HIGH PERFORMANCE COMPUTING for SCIENCE & ENGINEERING (HPCSE) I

HS 2021

EXERCISE 05: MONTE CARLO INTEGRATION & OPENMP

Pascal Weber (webepasc@ethz.ch)

Computational Science and Engineering Lab
ETH Zürich

25.11.2021

Outline

I. Exercise 1 (Monte-Carlo Integration)
II. Exercise 2/3 (OpenMP Bughunt)

Monte-Carlo Integration

We want to perform integration $I = \int_\Omega f(x)\,\mathrm{d}x$ over the domain Ω

From probability theory we know that the expectation value of $f(x)$ for a uniform distribution $\mathcal{U}_\Omega(x)$ over the domain Ω reads

$$\mathbb{E}_{x\sim\mathcal{U}_\Omega(\cdot)}[f(x)] = \frac{1}{|\Omega|}\int_\Omega f(x)\,\mathrm{d}x$$

From the central-limit theorem we know we can approximate an expectation value by using samples $x_i \sim \mathcal{U}_\Omega(\,\cdot\,)$ with $i = 1,\ldots,N$ from the distribution

$$\mathbb{E}_{x\sim\mathcal{U}_\Omega(\cdot)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

Combining this knowledge is the foundation for **Monte-Carlo Integration**, where we approximate the integral by evaluating the function at randomly sampled locations

$$I \approx \frac{|\Omega|}{N}\sum_{i=1}^{N} f(x_i)$$

Estimating π using Monte-Carlo Integration

We know that we can write the volume of an object as an integral over the characteristic function χ in an enclosing domain Ω

$$V = \int_{\Omega} \chi(x)\ \mathrm{d}x$$

For the unit circle, the characteristic function is

$$\chi(x, y) = \begin{cases} 1, & x^2 + y^2 \leq 1, \\ 0, & \text{otherwise}. \end{cases}$$

We know that the area of the circle is π and thus we know that

$$\pi = 4 \int_{[0,1]^2} \chi(x, y)\ \mathrm{d}x\mathrm{d}y$$

Using Monte-Carlo integration we can perform the integral by taking samples $x_i \sim \mathcal{U}_{[0,1]}(\,\cdot\,)$ and $y_i \sim \mathcal{U}_{[0,1]}(\,\cdot\,)$ with $i = 1,\ldots,N$

$$\pi \approx \frac{4}{N} \sum_{i=1}^{N} f(x_i, y_i)$$

[Pseudo] Random Number Generators

It is hard to get true random numbers. In practice we therefore use pseudo random numbers, which are **deterministic sequences**!

Examples:

- Congruential Generators: $x_i = \left(c \cdot x_{i-1}\right) \bmod p$

$\rightarrow$ maximal period for Mersenne prime numbers $p \equiv M_q = 2^q - 1$ and $c^{p-1} \bmod p = 1$

- Lagged-Fibonacci: $x_i = x_{i-a} \oplus x_{i-b} := \left(x_{i-a} + x_{i-b}\right) \bmod 2, \quad a < b$

We will use **std::default_random_engine** to generate pseudo random numbers.

«It is the library implemention's selection of a generator that provides at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use.»

The one that is 'normally' used is **std::mt19937** (Mersenne-Twister with sequence length $2^{19937} - 1$, uniform distribution and fast)

Inspired from «Computational Physics» course by Prof. Dr. Hans Herrmann
See https://en.cppreference.com/w/cpp/numeric/random and https://de.wikipedia.org/wiki/Mersenne-Twister

Exercise 1 - montecarlo.cpp

Implementation of characteristic function

```cpp
// Characteristic function for unit circle
inline double F(double x, double y)
{
    if (x * x + y * y < 1.) { // inside unit circle
        return 1.;
    }
    return 0.;
}
```

Implementation of Monte-Carlo integration

```cpp
// Method 0: serial
double C0(size_t N)
{
    // random generator with seed 0
    std::default_random_engine g(0);

    // uniform distribution in [0, 1]
    std::uniform_real_distribution<double> u;

    // perform Monte-Carlo integration
    double pi = 0.;
    for (size_t i = 0; i < N; ++i) {
        double x = u(g);
        double y = u(g);
        pi += F(x, y);
    }
    return 4 * pi / N;
}
```

you are asked to parallelize this code in several ways

be careful with the random number generators / distributions

```cpp
// Method 1: parallelize C0 without using arrays
double C1(size_t N)
{
    double pi = 0.;

    // TODO: Implement Method 1

    return 4 * pi / N;
}

// Method 2: parallelize C0 only using
// `omp parallel for reduction`, use arrays without padding
double C2(size_t N)
{
    double pi = 0.;

    // TODO: Implement Method 2

    return 4 * pi / N;
}

// Method 2: parallelize C0 only using
// `omp parallel for reduction`, use arrays without padding
double C3(size_t N)
{
    double pi = 0.;

    // TODO: Implement Method 3

    return 4 * pi / N;
}
```

padding should be of the size of one cache line [usually 64 bytes]

