Konrad HINSEN

Centre de Biophysique Moléculaire (Orléans)

AND

Synchrotron Soleil (St Aubin)

Defining the problem

~~Writing code that runs fast~~

Writing ***correct*** code that runs fast

Don’t optimize code that has no unit tests !!!

Defining the problem

Efficient code =

efficient algorithms + efficient implementations

big pay-off
mostly risk-free
domain-specific

variable pay-off
major risk of introducing bug
language and platform-specific

Profiling

Principles of Profiling

Observe the behaviour of a program while it is running:

- Measure execution time per function
- Count how often a function is called
- Follow memory allocation and deallocation

Profiling steps

1) Run the program under profiler control
   - Execution statistics are collected
   - Program is slowed down!

2) Analyze the statistics
   - Identify the functions that use most of the CPU time
   - Check memory allocations
   - ...

Using Python's cProfile module

- Basic use: `python -m cProfile my_script.py`

- Keeping the execution statistics in a file for later analysis:

```
python -m cProfile -o my_script.profile my_script.py
```

- Profiling part of a program:

```
import cProfile
cProfile.run("my_function()")
```

- Inspecting the statistics:

```
import pstats
p = pstats.Stats("my_script.profile")
p.print_stats("time")
```

How it works:

- Modifies the interpreter to call a bookkeeping routine when a function is called and when it returns.

- Use this to measure the execution time of each function.

Optimzing with Cython

Example: Python

```
def exp(x, terms = 50):
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Note: This is not the best algorithm for calculating an exponential function!

Example: Cython

Automatic conversion Python->C

```
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
```

Declaration of C variables

```
    sum = 0.
    power = 1.
    fact = 1.
```

Conversion to integer loop

```
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Loop in C

Automatic conversion C->Python

Performance

50 000 exponential calculations on my laptop:

Python: 0.455 s

Cython: 0.018 s

math.exp: 0.007 s

math.exp uses a better algorithm than our Cython function!

Let's check if we can do more optimisations:

```
cython -a exp_cython.pyx
```

One more tiny optimization

```
cimport cython

@cython.cdivision(True)
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
```

Compiler directive:
use C division rules
instead of Python division rules

Pure Python mode

```
import cython

@cython.locals(x=cython.double, terms=cython.int,
               sum=cython.double, power=cython.double,
               factorial=cython.double, i=cython.int)
def exp(x, terms = 50):
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Python -> Cython

1) Write a Python module and test it.
2) Use a profiler to find the time-intensive sections.
3) Change name from module.py to module.pyx. Write setup.py for compilation. Compile and test again.
4) In the critical sections, convert the loop indices to C integers (cdef int ...).
5) Convert all variables used in the critical sections by C variables.

Relelasing the GIL

```
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
    with nogil:
        sum = 0.
        power = 1.
        fact = 1.
        for i in range(terms):
            sum += power/fact
            power *= x
            fact *= i+1
    return sum
```

Other optimisation techniques

Staying in plain Python

- Avoid loops
- For long loops, use `xrange` instead of `range`.
- Try list comprehensions instead of iterative `append`.
- Use the highly optimised data structures in Python, in particular sets and dictionaries

Python compilers

- Numba: http://numba.pydata.org/
- Pythran: https://pythonhosted.org/pythran/
- PyPy: http://pypy.org/