Homework #3—DSC 520

Assigned: Monday, November 6, 2023
Due: Monday, November 20, 2023

Reading & viewing: If you would like additional experience with OpenMP, please view videos posted to myCourses.

You must use C/C++ or Fortran to solve these problems. Other languages (Python, Matlab, Mathematica, gnuplot, etc) should be used only to visualize the data (ie making plots).

Report: Put all the figures and answers asked for in all the questions into a single well organized PDF document (prepared using LaTeX, a Jupyter notebook, or something else). This PDF report should be just a few pages (aim for less than 4 pages). **Please print out and turn in your report before midnight by the due date**. Your report should also be uploaded to Bitbucket.

Code: On every homework you turn in, be sure to upload the computer code you used to generate your solutions to Bitbucket. Include all code used such as the C/C++ code to solve the problem, any code used to make figures or cary out analysis, and any Unix scripts. Please strive towards automation where there is one script that solves the entire problem: it will compile the code, run the code, and generate any figures or plots.

The main goal for this assignment is to write a parallelized program to solve a challenging real-world task. You will

1. (problem 1) Write the code in serial
2. (problem 2) Parallelize the code using OpenMP
3. (problem 3) Use your code to solve a hard problem. In particular, you will be computing high-dimensional integrals. Such integrals appear all the time when working with models in Bayesian statistics and machine learning contexts.

Recall that you've already written programs to approximate the integral

$$I = \int_R f(x)\,\mathrm{d}x\,,$$

by Monte Carlo integration

$$\hat{I}(N) = V\frac{1}{N}\sum_{i=1}^{N} f(X_i)\,, \tag{1}$$

where V is the volume of the region defined by R and X_i (for $i = 1, 2, \ldots$) are independent and identically distributed (uniform) random variables. In the last homework, you computed π this way, for example.

In this problem, you are asked to compute a 10-dimensional integral

$$I = \int_R f(x) d^{10}x\,. \tag{2}$$

where $f(x)$ is an arbitrary 10-dimensional function and the region R is a 10-dimensional box centered on the origin with sides of length 2. That is

$$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10} \in [-1, 1]\,.$$

In Equation (2) (and sometimes below) we use compact notation for high dimensional quantities which are awkward to write. For example,

$$d^{10}x = \mathrm{d}x_1\,\mathrm{d}x_2\,\mathrm{d}x_3\,\mathrm{d}x_4\,\mathrm{d}x_5\,\mathrm{d}x_6\,\mathrm{d}x_7\,\mathrm{d}x_8\,\mathrm{d}x_9\,\mathrm{d}x_{10}\,,$$

and $x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) \in \mathbb{R}^{10}$. Such shorthands are very commonly used.

Problem 1: (*5 points*) Writing the serial code

(a) (5 points) Implement your Monte Carlo integration algorithm in C, C++ or Fortran. For full credit, your program should accept N as input and output the approximate value (1), which I will denote by $\hat{I}(N)$, computed by the Monte Carlo method. Test your code on a case you know the answer to. For example, let

$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) = 1 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10}\,, \tag{3}$$

then the exact paper-and-pencil value of the integral is 2^{10} (can you see why?). Provide evidence that your code is working by showing the Monte Carlo estimate of the integral is converging to 2^{10} at the correct rate of $N^{-1/2}$ exactly as you have done on homework 1 and 2.

Because this problem is intended to focus on parallelization with OpenMP, I have provided my answer to this problem on the next page. You can use this code, write your own, or modify my code shown on the next page. **Even if you are using my code, you still need to run the code and demonstrate the integral is converging to 2^{10} at the correct rate**. If you are writing your own code, or making significant modifications to mine, please consider the following tips:

- The for-loop (over Monte Carlo samples) is going to run over large, positive integers. To access larger values (without overflowing) you should consider using specialized data types (long int, long long int, unsigned int, etc). Its up to you to figure out what works best (please see my hw2 solutions for tips). Also, you may need to search how to do this for your compiler/language. For example, "unsigned long int" and "unsigned long long int" are somewhat exotic and there may not be a standard way of declaring them.

- It will be easier (and in the long run better) for you to write a code that works for an arbitrary number of dimensions instead of specializing it to 10. To do this you should use arrays whenever possible. This is what I have done. For example, the random variable x has been declared as:

```
const int dim = 10;
double x[dim];
```

MonteCarlo10D.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <limits.h>

double sample_interval(const double a, const double b) {

  double x = ((double) rand())/((double) RAND_MAX);
  return (b-a)*x + a;

