1 Profiler for C/C++

1.1 Valgrind

Valfrind is a well known C/C++ profiler for Unixs, please go to check http://valgrind.org/.

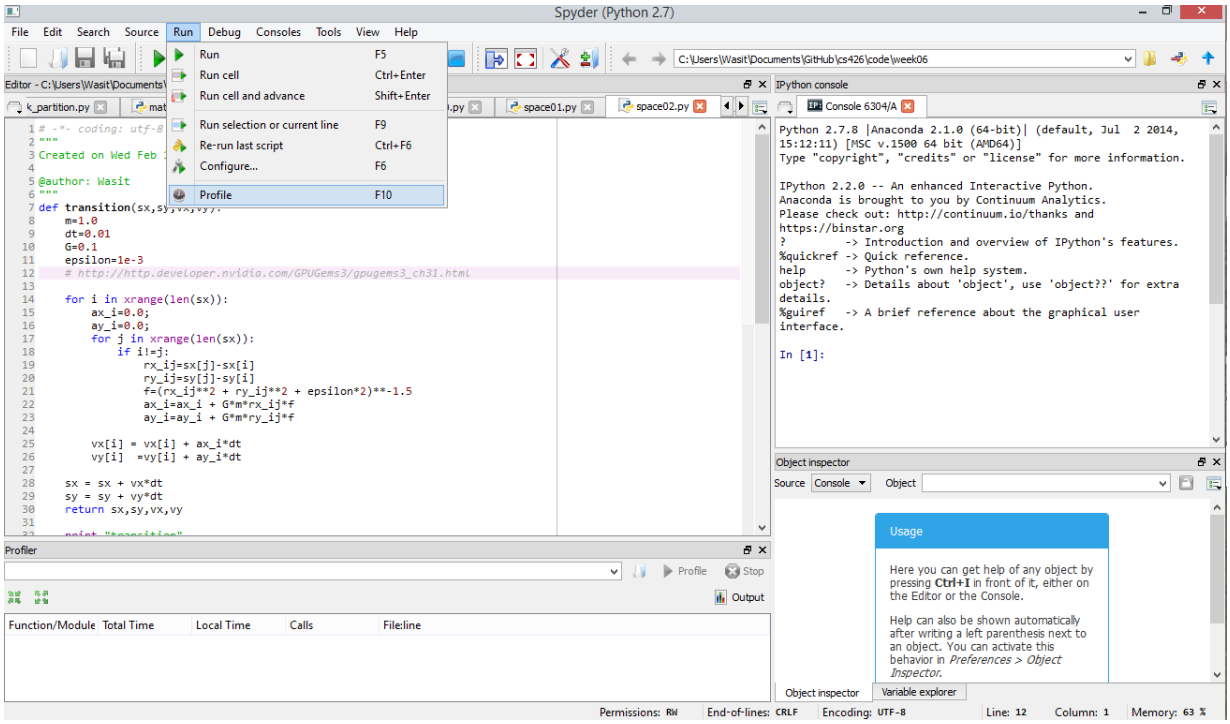
To install in an Ubuntu: $sudo apt-get install valgrind

1.2 Gprof

1. Compile with –pg gcc -Wall –pg –o main main.c
2. Execute the executable file. The result will be saced in gmon.out

2 Profiler for python

In Spyder that comes along with Python Anaconda, we can use the profile tool by pressing F10 key or find it on the menu bar.



3 Monte Carlo to find Pi

The example of parallelism implementation to perform Monte Carlo simulation to estimate the value of Pi. The coordinate (x,y) is sampled from uniform random distribution, where the coordinate is in a square $0 <= x < 1 \; and \; 0 <= y < 1$. By using C++11 built-in function;

```
std::default_random_engine e((unsigned int)time(0)+ID);
std::uniform_real_distribution<double> distribution(0.0,1.0);
```

Then, the boundary of a unit circle is calculated by a simple Euclidian distance and a sample in the circle boundary is counted.

```
if(x*x+y*y<1.0){
    sum[ID]++;
}
```

The ratio between samples in the unit circle divided by total sample is approximately equal to the ratio between a quarter of a unit circle divided by a unit area of the square.

$$\frac{A_{circle}}{A_{square}} = \frac{N_{in}}{N_{total}}$$

Here the simple code extended from the previous lecture.

```
#include <iostream>
using namespace std;
#include <omp.h>
#include <random>
#include <math.h>

#define N_cores 8
int main(void){
        unsigned long long sum[N_cores]={};
 int spt = 1e7;//sample per thread
 int ID,i;
        #pragma omp parallel shared(sum,spt) private(ID,i)
        {
                ID = omp_get_thread_num();
       std::default_random_engine e((unsigned int)time(0)+ID);
             std::uniform_real_distribution<double>
distribution(0.0,1.0);
                //int x[1000];
                //initialization
                for(i=0;i<spt;i++){
                        double x = distribution(e);
             double y = distribution(e);
             //cout<<"L0-->core:"<< ID <<"    "<<x<<","<<y<<std::endl;
             if(x*x+y*y<1.0){
                   sum[ID]++;
             }
                }
       #pragma omp critical
                cout<<"L0-->core:"<< ID <<", sum:"<<
sum[ID]<<std::endl;
       #pragma omp barrier
       if(!(ID%2)){
             sum[ID]+=sum[ID+1];
             cout<<"L1-->core:"<< ID <<", sum:"<< sum[ID]<<std::endl;
       }
       #pragma omp barrier
       if(!(ID%4)){
             sum[ID]+=sum[ID+2];
             cout<<"L2-->core:"<< ID <<", sum:"<< sum[ID]<<std::endl;
       }
       #pragma omp barrier
       if(!(ID%8)){
             sum[ID]+=sum[ID+4];
             cout<<"L3-->core:"<< ID <<", sum:"<< sum[ID]<<std::endl;
       }
 }

 cout<<"M_PI  : "<<M_PI<<std::endl;
 cout<<"Result: "<<(double)sum[0]/N_cores/spt*4.0f<<std::endl;
        return 0;
}
```

4 OMP for reduction

Using unroll for-loop is a little bit complicated but it gives a better idea of how the parallel process actually works. Here is another option to perform for-loop reduction in much easier way.

The predefine code `#pragma omp parallel for reduction (+:sum)` tell the complier that in the for-loop the variable sum from different threads will be reduced into a single variable in the master thread using summation operator.

```
#include <iostream>
using namespace std;
#include <omp.h>

#define SIZE 8
int main(void){
 int x[SIZE];
 int sum=0;
 for(int i=0;i<SIZE;i++){
       x[i]=i;
 }
 #pragma omp parallel for reduction (+:sum)
 for(int i=0;i<SIZE;i++){
               sum+=x[i];
        }
 cout<<sum<<std::endl;
 return 0;
}
```