Homework #3—EAS 520 & DSC 520 & MTH 499

Assigned: Monday, November 6, 2017
Due: Tuesday, November 21, 2017

Reading & viewing: If you would like additional experience with OpenMP, please consider viewing all or part of `https://www.youtube.com/watch?v=6FMn7M5jxrM` and consulting `https://computing.llnl.gov/tutorials/openMP/`

On every homework you turn in, be sure to upload the computer code you used to generate your solutions to Bitbucket (that is, commit your code to the local git repository and push these changes to Bitbucket). Put all the figures and answers asked for in all the questions into a single well organized PDF document (prepared using LaTeX) and upload it onto the myCourses website.

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

1. Write the code in serial
2. Parallelize the code using OpenMP
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.

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

$$V = \int_R \mathrm{d}x\,,$$

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 also evaluated 2-dimensional integrals to compute a so-called Bayesian evince factor for a 2-dimensional model (please see lecture notes for more).

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

$$\mathcal{Z} = \int_R \mathcal{L}(x) d^{10}x\,. \tag{2}$$

where $\mathcal{L}$ 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: (*30 points*) Writing the serial code

(a) Implement your Monte Carlo integration algorithm in C, C++ or Fortran. For full credit, your program should

1. Accept N as input and output the approximate value (1), which I will denote by $\hat{I}(N)$, computed by the Monte Carlo method.
2. Your program should also 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 $16, 64, 256$ iterations. The reason for outputting values after $i = 4^k$ iterations is explained later on, but it should be somewhat obvious: You cannot output all values (which would slow down your program and lead to large data files), while outputting just the final value throws away too much information.
3. Always output the very final approximate value $\hat{I}(N)$ since this is expected to be the most accurate, and your program has worked hard to compute it!

NOTE: As discussed in class, 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.

HINT: You are welcome to start with the code I wrote to solve one of the previous homework problems.

HINT: It will be easier (and in the long run better) for you to write a code which works for an arbitrary number of dimensions instead of specializing it to 10. To do this you should use arrays whenever possible. Lab 8 gives a hint on this. For example, the random variable x could be declared as:

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

(b) Test your code on a case you know the answer to. For example, let

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

then the exact paper-and-pencil value of the integral is 0 (can you see why?). Provide evidence that your code is working by showing the Monte Carlo estimate of the integral is converging to 0.

HINT: Be mindful of floating-point errors. There will be lots of arithmetic here, and the result of these computations will be a value close to 0. Recall an earlier lecture/homework where catastrophic cancellation was problematic. For this problem, you'll want to avoid adding/subtracting large numbers: multiply the "cumulative sum" by V only when you want to output the integral's value $\hat{I}(i)$.

(c) 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

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

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 $\left|\hat{I}(4^k) - I\right|$:

$$\left|\hat{I}(4^k) - I\right| \approx \left|\hat{I}(4^k) - \hat{I}(4^{k+1})\right|\,. \tag{5}$$

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 $\hat{I}(4^{k+1})$).

1. Add the error estimator Eq. (5) to your code. That is, have your code output the value $\left|\hat{I}(4^k) - \hat{I}(4^{k+1})\right|$ along with the value of the integral $\hat{I}(4^k)$.

2. Rerun the test in part b. On a single figure, plot the actual error, $\left|\hat{I}(4^k) - 0\right|$, the estimated error, and the theoretically expected error model. Consider up to large values of 4^k.

Problem 2: (*35 points*) Parallelizing your code.

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

(a) Use OpenMP to parallelize the for-loop over the number of random samples. That is, each thread should make independent random draws, independently evaluate $\mathcal{L}(x)$ at the random value, and update the sum. Updating the sum will require coordination between threads! For full credit, your program **must** continue to output the integral's value and the error estimate 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.

HINT: Be careful of race conditions.

HINT: 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.

(b) 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. 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).

NOTE: The walltime is a computer science term. It is the time elapsed from a "clock on the wall". This is the time from the moment you execute the program to the moment its finished. The term is used to differentiate from the *cputime*.

(c) Rerun the test given in problem 1 to show the code still works when using multiple threads. Provide evidence that the code works (produce figures or provide output and compare to the serial version). Use OpenMP routines to output the number of threads and provide evidence that multiple threads are indeed being used.

Problem 3: (*35 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 Stampede2.

(a) Modify your OpenMP parallelized code to let

$$\mathcal{L}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) = \exp\left(-\sum_{i=1}^{9}\left[(1 - x_i)^2 + 100\left(x_{i+1} - x_i^2\right)^2\right]\right), \tag{6}$$

which is the higher dimensional extension of the 2-dimensional function you used in homework 2.

(b) Propose a simple test that you have correctly implemented the equation. That is, what might you try doing to check there's no mistakes in your code for Eq. (6). Perform this test.

(c) Compile and run the code for a small test case, say $N = 1024 = 4^5 = 2^{10}$ (which suggests about 2 sampling points per dimension in our 10-dimensional space). With so few points, the value should be far from the true value.

(d) You will now run your code on Stampede2 using a single compute node (either submit to the job queue or run your program iteratively). 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). The way you update shared variables among the threads is also important (e.g. critical, atomic, reduction, etc [see lab 7]). When reporting your scalability results, mention the choices your made and what attempts (if any) you explored to improving the code's scaling performance.

1. *strong scaling test*: keep the problem sized fixed (you hold N constant) and let the number of threads increase. Let $N = 10^6$ and compute the code's speedup using 1, 2, 4, 8, 16, 32, and 64 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 Stampede using a single compute node (submit with sbatch). Let $N = \text{threads} \times 10^5$ and compute the code's speedup using 1, 2, 4, 8, 16, 32, and 64 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 63 cores unused.

(e) The goal of this last part is to compute the integral (2), with the likelihood function $\mathcal{L}$ given by Eq. (6), to about 1 digit of accuracy. This may be difficult. In fact this problem comes from a recently published research paper and so we (the class) are trying to reproduce their published result. Please post any issues or difficulties you encounter on Piazza.

Run your code on Stampede2 using the number of threads you find to be best suited for the job (this will depend on your scalability results). Plot or report i vs $\hat{I}(i)$ and the estimated error. Continue to output every $i = 4^k$ iterations, where $k = 2, 4, 5, \dots,$ as well as outputting the final value computed using all N points. Go to as high of a value of N as you need to compute the integral to 1 digit of accuracy. Use your error estimator to inform you of the accuracy. From your data, what do you think is the integral's value and how accurate is its value?

NOTE: You may need your job to run longer than a few minutes. This can be specified in your batch script (recall the file "stampede.job"). How long should you request? You should estimate the expected runtime using timing data from part c and d 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 5 hours please email me before submitting the job!!!**

HINT: To copy data off of stampede, you may want to use the Unix function "scp" or use the git workflow we discussed in class.