Python in High performance computing

Jussi Enkovaara
Outline

• Why Python?
• High performance issues
• Python challenges
• Case study: GPAW
Why Python?
What is Python?

• Modern, interpreted, object-oriented, full featured high level programming language

• Portable (Unix/Linux, Mac OS X, Windows)

• Open source, intellectual property rights held by the Python Software Foundation

• Python versions: 2.x and 3.x
  – 3.x is not backwards compatible with 2.x
Why Python?

- Fast program development
- Simple syntax
- Easy to write well readable code
- Large standard library
- Lots of third party libraries
  - Numpy, Scipy
  - Mpi4py
  - ...
Data types

- **Integers**

- **Floats**

- **Complex numbers**

- **Basic operations**
  - +, -, *, / and **

- **Strings are enclosed by “” or ’’**
  - + and * operators

```python
x = 2
x = 3.0
x = 4.0 + 5.0j
s1 = "very simple string"
s2 = 'same simple string'
s3 = "this isn't so simple"
s4 = 'is this “complex”'

>>> "Strings can be " + "combined"
'Strings can be combined'
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!'
```
Data types

- Python is dynamically typed language
  - no type declarations for variables
- Variable does have a type
  - incompatible types cannot be combined

```python
print "Starting example"
x = 1.0
for i in range(10):
    x += 1
y = 4 * x
s = "Result"
z = s + y  # Error
```
Dynamic typing

- No separate functions for different datatypes

```python
def add(x, y):
    result = x + y
    return result
```

- Works for any numeric type
  - No duplicate code e.g. for real and complex numbers
Powerful data structures: List

- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place of the list
List example

- Simple C-code

```c
#include <stdio.h>
#include <stdlib.h>

int comp(const void * a, const void * b)
{
    const int *ia = (const int*)a;
    const int *ib = (const int*)b;
    return *ia - *ib;
}

int main(int argc, char **argv)
{
    int* array;
    int i;
    array = (int*) malloc(3*sizeof(int));
    array[0] = 4;
    array[1] = 2;
    array[2] = 6;

    int* array2;
    array2 = (int*) malloc(4*sizeof(int));
    for (i=0; i < 3; i++)
        array2[i] = array[i];
    array2[3] = 1;
    free(array);
    array = array2;

    ...
    printf("Before sorting\n");
    for (i=0; i < 4; i++)
        printf("%d ", array[i]);
    printf("\n");
    qsort(array, 4, sizeof(int), comp);
    printf("After sorting\n");
    for (i=0; i < 4; i++)
        printf("%d ", array[i]);
    printf("\n");
}
```

...
List example

• Same in Python

```python
array = [4, 2, 6]
array.append(1)
print "Before sorting", array
array.sort()
print "After sorting", array
```
Powerful data structures: Dictionary

- Dictionaries are associative arrays
- Unordered list of key - value pairs
- Values are indexed by keys
- Keys can be strings or numbers
- Value can be any Python object
Dictionary example

• Data for chemical elements

```python
... atomic_data['H'] = data1
atomic_data['Li'] = data2
...

data = atomic_data['Fe']
name = data['name']
Z = data['atomic number']
density = data['density']
```
Summary

- Python can increase the performance of programmer drastically
- Powerful data structures
- Object-orientation
- Simple text processing and I/O
- Dynamic typing
  - can also be source of errors
Numpy
Numpy – fast array interface

• Standard Python is not well suitable for numerical computations
  – lists are very flexible but also slow to process in numerical computations

• Numpy adds a new array data type
  – static, multidimensional
  – fast processing of arrays
  – some linear algebra, random numbers
Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed
Array operations

• Most operations for numpy arrays are done element-wise
  – +, -, *, /, **

• Numpy has special functions which can work with array arguments
  – sin, cos, exp, sqrt, log, ...

• Operations are carried out in compiled code
  – e.g. loops in C-level

• Performance closer to C than “pure” Python
Linear algebra

- Numpy has routines for basic linear algebra
  - Numpy can be linked to optimized BLAS/LAPACK

- Performance in matrix multiplication
  - \( C = A \times B \)
  - matrix dimension 200
  - pure python: 5.30 s
  - naive C: 0.09 s
  - numpy.dot: 0.01 s
Summary

• Numpy provides a static array data structure
• Multidimensional arrays
• Fast mathematical operations for arrays
• Tools for linear algebra and random numbers
C - extensions
C - extensions

• Some times there are time critical parts of code which would benefit from compiled language

