

# High-level programming with Python and BSP

Åsmund Ødegård, Ola Skavhaug, Hans Petter Langtangen  
Konrad Hinsén

Simula Research Laboratory and University of Oslo, Norway  
Centre de Biophysique Moléculaire (CNRS), France

# Outline

- Scientists on the move
- Environments
- Language of our choice: Python
- Interface or Extend! Design issues
- Example: Solving a PDE in Parallel with Python and BSP

# On the Move

- Scientists are on the move:
  - From compiled languages (Fortran, C/C++)
  - To Problem Solving Environments (PSE) like Matlab and high-level languages
- Why? Matlab is easier to use and feels more productive
- We can extend the “Matlab way of working” far beyond Matlab

# On the Move - Why?

PDE Software is much more than number crunching:

- simulation, data analysis, visualization procedures
- set-up of comprehensive numerical experiments
- reporting results of such experiments
- file/data format conversion
- creating user-friendly tools for others
- ...

A PSE or some high-level language can do these tasks better than Fortran, C/C++

We still have to do the number crunching

# Environments

- We focus on three different environments
- Compiled low-level languages, Fortran and C/C++
- Apply a PSE, Matlab
- High-level language, Python

# Characteristics of Fortran/C/C++

- Very efficient and fast
- Possible to optimize with respect to hardware
- Lots of excellent numerical libraries
- Tedious to program, error-prone
- Attention drawn away from design issues to languages issues
- Lack features for all but number-crunching

# Characteristics of Matlab

- Simple and intuitive syntax
- High level commands
- Interactive and programmable
- Heavy numerics take place in optimized F77 code
- Many useful toolboxes
- Gluing of computing and visualization
- Easy to add GUIs

# Observations

- Scientific programming is both about complexity and performance
- Low-level language do not deal well with complexity
- High-level languages do!
- Utilites written in low-level language may be interfaced from high-level languages
- Matlab is mainly powerful for linear algebra problems
- High-level languages offer the desirable characteristics of Matlab, and much more!

# Language of Choice

- We choose Python because of its
  - simple and clean syntax
  - flexible support for object-oriented programming
  - rich set of available modules and utilities
  - Interface libraries: SWIG for C/C++, F2py for F77

# Features of Python

- Made for convenient programming (not for efficient code generation by a compiler)
- Interactive, interpreted (no compilation/linking)
- Easy-to-use high-level data types, e.g., nested, heterogeneous list and hash structures
- Automatic memory management
- Extensive run-time testing and clear error messages
- No declaration of variables or function arguments (C++ templates for “free”)
- Run-time code generation is possible

# More Features of Python

- Programs are much shorter than in Fortran, C/C++ and Java
- Extensive network/Internet programming support
- Extensive text processing support
- Rapid development of graphical user interfaces
- Wide file handling functionality
- Numerical Python: fast array operations
- ScientificPython: tools for parallel programming
- Fast program–test–debug–modify cycle

# Interface or Extend!

Two quite different approach:

- Interface an existing application
  - SWIG, F2py, by hand
  - “Drive” the simulation from Python
  - Easy to realize, but limited use, majority of the code still written in a low–level language
- Design your application in Python
  - Abstractions close to scientific concepts
  - Compact and more readable programs
  - General–purpose libraries can still be integrated
  - Phase 2: Reimplement time–critical parts in F77/C/C++

# Example: Solving a PDE in Parallel

- We want to design a parallel solver in Python
- Use Numerical Python (NumPy) for high-performance numerics
- Use the Python-BSP module included in ScientificPython for parallel programming
- Python-BSP is an interface to a BSP-library (Bulk Synchronous Parallel), BSPLib (Oxford) or PUB (Paderborn Uni.)

