Different ways to improve python code efficiency

Pierre-Yves Dupont

Simple example

Fibonacci number

fibo.py

```
1 def fibo(n):
2     if n == 0 or n == 1:
3         return n
4     return fibo(n - 2) + fibo(n - 1)
```

Pure python version: fibo(38) computed in 15s

Cython

- apt-get install cython

- apt-get install cython
- programming language based on python (Pyrex)

- apt-get install cython
- programming language based on python (Pyrex)
- simplify development of C extensions for Python

- apt-get install cython
- programming language based on python (Pyrex)
- simplify development of C extensions for Python
- possible to transform python code in C code

- apt-get install cython
- programming language based on python (Pyrex)
- simplify development of C extensions for Python
- possible to transform python code in C code
- used in Scipy or SAGE

Cython on fibo.py

```
#build fibo.c file
cython fibo.py
#build fibo.so library
gcc fibo.c -o fibo.so -shared -pthread -fPIC \
    -fwrapv -O2 -Wall -fno-strict-aliasing $(pkg-config python --cflags)
```

Using fibo.so

```
import fibo
print fibo.fibo(38)
```

Simple Cython version: fibo(38) computed in 10s (33%)

More improvement

fibo.*pyx*

```
1 cpdef int fibo(int n):
2     if n == 0 or n == 1:
3         return n
4     return fibo(n-2) + fibo(n-1)
```

Cython compiling

```
1 cython fibo.py
2 gcc fibo.c -o fibo.so -shared -pthread -fPIC \
3     -fwrapv -O2 -Wall -fno-strict-aliasing $(pkg-config python --cflags)
```

Cython/pyx version: fibo(38) computed in 5s (66.7%)

Standalone executable

```
cython --embed test_fibo.py -o fibo.c
gcc fibo.c -o fibo $(pkg-config python --cflags) -lpython2.7
./fibo
```

See more on `http://docs.cython.org/index.html`

Pure C version

fibo_pure_c.c

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

long fibo(long n){
  if(n == 0 || n == 1){return n;}
  return fibo(n - 2) + fibo(n - 1);
}

void main(int argc, char* argv[]){
  printf("%ld\n",fibo(38));
}
```

C version: fibo(38) computed in 0.5s

How to extend python with pure C libraries

SWIG

- Simplified Wrapper and Interface Generator

SWIG

- Simplified Wrapper and Interface Generator
- Tool used to connect C/C++ libraries to programs written in Python, Perl, Ruby, R, Java...

SWIG

- Simplified Wrapper and Interface Generator
- Tool used to connect C/C++ libraries to programs written in Python, Perl, Ruby, R, Java...
- apt-get install swig

Connect fibo_pure_c.c to Python

Interface file fibo_pure_c.i

```
1 %module fibo_pure_c
2 %{
3 extern long fibo(long n);
4 %}
5 extern long fibo(long n);
```

Wrapper generation

```
1 #generate fibo_pure_c_wrap.c
2 swig -python fibo_pure_c.i
3 #build the library
4 gcc -c fibo_pure_c.c fibo_pure_c_wrap.c \
5     -I/usr/include/python2.7
6 ld -shared fibo_pure_c.o fibo_pure_c_wrap.o -o _fibo_pure_c.so
```

Use the C library in python

```
1 import fibo_pure_c as fibo
2 print fibo.fibo(38)
```

Python with pure C library: computation of fibo(38) in 0.5s

Another problem: evaluation of Pi

Pi

$$\pi = \int_0^1 f(x)\,\mathrm{d}x, \textit{ with } f(x) = \frac{4}{1+x^2} \tag{1}$$

$$\pi = \frac{1}{n}\sum_{i=1}^{n} f(x_i), \textit{ with } x_i = \frac{i-\frac{1}{2}}{n} \textit{ for } i = 1, \ldots, n \tag{2}$$