• It is relatively straightforward to create a Python interface to C-functions

• Some tools can simplify the interfacing
  – SWIG
  – Cython, pyrex
Passing a Numpy array to C

**Python**

```python
import myext

a = np.array(...)
myext.myfunc(a)
```

**C: myext.c**

```c
#include <Python.h>
#define NO_IMPORT_ARRAY
#include <numpy/arrayobject.h>

PyObject* my_C_func(PyObject *self, PyObject *args)
{
    PyArrayObject* a;
    if (!PyArg_ParseTuple(args, "O", &a))
        if (!PyArg_ParseTuple(args, "O", &a))
            return NULL;
    return NULL;
    ...  
}
```
Accessing array data

• myext.c

```c
...
PyArrayObject* a;
int size = PyArray_SIZE(a);
double *data = (double *) a->data;
for (int i=0; i < size; i++)
{
    /* Process data */
}
Py_RETURN_NONE;
```
Defining the Python interface

- myext.c

```c
static PyMethodDef functions[] = {
   {"myfunc", my_C_func, METH_VARARGS, 0},
   {0, 0, 0, 0}
};

PyMODINIT_FUNC initmyext(void)
{
   (void) Py_InitModule("myext", functions);
}
```

- Build as a shared library

```
gcc -shared -o myext.so -I/usr/include/python2.6 -fPIC myext.c
```

- Use in Python script

```python
import myext

a = np.array(...)
myext.myfunc(a)
```
Mpi4py
Mpi4py

- Mpi4py provides Python interface to MPI
- Object-oriented interface similar to standard C++
- Communication of arbitrary (serializable) Python objects
- Communication of contiguous NumPy arrays at nearly C-speed
Simple examples

• Parallel “hello”, no communication

```python
from mpi4py import MPI

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

print "I am rank", rank
```

• Communicating Python objects (pickle under hood)

```python
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

Extra material
Simple examples

• Numpy arrays (nearly C speed)

```python
from mpi4py import MPI
import numpy

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

if rank == 0:
    data = numpy.arange(100, dtype=numpy.float)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float)
    comm.Recv(data, source=0, tag=13)
```

• Note the difference between upper/lower case!
  – send/recv: general Python objects, slow
  – Send/Recv: continuous arrays, fast

Extra material
Python challenges
Python initialization

- **import** statements in Python trigger lots of small-file I/O
- In parallel calculations all processes perform the same I/O
- Introduces severe bottleneck with large number (> 512) of processes
- In Blue Gene P, importing NumPy + application specific modules with ~32 000 processes can take **45 minutes**!
Python initialization

• In Blue Gene P, install Python modules to ramdisk
• In Cray, create special Python interpreter
  – Single process does I/O, data broadcast to others with MPI
Global interpreter lock

• There is threading support in Python level
• Global interpreter lock in (CPython) interpreter:
  – Only single thread is executed at time
• Threading has to be implemented in C-extensions
  – Higher granularity than algorithmically necessary
Case study: GPAW
GPAW

• Software package for electronic structure simulations in atomic scale nanostructures
• Implemented in combination of Python and C
• Massively parallelized
• Open source under GPL
• 20-30 developers in Denmark, Finland, Sweden, Germany, UK, US


wiki.fysik.dtu.dk/gpaw
GPAW developers
Python + C implementation

**Lines of code:**
- **Python**
  - Fast development
  - Slow execution
  - High level algorithms
- **C**

**Execution time:**
- **C**
  - Fast execution
  - Slow development
  - Main numerical kernels

BLAS, LAPACK, MPI, NumPy
Python + C implementation

Time line of GPAW's codebase
Parallelization in GPAW

- Message passing with MPI
- Custom Python interface to MPI
- MPI calls both from Python and from C

```python
# MPI calls within the apply C-function
hamiltonian.apply(psi, hpsi)
# Python interface to MPI_Reduce
norm = gd.comm.sum(np.vdot(psi, psi))
```

- All the normal parallel programming concerns
Parallel scalability

- **Ground state DFT**
  - 561 Au atom cluster
  - ~6200 electronic states
  - Blue Gene P, Argonne

- **TD-DFT**
  - 702 Si atom cluster
  - ~2800 electronic states
  - Cray XT5 Jaguar, Oak Ridge
Summary

- Python can be used in massively parallel high performance computing
- Combining Python with C one gets best of both worlds
  - High performance for programmer
  - High performance execution
- GPAW: ~25% of peak performance with 2048 cores