# Python-BSP Highlights

- Simpler model than MPI
- Two different scopes: local and global objects
- Global objects exist on the “parallel machine”
- Data in global object are evenly distributed among the processes (possibly under user–control)
- Transfer complex data types between processes
- Deadlocks impossible (compare MPI...)
- Abstraction can make parallel part almost invisible in application:  $x = A*b$
- Done with operator overloading

# Model Problem: Option Pricing

$$\frac{\partial P}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 P}{\partial S^2} + rS \frac{\partial P}{\partial S} - rP = 0, \quad (1)$$

$$P(S, T) = \max(E - S, 0),$$

$$P(0, t) = Ee^{-r(T-t)},$$

$$P(S, t) = 0 \quad \text{as } S \rightarrow \infty.$$

- We solve the Black–Scholes equation for the so–called European vanilla options with one underlying asset, a backward parabolic PDE
- Discretization: Three–point finite difference method in the spatial domain, and a backward difference in time
- Result in a linear, three–diagonal system at each time–step

# Experiments

- Linear system solved with Jacobi's method
- Simple, old-fashioned, but think e.g. smoother for a multigrid method
- More underlying assets will give a multidimensional problem, and a sparse matrix
- Only NumPy arrays are used
- Distributed vectors and tri-diagonal matrix is implemented to have operators without auxiliary storage

# Code: Simulator

```
# Create global classes
gDistTriDiagMatrix = ParClass(DistTriDiagMatrix)
gDistVector = ParClass(DistVector)

# Create global objects, using global classes
A = gDistTriDiagMatrix(_l, _d, _u)      # Fill global matrix
x = gDistVector(_x)                    # and vec (using final condition)
Dinv = gDistVector(_Dinv);
e = ParConstant(_e)

# Main program, solve problem
for n in xrange(1, N+2):
    b=x.copy()                          # Update right hand side
    b[0] = E*exp(-r*n*dt); b[-1] = .0;  # Boundary conditions
    x = Jacobi(A, x, b, Dinv, e, maxIter)
```

# Code: Jacobi's Method

```
# Jacobi's method in matrix-vector form
# The vectors and matrix can be either parallel or scalar

def Jacobi(A, x, b, Dinv, e, max_iter = 500):
    r = A*x; r -= b;
    i = 0
    x_norm = x.norm()
    if x_norm < e: tolerance = e           # use an absolute conv. cri
    else: tolerance = e*x_norm           # use a relative conv. crit
    while (r.norm() > tolerance) & (i < max_iter):
        i += 1
        r *= Dinv; x -= r;
        r = A*x; r -= b;
    return x
```

# Experiments

- Bottleneck: matrix–vector product
- Re–implement in F77/C, two options:
  - Write a simple function in language of choice, interface with SWIG, F2py
  - Write it by hand to gain full control
- We have used the latter approach