```
1  import sys
2  PI = 3.141592653589793
3
4  def f(a):
5      return 4.0/(1.0+a**2)
6
7  def main():
8      while(1):
9          n = raw_input("Enter the number of intervals: (0  quits)\n")
10         try:
11             n=int(n)
12         except ValueError:
13             return 2
14         if n == 0: break
15         #LOOP TO PARALLELIZE
16         h = 1.0/n
17         sum = 0.0
18         for i in range(1,n):
19             x = h*(i - 0.5)
20             sum += f(x)
21         pi = h * sum
22         #END
23         sys.stdout.write("pi is approximatly: %.16f Error is: %.16f \n"\
24         % (pi, abs(pi-PI)));
```

$n = 10^8$ pi evaluated in 25s, $n = 10^9$ crash

Weave

```
1  import sys
2  from scipy import weave
3  from scipy.weave import converters
4  import time
5  PI = 3.141592653589793
6
7  code="""
8  int i;
9  double x;
10 double sum = 0.0;
11 for(i = 1;i <= n; i++) {
12     x = h*((double)i-(double)0.5);
13     sum += (double)4.0/((double)1.0+(x*x));
14 }
15 return_val = sum;
16 """
17 vars = "h n".split()
18 ...
```

Weave

```
...
def main():
    while(1):
        n = raw_input("Enter the number of intervals: (0  quits)\n")
        try:
            n=int(n)
        except ValueError:
            return 2
        if n == 0: break
        #LOOP TO PARALLELIZE
        h = 1.0/n
        sum = float(0.0)
        tps = time.time()
        sum = weave.inline(code, vars,
              type_converters = converters.blitz,
              compiler = 'gcc')
        print time.time() - tps
        pi = h * sum
        #END
        sys.stdout.write("pi is approximatly: %.16f Error is: %.16f \n"\
        % (pi, abs(pi-PI)));
```

$n = 10^8$ pi evaluated in 0.3s, $n = 10^9$ pi evaluated in 4s

MPI and OpenMP

Parallel computing with MPI and OpenMP

MPI

- Message Passing Interface

Parallel computing with MPI and OpenMP

MPI

- Message Passing Interface
- Fortran 77, 90, 95 and C/C++

Parallel computing with MPI and OpenMP

MPI

- Message Passing Interface
- Fortran 77, 90, 95 and C/C++
- Interfaces for C#, Java...

Parallel computing with MPI and OpenMP

MPI

- Message Passing Interface
- Fortran 77, 90, 95 and C/C++
- Interfaces for C#, Java...
- Parallel machines and on workstation clusters

Parallel computing with MPI and OpenMP

OpenMP

- Open MultiProcessing

Parallel computing with MPI and OpenMP

OpenMP

- Open MultiProcessing
- Fortran 90, 95 and C/C++

Parallel computing with MPI and OpenMP

OpenMP

- Open MultiProcessing
- Fortran 90, 95 and C/C++
- Shared memory multiprocessing programming

Parallel computing with MPI and OpenMP

OpenMP

- Open MultiProcessing
- Fortran 90, 95 and C/C++
- Shared memory multiprocessing programming
- Most of the processor architectures compatible with MPI

Parallel computing with MPI and OpenMP

OpenMP

- Open MultiProcessing
- Fortran 90, 95 and C/C++
- Shared memory multiprocessing programming
- Most of the processor architectures compatible with MPI
- Based on *pragmas* (compiler-specific preprocessor directives)

Simple C version

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #define PI 3.14159265358979323846264338327950291
5
6 double f(double a) {
7   return (double)4.0/((double)1.0+(a*a));
8 }
9 ...
```

