ETH zürich

High Performance Computing for Science and Engineering I

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

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

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

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

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

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

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.

   File `solution_code/montecarlo.cpp` contains reference implementations:

   1. private instance of random generator defined inside the parallel region; (5 pt), -1pt for each mistake
   2. array of generators without padding; (5 pt), -1pt for each mistake
   3. array of structures containing a random generator and padding `char[64]`. (5 pt), -1pt for each mistake

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.

   Fig. 1 show the execution time on a full node on euler Euler (bsub -W 02:00 -n 24 -R fullnode -Is bash). False sharing is observed for the implementation using array without padding. However, we find that the result depends on the used optimization flags. ((5 pt)+(5 pt) for each of the plots)

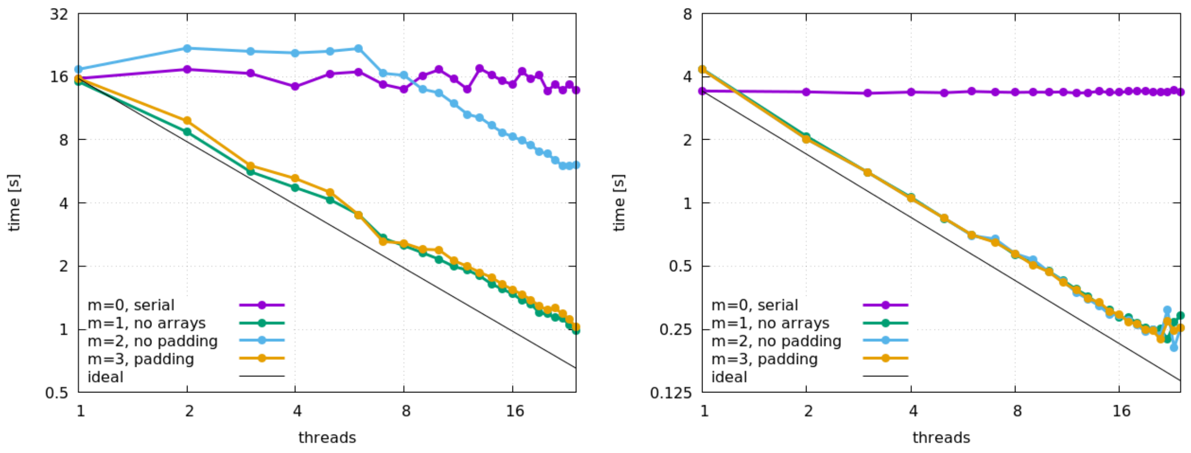

Figure 1: Runtime on Euler II, compiler gcc (GCC) 8.2.0 with `-O0` (left) and `-O3` (right).

c) Answer the following questions:

- Is the amount of computational work equal among all threads (for large number of samples)?
  Yes, equal apart from an imbalance if N is not divisible by the number of threads which is negligible for large N. (2 pt)
- Do you observe perfect scaling of your code? Explain why.
  Without false sharing (i.e. methods `C1` and `C3`), the code should have perfect scaling as all threads are independent and all data fits in cache. (2 pt)
- 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.
  If executed with one thread, any version produces identical values of the integral between multiple runs as long as the generator is initialized with the same seed. (2 pt) In practice, this is also observed for parallel execution. However, reordering of floating point operations introduced by the reduction may cause slight difference close to machine precision. (2 pt) Changing the integrand to a continuous function would make the effect stronger. (2 pt)

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.

```c
#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]

In order to avoid data races between different updates of shared variable pos, the code puts the increment in a atomic construct. However, the code is incorrect, because there is a data race between the read of pos right before the atomic construct and the write of pos within the construct.

Changing the body of the if-statement to one of the following gives the correct result:

```c
        int mypos;
        #pragma omp critical
        {
          mypos = pos;
          pos++;
        }
        good_members[mypos] = i;
```

```c
        int mypos;
        #pragma omp atomic capture
        mypos = pos++;

        good_members[mypos] = i;
```

(5 + 5 pt) for the correct problem identification and providing a solution, -2pt for partial answers

[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.

```
// 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);
     }
   }
 }
}
```

1. The keywords *nowait* and *parallel* in the first for loop should be removed.
2. A barrier must be placed after *do_work* as a thread could move on to update the value of *t* while other threads have not reached the function *do_work* yet

(5 + 5 pt) for the correct problem identification and providing a solution, -2pt for partial answers

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
12          {
13            #pragma omp for
14            for (j=0; j<n; j++)
15            {
16              work(i, j);
17            }
18          }
19        }
20      }
21    }
```

1. Lines 6 and 8 can be combined into a single `#pragma omp parallel for`. This reduces the implicit barriers of these two lines from two to one.
2. Lines 11 and 13 can be combined into a single `#pragma omp parallel for`. This reduces the implicit barriers of these two lines from two to one.
3. Use only a single OpenMP directive (no nested parallelism) by using a collapse clause: `#pragma omp parallel for collapse(2)`. In this case the code would look as follows:

(5 + 5 pt) for the correct problem identification and providing a solution, -2pt for partial answers

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

void nesting(int n)
{
  int i, j;
  #pragma omp parallel for collapse(2)
  for (i=0; i<n; i++)
  for (j=0; j<n; j++)
  {
    work(i, j);
  }
}
```