# Experiments

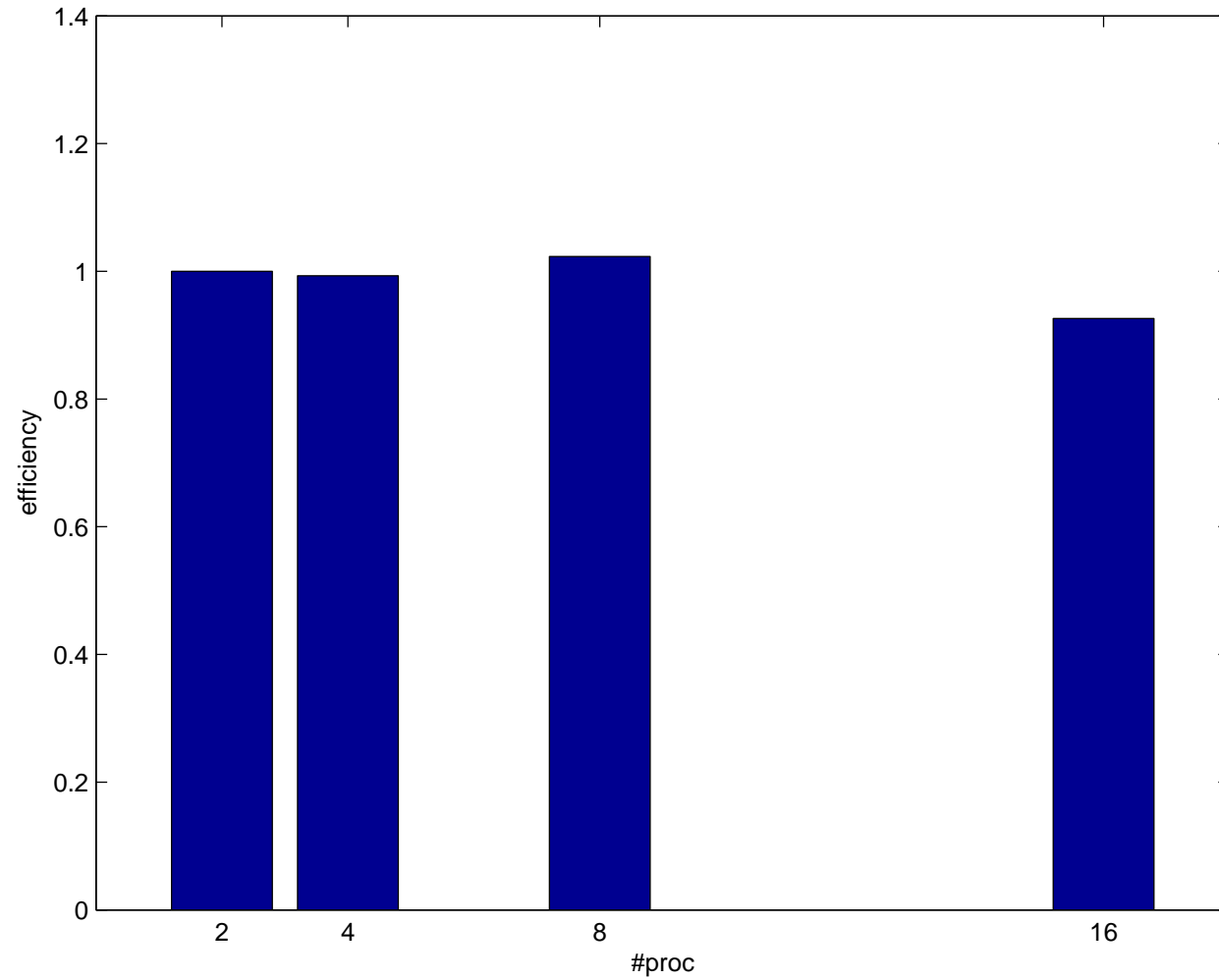
- For comparison, we have implemented several versions of the same solver:
  - Pure Python version
  - Python with handwritten C extension
  - F77 and C versions
  - Matlab
  - Parallel C/MPI version
- The version with a C extension is an example of a mixed–language implementation
- We use 2 timesteps and 500 Jacobi iterations in all experiments