Simple C version

```
1  int main(int argc, char* argv[])
2  {
3    int n, i;
4    double h, pi, sum, x ;
5    for(;;) {
6      printf("Enter the number of intervals: (0  quits)");
7      if(!scanf("%lu",&n)){return 2;}
8      if(n ==0)
9        break;
10     //LOOP TO PARALLELIZE
11     h = ((double)1.0)/(double)n;
12     sum = 0.0;
13     for(i =1;i<=n;i++) {
14       x = h*((double)i-(double)0.5);
15       sum += f(x);
16     }
17     pi = h*sum;
18     //END
19     printf("pi is approximatly: %.16f Error is: %.16f \n",\
20     pi, fabs(pi-PI));
21   }
22   return EXIT_SUCCESS ;
23 }
24 $n = 10^9$ pi evaluated in 4sec
```

Pi evaluation using MPI

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <mpi.h>
5  #define PI 3.14159265358979323846264338327950291
6
7  double f(double a) {
8    return (double)4.0/((double)1.0+(a*a));
9  }
10 ...
```

Pi evaluation using MPI

```
1  int main(int argc, char* argv[])
2  {
3    int n, i;
4    double h, pi, sum, x ;
5
6    double mypi;
7    int myid, numprocs, islave;
8    MPI_Status status;
9    MPI_Init(&argc, &argv);
10   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
11   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
12   n=0;
13 ...
```

```
1   for(;;) {
2     if (myid == 0){
3       printf("Enter the number of intervals: (0  quits)\n");
4       if(!scanf("%d",&n)){return 2;}
5     }
6     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
7     if(n ==0)
8       break;
9     //LOOP TO PARALLELIZE
10    h = ((double)1.0)/(double)n;
11    sum = 0.0;
12    for(i = myid + 1; i <= n; i += numprocs){
13      x = h * ((double)i - (double)0.5);
14      sum += f(x);
15    }
16    mypi = h*sum;
17    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
18    //END
19    if(myid == 0){
20      printf("pi is approximatly: %.16f Error is: %.16f \n",\
21      pi, fabs(pi-PI));
22    }
23  }
24  MPI_Finalize();
25  return EXIT_SUCCESS ;
26 }
```

Compilation and Run

```
mpicc -g -O2 pi_mpi.c -o pi_mpi #mpi_c++ exists
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- Different compiler, some differences with standard gnu compiler

Compilation and Run

```
mpicc -g -O2 pi_mpi.c -o pi_mpi #mpi_c++ exists
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- Different compiler, some differences with standard gnu compiler
- The code is completely parallelized

Compilation and Run

```
mpicc -g -O2 pi_mpi.c -o pi_mpi #mpi_c++ exists
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- Different compiler, some differences with standard gnu compiler
- The code is completely parallelized
- A lot of differences between simple C code and MPI code

Compilation and Run

```
mpicc -g -O2 pi_mpi.c -o pi_mpi #mpi_c++ exists
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- Different compiler, some differences with standard gnu compiler
- The code is completely parallelized
- A lot of differences between simple C code and MPI code
- $n = 10^9$ pi evaluated in 2s (12 threads)

Pi evaluation using OpenMP

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <omp.h>
5  #define PI 3.14159265358979323846264338327950291
6
7  double f(double a) {
8    return (double)4.0/((double)1.0+(a*a));
9  }
10 ...
```

```c
int main(int argc, char* argv[]){
  int n, i;
  double h, pi, sum, x ;
  for(;;) {
    printf("Enter the number of intervals: (0  quits)\n");
    if(!scanf("%u",&n)){return 2;}
    if(n ==0)
      break;
    //LOOP TO PARALLELIZE
    h = ((double)1.0)/(double)n;
    sum = 0.0;
#pragma omp parallel for num_threads(12) private(i,x) reduction(+:sum)
    for(i =1;i<=n;i++) {
      x = h*((double)i-(double)0.5);
      //#pragma omp critical
      sum += f(x);
    }
    pi = h*sum;
    //END
    printf("pi is approximatly: %.16f Error is: %.16f \n",\
    pi, fabs(pi-PI));
  }
  return EXIT_SUCCESS ;
}
```

Compilation and Run

```
gcc -fopenmp -Wall -O2 pi_openmp.c -o pi_openmp #g++ works fine
./pi_openmp
```