The goal of the exercise is to observe the problems that can occur from false-sharing

Exercise 1 - Makefile

some syntax
$ to access content of variable
@ contains command-line arguments
> forwards output
&2 is stderr
| pipelining (forward output to next)
commands -> check man xyz
set setting of bash
echo print variable
eval runs the expression
grep search words

Compile and Launch Benchmarks

for OpenMP compatibility. Use
env2lmod; module load gcc/8.2.0

compiles the executable

runs benchmark

plot results

```
CXX = g++
CXXFLAGS = -O3 -Wall -Wextra -pedantic -std=c++14 -fopenmp

all: montecarlo

montecarlo: montecarlo.cpp
	$(CXX) $< $(CXXFLAGS) -o montecarlo

run: montecarlo
	./varym $(N)

plot: run
	./plot

clean:
	rm -rf montecarlo out results.png

.PHONY: all clean run plot runplot
```

```
#!/bin/bash

gnuplot << EOF

set terminal pngcairo

set xlabel 'threads'
set ylabel 'time [s]'
set output 'results.png'
set grid
set key Left left bottom
set logscale x 2
set logscale y 2
t0 = `sed 's,.* ,,;q' out/m0`
set style data lp
plot \
  "out/m0" lw 3 pt 7 t 'm=0, serial', \
  "out/m1" lw 3 pt 7 t 'm=1, no arrays', \
  "out/m2" lw 3 pt 7 t 'm=2, no padding', \
  "out/m3" lw 3 pt 7 t 'm=3, padding', \
  t0/x w l lw 1 lc 'black' t 'ideal'

EOF
```

varym

```
#!/bin/bash

set -eu

o=out

rm -f $o/m*
mkdir -p $o

for m in 0 1 2 3 ; do
  c="./varynt $m $@ > $o/m$m"
  echo "$c" >&2
  eval "$c"
done
```

varynt

```
#!/bin/bash

set -eu

maxnt=${OMP_NUM_THREADS:-4}

for nt in `seq 1 $maxnt` ; do
  c="OMP_NUM_THREADS=$nt ./montecarlo $@"
  echo "$c" >&2
  o=`eval "$c"`
  wt=`echo "$o" | grep time | cut -d' ' -f2`
  echo "$nt" "$wt"
done
```

http://stanford.edu/~wpmarble/webscraping_tutorial/regex_cheatsheet.pdf

Exercise 2

Question 2: OpenMP Bug Hunting I (20 points)

Identify and explain any bugs in the following OpenMP code. Propose a solution. Assume all headers are included correctly.

```
#define N 1000

extern struct data member[N];  // array of structures, defined elsewhere
extern int is_good(int i); // returns 1 if member[i] is "good", 0 otherwise

int good_members[N];
int pos = 0;

void find_good_members()
{
  #pragma omp parallel for
  for (int i=0; i<N; i++) {
    if (is_good(i)) {
      good_members[pos] = i;

      #pragma omp atomic
      pos++;
    }
  }
}
```

Hints:

- In your solution you can use "omp critical" or "omp atomic capture"[1]

Exercise 3

Question 3: OpenMP Bug Hunting II (20 points)

a) Identify and explain any *bugs* in the following OpenMP code. Propose a solution. Assume all headers are included correctly.

```
// assume there are no OpenMP directives inside these two functions
void do_work(const float a, const float sum);
double new_value(int i);

void time_loop()
{
  float t = 0;
  float sum = 0;

  #pragma omp parallel
  {
    for (int step=0; step<100; step++)
    {
      #pragma omp parallel for nowait
      for (int i=1; i<n; i++)
      {
        b[i-1] = (a[i]+a[i-1])/2.;
        c[i-1] += a[i];
      }

      #pragma omp for
      for (int i=0; i<m; i++)
        z[i] = sqrt(b[i]+c[i]);

      #pragma omp for reduction(+:sum)
      for (int i=0; i<m; i++)
        sum = sum + z[i];

      #pragma omp critical
      {
        do_work(t, sum);
      }

      #pragma omp single
      {
        t = new_value(step);
      }
    }
  }
}
```

b) Identify and explain any *improvements* that can be made in the following OpenMP code. Propose a solution. Assume all headers are included correctly.

```
void work(int i, int j);

void nesting(int n)
{
  int i, j;
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0; i<n; i++)
    {
      #pragma omp parallel
      {
        #pragma omp for
        for (j=0; j<n; j++)
        {
          work(i, j);
        }
      }
    }
  }
}
```