Python: The Next Supercomputing Language

Cameron Sparr
Intern, NCAR SIParCS program
Summer 2012

Mentor:
Davide Del Vento
NCAR, Boulder, CO
Roadmap

- Why Python?
- The PyPy Project
  - Previous attempts at higher-performance Python
- Python performance techniques
- Vectorization and NumPy
- Converting code from Fortran > Python
  - Conversion challenges
- Code that I converted over the summer
- Results of code conversions
- Future work
Why Python?

• Python is a high-level programming language that is designed to emphasize code readability and very clear syntax.
• Python forces the programmer to write code that is indented and readable.
• Large, active, strong user community.
• And, interactive….
Interactive!!
The PyPy Project

- PyPy is an alternative implementation of the Python Language (CPython is the standard implementation).

- The name comes from it being a Python language interpreter written in Python…
The PyPy project refers to two things:

1. RPython – a restricted subset of Python that is used for writing Virtual Machines (language interpreters). RPython can be used to write a VM for ANY language (not just Python). This is mostly of interest to programming language designers.

2. PyPy interpreter – PyPy is the Python VM written with RPython, which is what my project is primarily concerned with.
Benefits of PyPy

• If you write a VM in RPython, and pass it through the PyPy translator toolchain, you get a Just-In-Time compiler (JIT) “for free”
JIT?

- Just-In-Time Compiler
- A JIT works behind-the-scenes to find loops that may be converted to machine code. When a loop is detected, a marker is left, and the loop is converted to machine code. All subsequent executions of the loop will then call the machine code version.
PyPy vs. CPython

• PyPy is much faster than CPython when working with computationally intense code:

```python
import numpy as np

def run(n):
    a = np.zeros((n,n))
    for i in xrange(n):
        for j in xrange(n):
            a[i,j] = (a[i,j] + 1. * np.pi) / 2
```
PyPy vs. CPython

\[ n = 5000 \]

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Time (s)</th>
<th>Speed x</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython 2.7.1</td>
<td>53.068</td>
<td>1x</td>
</tr>
<tr>
<td>PyPy 1.9.0</td>
<td>1.62</td>
<td>33x</td>
</tr>
</tbody>
</table>

PyPy is \textbf{33x} faster in this example
import numpy as np

def run(n):
    a = np.zeros((n,n))
    for i in xrange(n):
        for j in xrange(n):
            a[i,j] = (a[i,j] + 1. * np.pi) / 2

import numpy as np

def run(n):
    a = np.zeros((n,n))
a[::,::] = (a[::,::] + 1. * np.pi) / 2
## Example Results

\[ N = 5000 \]

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Time (s)</th>
<th>Speed x</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython 2.7.1, vectorized</td>
<td>0.357</td>
<td>1x</td>
</tr>
<tr>
<td>PyPy 1.9.0, vectorized</td>
<td>0.357</td>
<td>1x</td>
</tr>
<tr>
<td>PyPy 1.9.0, not vectorized</td>
<td>1.62</td>
<td>4.54x</td>
</tr>
<tr>
<td>CPython 2.7.1, not vectorized</td>
<td>53.068</td>
<td>148.65x</td>
</tr>
</tbody>
</table>
Jacobi Algorithm code:

• Numerical solution of the 2d heat equation using the Jacobi method, an iterative algorithm for solving systems of linear equations.
The Algorithm...

\[ Ax = b \]

\[ A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \]

\[ x_{i}^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n. \]
Fortran to Python conversion:

do iter=1,maxiter
    uold = u  ! old values
    dumax = 0.d0
    do j=1,n
        do i=1,n
            u(i,j) = 0.25d0*(uold(i-1,j) + uold(i+1,j) + uold(i,j-1) + uold(i,j+1) + h**2*f(i,j))
            dumax = max(dumax, abs(u(i,j)-uold(i,j)))
        enddo
    enddo
enddo

for iter in xrange(1,maxiter+1):
    uold = np.zeros(u.shape)
    uold[:, :] = u
    dumax = 0.0

    u[1:n+1, 1:n+1] = 0.25 * (uold[0:n, 1:n+1] + uold[2:n+2, 1:n+1] \ 
                              + uold[1:n+1, 0:n] + uold[1:n+1, 2:n+2])
    dumax = np.max(np.abs(u-uold))
## Jacobi Results

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Time (s)</th>
<th>Speed x</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortran, -O3 optimizations</td>
<td>237.68</td>
<td>1x</td>
</tr>
<tr>
<td>PyPy 1.9.0, vectorized</td>
<td>373.11</td>
<td>1.56x</td>
</tr>
<tr>
<td>PyPy 1.9.0, not vectorized</td>
<td>1001.74</td>
<td>4.2x</td>
</tr>
<tr>
<td>gfortran, no optimizations</td>
<td>2532.87</td>
<td>10.65x</td>
</tr>
</tbody>
</table>
Jacobi Results

- gfortran -O3: 1x
- PyPy, vectorized: 1.56x
- PyPy, not vectorized: 4.2x
- gfortran -00: 10.65x
Eulag is a EULerian-LAGrangian numerical solver for geophysical flows written in Fortran 77

- Attempted to convert “baby” version to Python
Challenges

• Fortran 77 eccentricities:
  – Common blocks
  – Generous use of MAX function across large matrices
  – Other deprecated functions
Future Work

• Pick a model written in Fortran 90 or newer
• Maybe QTCM1?
  – Written in fairly modern Fortran
  – Already been wrapped in Python using f2py, would provide an interesting comparison
  – Function-based
Conclusions

• Not worth converting existing models from Fortran/C to PyPy

• Cons of using PyPy vs. fortran/C:
  – Speed (this is improving)

• Pros of using PyPy vs. fortran/C:
  – Interactive
  – Faster development
Acknowledgements:

Thanks to:

• **Davide Del Vento**, for his mentorship.
• **Dan Nagle**, for his help with Fortran.
• **Randall LeVeque**, for allowing me to use the Jacobi iteration code he wrote in Fortran.