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

Fall semester 2018

Set 3 - Monte Carlo, OpenMP, False Sharing

Issued: October 12, 2018
Hand in (optional): October 19, 2018 23:59

Question 1: Parallel Monte Carlo using OpenMP

Monte Carlo integration is a method to estimate the value of an integral as the mean over a set of random variables. For instance,

$$\int_{\Omega} f(x)\,dx \;\approx\; \frac{|\Omega|}{N}\sum_{i=1}^{N} f(X_i),$$

where X_i are samples from a uniform distribution on the domain Ω. The algorithm can be applied for calculation of volume of arbitrary shapes, in which case the integrand is the indicator function. For a unit circle centered at $(0,0)$, it has the form

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

Therefore, the number π can be estimated as

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

where X_i and Y_i are samples from uniform distribution on the square $[0,1]\times[0,1]$.

a) Given a serial implementation of the algorithm provided in the skeleton code, write a parallel version using OpenMP by splitting the sampling work among multiple threads. Make sure you do not introduce race conditions and the 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, one for each of the following cases

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

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

   1. private instance of random generator defined inside the parallel region;
   2. array of generators without padding;
   3. array of structures containing a random generator and padding `char[64]`.

b) (**2 points**) 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. 1-2 show the execution time on a laptop and on Euler. False sharing is observed for the implementation using array without padding. However, this depends on the compiler version and optimization flags.

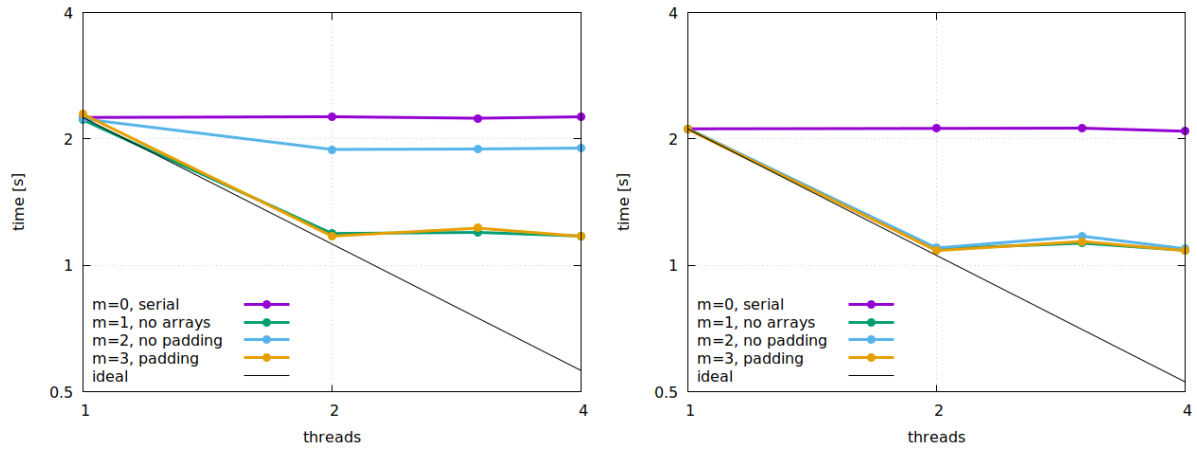

Figure 1: Runtime on Intel i7-5557U, compiler GCC 8.2.1 with `-O2` (left) and `-O3` (right).

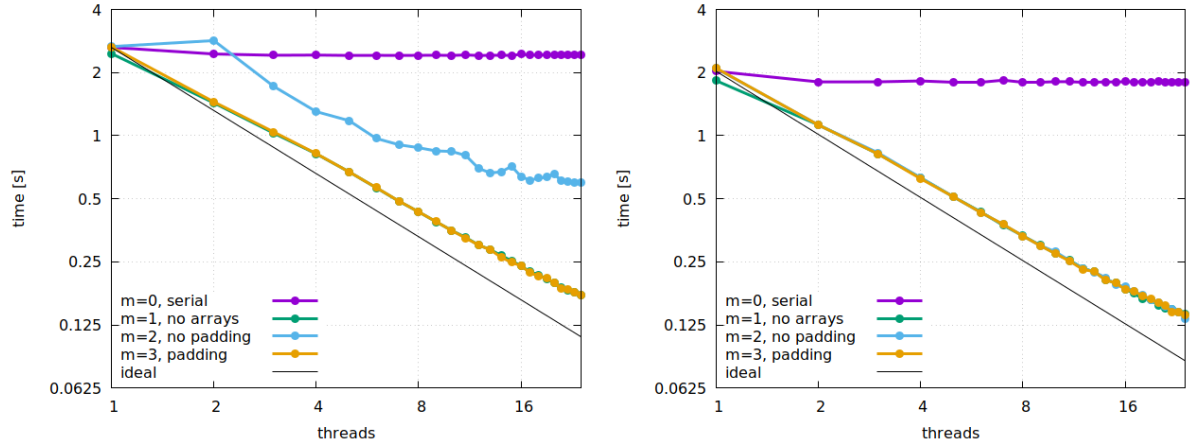

Figure 2: Runtime on Euler II, compiler GCC 4.8.2 with `-O2` (left) and `-O3` (right).

c) Answer the following questions:

- (**1 points**) 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.
- (**1 points**) 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.
- (**1 points**) 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. 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. Changing the integrand to a continuous function would make the effect stronger.

d) (Optional) Anderson's lock from Question 3 of the previous Set 2, involved a boolean array shared and simultaneously accessed by all threads. Discuss the possibility of false sharing in that case, propose a solution and measure the difference in performance.

False sharing is caused by array `flag` accessed by all threads in a while-loop and can be avoided with padding.

Question 2: OpenMP bug hunting

(**2 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]

In order to avoid data races between different updates of global 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:

```
1 int mypos;
2 #pragma omp critical
3 {
4     mypos = pos;
5     pos++;
6 }
7 good_members[mypos] = i;
```

```
1 int mypos;
2 #pragma omp atomic capture
3     mypos = pos++;
4
5 good_members[mypos] = i;
```

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