}

double func(double *x, int dim) {

  double f = 1.0;

  for(int i=0;i<dim;++i) {
    f += x[i]; // f = x1 + x2 + ... + x10
  }

  return f;
}

int main (int argc, char **argv) {

  const int dim  = 10; // dimension of the integration region R

  // a 2-by-2 10-dimensional box, each side is length 2
  const double V = 1024; // volume of R
  const double xL = -1.0;
  const double xR = 1.0;

  srand(time(NULL));
  long long int N = atoll( argv[1] ); // convert command-line input to N = number of points

  printf("Total points = %lli\n",N);

  double x[dim]; // array of random numbers
  double integral = 0;

  for (long long int i=1; i <= N; ++i) {

    for(int j=0; j < dim; ++j) {
      x[j] = sample_interval(xL,xR);
    }

    double f_i = func(x,dim);
    integral += f_i;
  }

  integral = (V/N)*integral;
  printf("MC points = %lld, integral = %1.5e\n",N,integral);

  return 0;
}
```

Problem 2: (*10 points*) Parallelizing your code.

Before continuing, make a copy of the serial code (which is the answer to problem 1).

(a) (5 points) Use OpenMP to parallelize the for-loop over the number of N random samples. That is, each thread should make independent random draws, independently evaluate $f(x)$ at the random value, and update the sum. Updating the sum will require coordination between threads! Rerun the test given in problem 1 to show the code still works when using multiple threads (try with at least 2). Use OpenMP routines to output the number of threads and provide evidence that multiple threads are indeed being used.

(b) (5 points) Read up on OpenMP's function omp_get_wtime() and use this to measure the code's *walltime* from within the code; this allows for more accurate timing as compared to Unix's time. omp_get_wtime is specifically meant for timing parallel code. Start the timer at the very beginning of the program and stop the timer at the very end of the program. Compare the omp timer to the one you have been using (probably Unix's program time). Use sufficiently many random samples to get realistic timing estimates, say $N = 10^7$.

Problem 3: (*10 points*)
Let us now apply our Monte Carlo code to a more complicated function to which the true value of the integral is unknown. We will also perform a speedup test on Anvil.

(a) (2 points) Modify your OpenMP parallelized code to let

$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) = \exp\left(-\sum_{i=1}^{9}\left[(1-x_i)^2 + 100\left(x_{i+1} - x_i^2\right)^2\right]\right). \tag{4}$$

This is called the Rosenbrock function. Its famous (and has its own Wikipedia page) because it's widely used as an important test function for numerical optimization algorithms (like the ADAM optimizer used in deep learning), stochastic sampling algorithms, and integration algorithms.

Propose and carry out a simple test to check that you have correctly implemented the equation (4). That is, what might you try doing to check there's no mistakes in your code for Eq. (4)? Your test should be narrowly focused – do not propose a test that involves other parts of the code.

(b) (5 points) You will now run your code (using Eq. 4 as the integrand) on Anvil using a single compute node (either submit to the job queue or run your program interactively). You will carry out a strong and weak scaling test. Depending on the choices you made to parallelize your code (in problem 2) your scaling results may look good or bad. If you are not happy with your results, you should keep in mind that the random number generator may impact results (you could try rand, rand_r, drand48, and others – if gcc cannot compile drand48 try using "-std=gnu99"). The way you update shared variables among the threads is also important (e.g. critical, atomic, or reduction). When reporting your scalability results, mention the choices you made and what attempts you explored to improve the code's scaling performance. **For full credit, please try at least 2 different approaches**.

1. *Strong scaling test*: keep the problem sized fixed (you hold N constant) and let the number of threads increase. Let $N = 10^8$ and compute the code's speedup using 1, 2, 4, 8, 16, 32, 64, and 128 threads. Your timing measurement should use the function omp_get_wtime(). Plot the number of threads versus speedup.
2. *Weak scaling* test: let the problem size N increase with the number of cores. Run your code on Anvil using a single compute node (submit with sbatch). Let $N = \text{threads} \times 10^7$ and compute the code's speedup using 1, 2, 4, 8, 16, 32, 64, and 128 threads. Your timing measurement should use the function omp_get_wtime(). Plot the number of threads versus speedup.

   NOTE: You will always run 1 thread per core, and so threads = cores. But in general they need not be the same. When running with, say, 1 thread you leave 127 cores unused if you have requested an entire node on Anvil.

(c) (3 points) The goal of this last part is to compute the integral (2), with the function f given by Eq. (4), to about 1 digit of accuracy. This will require a large value of N and your code has been efficiently parallelized.

Run your code on Anvil using the number of threads you find to be best suited for the job (this will depend on your scalability results). Plot or report N vs $\hat{I}(N)$ for increasingly large values of $N = 4^k$, where $k = 2, 4, 5, \ldots$. Go to as high of a value of N as you need to compute the integral to 1 digit of accuracy. From your data, what do you think is the integral's value and how accurate is its value?

HINT: You might find it useful to compare with my code in a simpler test case. Consider integrating this function on a 10-dimension box $[-.5, .5]$ (sides are length 1). For this smaller problem, I find the integral's value to be about 3.1×10^{-11} which took about 7 minutes using more than 17 billion MC points. The integral was converged by about 1 billion points.

NOTE: You may need your job to run longer than a few minutes. This can be specified in your batch script (recall the file "anvil.job"). How long should you request? You should estimate the expected runtime using timing data from part b to inform your choice and using the fact that the runtime should be proportional to N. Always include a small fudge factor – if you estimate 2 hours, ask for 4 hours. Do not ask for more than you need (your job is more likely to run if you ask for smaller amounts), and **if you find your estimates are longer than 3 hours please email me before submitting the job!!!**

Bonus: (*+10 points on the exam; Needed for an A+*) Add an *error estimator* to your code.

Recall we expect the Monte Carlo (absolute or relative) error decreases as $E(N) \propto N^{-1/2}$. Continue to let $k = 2, 4, 5, \ldots$. Suppose you use 4^{k+1} random samples, then we have

$$E(4^{k+1}) \propto \frac{1}{\sqrt{4^{k+1}}} = \frac{1}{\sqrt{4 \times 4^k}} = \frac{1}{2}\frac{1}{\sqrt{4^k}} \propto \frac{1}{2}E(4^k)\,. \tag{5}$$

Because $\hat{I}(4^{k+1})$ is expected to be twice as accurate as $\hat{I}(4^k)$, we can use it to compute an *estimate* for the error $\left|\hat{I}(4^k) - I\right|$:

$$\left|\hat{I}(4^k) - I\right| \approx \left|\hat{I}(4^k) - \hat{I}(4^{k+1})\right|\,. \tag{6}$$

This allows you to estimate what you would like to know but cannot compute (the error $\left|\hat{I}(4^k) - I\right|$) from data which you have available ($\hat{I}(4^k)$ and $\hat{I}(4^{k+1})$).

Add the error estimator Eq. (6) to your program by following these steps.

(a) (2 points) Starting with your serial code (problem 1), modify your code to output the value $\left|\hat{I}(4^k) - \hat{I}(4^{k+1})\right|$ along with the value of the integral $\hat{I}(4^k)$. Your program should output the intermediate values of $\hat{I}(i)$ (the approximate value of the integral using i random samples) after every $i = 4^k$ iterations, where $k = 2, 4, 5, \ldots$. For example, if you let $N = 256$, your code should output the approximate value of the integral after 16, 64, and 256 iterations *without recomputing past values*. That is, if you have already computed $\hat{I}(16)$ do not "restart" the computation – instead compute the remaining $48 = 64 - 16$ Monte Carlo samples to obtain $\hat{I}(64)$. We choose this sequence to output for reasons mentioned above.

(b) (2 points) Rerun the test in problem 1, part b. On a single figure, plot the actual error, $\left|\hat{I}(4^k) - 2^{10}\right|$, the estimated error, and the theoretically expected error model. Consider up to large values of 4^k.

(c) (6 points) Now extend your OpenMP parallelized program to output the integral's value every $i = 4^k$ iterations, where $k = 2, 4, 5, \ldots$. Have one of the threads output this number, not all of them – this will also require coordination among the team of threads. There are many ways to implement this. Don't be afraid to restructure the code if it allows for an easier implementation of OpenMP parallelism. Demonstrate the code is working by reporting on the test problem. Here is example output from my code running with 4 threads:

```
>>> ./mcOMP_optimized 1073741824 4
MC points = 4, integral = 3.491e+02, err est (for i = 0) = 6.154e+02
MC points = 16, integral = 1.313e+03, err est (for i = 4) = 1.998e+02
MC points = 64, integral = 1.507e+03, err est (for i = 16) = 7.932e+02
MC points = 256, integral = 1.141e+03, err est (for i = 64) = 2.107e+02
```

```
MC points = 1024, integral = 1.020e+03, err est (for i = 256) = 1.034e+02
MC points = 4096, integral = 1.045e+03, err est (for i = 1024) = 1.142e+01
MC points = 16384, integral = 1.014e+03, err est (for i = 4096) = 1.067e+01
MC points = 65536, integral = 1.009e+03, err est (for i = 16384) = 9.849e+00
MC points = 262144, integral = 1.012e+03, err est (for i = 65536) = 2.292e-01
MC points = 1048576, integral = 1.020e+03, err est (for i = 262144) = 1.872e+00
MC points = 4194304, integral = 1.024e+03, err est (for i = 1048576) = 6.464e-01
MC points = 16777216, integral = 1.023e+03, err est (for i = 4194304) = 1.051e+00
MC points = 67108864, integral = 1.024e+03, err est (for i = 16777216) = 2.052e-01
MC points = 268435456, integral = 1.024e+03, err est (for i = 67108864) = 2.295e-02
MC points = 1073741824, integral = 1.024e+03, err est (for i = 268435456) = 5.599e-02

>>> ./mcOMP_optimized 262144 4
MC points = 4, integral = 3.491e+02, err est (for i = 0) = 6.154e+02
MC points = 16, integral = 1.313e+03, err est (for i = 4) = 1.998e+02
MC points = 64, integral = 1.507e+03, err est (for i = 16) = 7.932e+02
MC points = 256, integral = 1.141e+03, err est (for i = 64) = 2.107e+02
MC points = 1024, integral = 1.020e+03, err est (for i = 256) = 1.034e+02
MC points = 4096, integral = 1.045e+03, err est (for i = 1024) = 1.142e+01
MC points = 16384, integral = 1.014e+03, err est (for i = 4096) = 1.067e+01
MC points = 65536, integral = 1.009e+03, err est (for i = 16384) = 9.849e+00
MC points = 262144, integral = 1.012e+03, err est (for i = 65536) = 2.292e-01
```