- Standard compiler, use openmp library

Compilation and Run

```
gcc -fopenmp -Wall -O2 pi_openmp.c -o pi_openmp #g++ works fine
./pi_openmp
```

- Standard compiler, use openmp library
- Only part of code (loops) are parallelized

Compilation and Run

```
gcc -fopenmp -Wall -O2 pi_openmp.c -o pi_openmp #g++ works fine
./pi_openmp
```

- Standard compiler, use openmp library
- Only part of code (loops) are parallelized
- Really close to original C code

Compilation and Run

```
gcc -fopenmp -Wall -O2 pi_openmp.c -o pi_openmp #g++ works fine
./pi_openmp
```

- Standard compiler, use openmp library
- Only part of code (loops) are parallelized
- Really close to original C code
- $n = 10^9$ pi evaluated in 3s (12 threads)

Pi evaluation using OpenMP and MPI

In MPI version of code, insertion of OMP pragma

```
1      h = ((double)1.0)/(double)n;
2      sum = 0.0;
3      #pragma omp parallel for reduction(+:sum) private(i,x)
4      for(i = myid + 1; i <= n; i += numprocs){
5        x = h * ((double)i - (double)0.5);
6        sum += f(x);
7      }
8      mypi = h*sum;
9      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Compilation and Run

```
mpicc -g -O2 -fopenmp pi_mpi.c -o pi_mpi
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- MPI compiler

Compilation and Run

```
mpicc -g -O2 -fopenmp pi_mpi.c -o pi_mpi
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- MPI compiler
- MPI used to share the computation on different nodes

Compilation and Run

```
mpicc -g -O2 -fopenmp pi_mpi.c -o pi_mpi
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- MPI compiler
- MPI used to share the computation on different nodes
- OpenMP used to make threads within a node

Compilation and Run

```
mpicc -g -O2 -fopenmp pi_mpi.c -o pi_mpi
mpirun -np 12 -hostfile hostfile ./pi_mpi
```

- MPI compiler
- MPI used to share the computation on different nodes
- OpenMP used to make threads within a node
- $n = 10^9$ pi evaluated in 2s (6 nodes, 2 threads)

Inlining parallelized C/C++ code in python

```
import sys
from scipy import weave
from scipy.weave import converters
import time
PI = 3.141592653589793

code="""
int i;
double x;
double sum = 0.0;
#pragma omp parallel for private(i,x) reduction(+:sum)
for(i = 1;i <= n; i++) {
    x = h*((double)i-(double)0.5);
    sum += (double)4.0/((double)1.0+(x*x));
}
return_val = sum;
"""
vars = "h n".split()
weave_omp = \
{
    'headers': ['<omp.h>'],
    'extra_compile_args': ['-fopenmp'],
    'extra_link_args': ['-lgomp']
}
```

Inlining parallelized C/C++ code in python

```
def f(a):
    return 4.0/(1.0+a**2)

def main():
    while(1):
        n = raw_input("Enter the number of intervals: (0  quits)\n")
        try:
            n=int(n)
        except ValueError:
            return 2
        if n == 0: break
        #LOOP TO PARALLIZE
        h = 1.0/n
        sum = float(0.0)
        tps = time.time()
        sum = weave.inline(code, vars,
                type_converters = converters.blitz,
                compiler = 'gcc', **weave_omp)
        print time.time() - tps
        pi = h * sum
        #END
        sys.stdout.write("pi is approximatly: %.16f Error is: %.16f \n" %
                (pi, abs(pi-PI)));
```

$n = 10^9$ pi evaluated in 1.3s (12 threads)

Comparison of the efficiency of the different methods

slow	430 seconds
numeric	2.76 seconds
Cython pyrex	2.55 seconds
fortran77	2.53 seconds
fortran90	0.60 seconds
fortran95	0.59 seconds
C	2.20 seconds