# Experiments

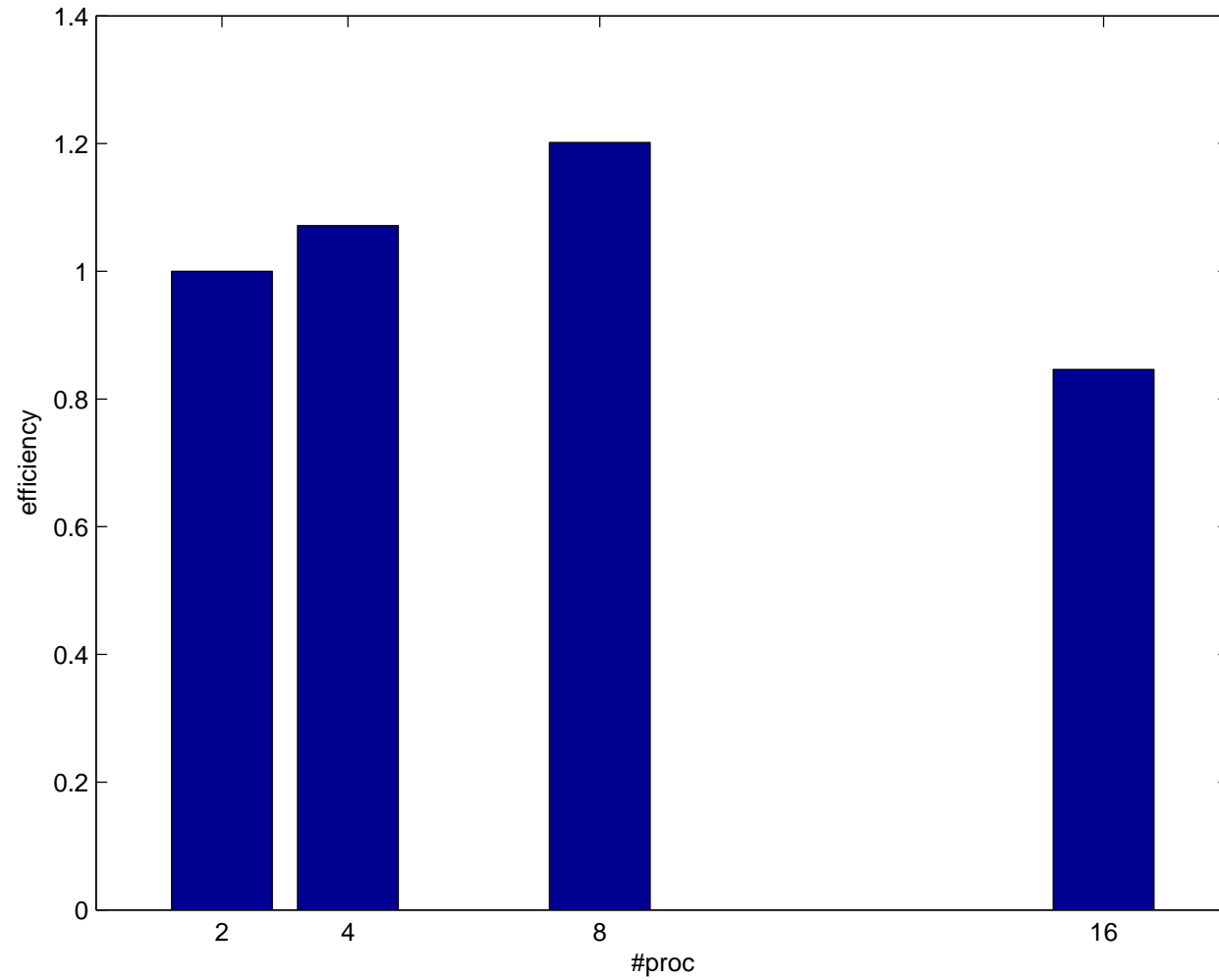
Problem size	$2^6 \cdot 10^3$	$2^{10} \cdot 10^3$	$2^{12} \cdot 10^3$	$2^{14} \cdot 10^3$
C	9.90	148.9	586.1	2394.6
Fortran	9.31	148.8	542.6	N/A
Matlab	25.25	338.34	1347.2	
Pure Python, one CPU	68.78	850.38	3240.4	
Pure Python, two CPUs	50.28	545.0	2095.0	
Python one CPU	29.71	273.5	1072.4	4439.45
Python, two CPUs	24.71	183.90	686.36	3442.31

Table 1: Total run time for the simulations on a Dual Athlon-MP2000

# Efficiency: Python-BSP



# Efficiency: C-MPI



# Some Concluding Remarks

- Implementation in high-level languages can solve some real problems
- Designing in such a language free your mind!
- For simple problems, we can achieve decent performance, also utilizing parallel machines
- High-level parallel implementation with Python-BSP is much easier than corresponding C-MPI implementation
- We should investigate harder parallel programs further to get a better impression of the possibilities