpenMP/OpenACC)
- Use MPI parallelization (through the mpi4py library) to distribute work on large machines

THANK YOU FOR YOUR ATTENTION!

Psydac's repository: `https://github.com/pyccel/psydac`