P. Koumoutsakos
S. Martin
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2021

Set 5 - Monte Carlo Integration, OpenMP

Issued: November 26, 2021
Hand in (optional): December 10, 2021 12:00

Question 1: Parallel Monte Carlo using OpenMP (35 points)

Monte Carlo integration is a method to estimate the value of an integral over a domain Ω by taking samples $x_i \sim \mathcal{U}(\Omega)$ from a uniform distribution on the domain Ω. First, note that the integral can be expressed as an expectation value over the uniform distribution

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

where $|\Omega|$ is the volume of the domain Ω. The central limit theorem states that we can estimate the expectation value using the average

$$\mathbb{E}_{x\sim\mathcal{U}(\Omega)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)\,, \quad \text{for } x_i \sim \mathcal{U}(\Omega) \tag{2}$$

Combining eq. (1) and eq. (2) we conclude that we can compute the integral as

$$\int_\Omega f(x)\,\mathrm{d}x = \frac{|\Omega|}{N}\sum_{i=1}^{N} f(x_i)\,, \quad \text{for } x_i \sim \mathcal{U}(\Omega) \tag{3}$$

In the skeleton code you are given a serial implementation of this algorithm estimating the value of π. The goal of the exercise is to parallelize the code using openMP.

In the skeleton code we estimate π by using that we can compute the area of an arbitrary object $\Omega^{(s)}$ by integrating the indicator function $\chi^{(s)}$ (i.e. $\chi^{(s)}(x) = 1$ for $x \in \Omega^{(s)}$ and $\chi^{(s)}(x) = 0$ for $x \in \Omega \setminus \Omega^{(s)}$) over an enclosing domain Ω (i.e. $\Omega^{(s)} \subseteq \Omega$). For a unit circle centered at $(0, 0)$, the characteristic function has the form

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

Therefore, the number π can be estimated as

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

where (x_i, y_i) are samples from uniform distribution on the square $[0, 1] \times [0, 1]$.

In order to compile the code you will need to call `env2lmod; module load gcc/8.2.0` on Euler.

a) Given a serial implementation of the algorithm provided in the skeleton code, write a parallel version using OpenMP. Make sure you do not introduce race conditions and that random generators are initialized differently on each thread. For storing the thread-local data, you may need to use arrays indexed by the thread id or rely on data-sharing attributes of OpenMP. Provide three versions of the implementation

   1. use any OpenMP directives, arrays are not allowed;
   2. the only available directive is `#pragma omp parallel for reduction`, arrays allowed but without additional padding, this may cause false sharing;
   3. the only available directive is `#pragma omp parallel for reduction`, arrays allowed and must include padding to avoid false sharing.

b) Run the program both on your computer and on Euler. The makefile provides various tools

   - `make` builds the executable,
   - `make run` runs the executable for all available methods (`m=0,1,2,3`) varying the number of threads between 1 and `OMP_NUM_THREADS` (if set, otherwise 4) and writes the timings to new directory `out`,
   - `make plot` plots the timings collected in directory `out`.

   Compare the plots for the methods you implemented, see if the results can be explained by false sharing.

c) Answer the following questions:

   - Is the amount of computational work equal among all threads (for large number of samples)?
   - Do you observe perfect scaling of your code? Explain why.
   - Do you get exactly the same numerical results if you run the same program under the same conditions twice? Are there reasons for slight changes in the results? Consider cases of (a) serial program, (b) OpenMP with one thread, (c) OpenMP with multiple threads.

Question 2: OpenMP Bug Hunting I (10 points)

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

```
1   #define N 1000
2
3   extern struct data member[N];  // array of structures, defined elsewhere
4   extern int is_good(int i); // returns 1 if member[i] is "good", 0 otherwise
5
6   int good_members[N];
7   int pos = 0;
8
9   void find_good_members()
10  {
11    #pragma omp parallel for
12    for (int i=0; i<N; i++) {
13      if (is_good(i)) {
14        good_members[pos] = i;
15
16        #pragma omp atomic
17        pos++;
18      }
19    }
20  }
```

Hints:

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

[1] omp atomic capture: OpenMP specs 3.1, section 2.8.5, especially page 74, lines 8–13.

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.

```
1         // assume there are no OpenMP directives inside these two functions
2         void do_work(const float a, const float sum);
3         double new_value(int i);
4
5         void time_loop()
6         {
7           float t = 0;
8           float sum = 0;
9
10          #pragma omp parallel
11          {
12            for (int step=0; step<100; step++)
13            {
14              #pragma omp parallel for nowait
15              for (int i=1; i<n; i++)
16              {
17                b[i-1] = (a[i]+a[i-1])/2.;
18                c[i-1] += a[i];
19              }
20
21              #pragma omp for
22              for (int i=0; i<m; i++)
23                z[i] = sqrt(b[i]+c[i]);
24
25              #pragma omp for reduction(+:sum)
26              for (int i=0; i<m; i++)
27                sum = sum + z[i];
28
29              #pragma omp critical
30              {
31                do_work(t, sum);
32              }
33
34              #pragma omp single
35              {
36                t = new_value(step);
37              }
38            }
39          }
40        }
```

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

```
1         void work(int i, int j);
2
3         void nesting(int n)
4         {
5           int i, j;
6           #pragma omp parallel
7           {
8             #pragma omp for
9             for (i=0; i<n; i++)
10            {
11              #pragma omp parallel
12              {
13                #pragma omp for
14                for (j=0; j<n; j++)
15                {
16                  work(i, j);
17                }
18              }
19            }
20          }
21        }
```