I Assignment 4: Parallelization and Relaxation

due: Feb 21, 2017 – 11:59 PM

GOAL: Some problems are too big to be solved by a single core. As a first step into parellel programming, we're going to look at `OpenMP` for two applications: multi-dimensional integration, based on our work on the previous problem set, and one dimensional relaxation, based on new material.

This problem set is split into two parts.

In the first part, we're going to take a first step into parallelization by studying a naively parallelizable problem: multi-dimensional integration. In its naivest form, multidimensional integrals can be broken up recursively into single dimensional integrals. We'll begin with a non-parallel example of a two dimensional integral before parallelizing a multi dimensional integral. Before that, though, we'll write a standard routine: a parallel hello world!

In the second part, we're going to study relaxation. Relaxation comes up in a variety of real world applications, such as electrostatics and, perhaps more physically intuitive, heat flow. Instead of jumping into two dimensions, which we'll do on the next problem set, we're going to start with a one dimensional application: a metal rod being hit by a blow torch.

II Accessing SCC

We may not have this set up yet! If you can't access SCC or our project directory, skip these steps. We'll take care of this soon.

To access `SCC`, open a terminal and run:

```
ssh [usename]@scc1.bu.edu
```

Your username is your BU account name. Once you're on `SCC`, make sure you're in a `bash` shell. If you're in a `bash` shell by default, your command prompt will include your username:

```
[username]@scc1 ~]
```

If that's roughly what it looks like, you're all set. If your default shell is `csh`, it'll look like this:

```
scc1:~ %
```

If your command line looks like this, just run the command `bash` to get to the appropriate shell. Once you're there, let's navigate to our class project's active directory:

```
cd /projectnb/isingmag/
```

Create your own directory with your kerberos username. If you'd like to play around with compiling and running code, you can log into an interactive shell by running:

```
qrsh -l h_rt=1:00:00 -P isingmag
```

That will get you a one core interactive shell for one hour. Of course, it should be pretty clear how to try to get it for longer than an hour... but keep in mind, a longer interactive shell may not be available! This is a shared cluster, after all. There's a wonderful collection of information on BU's IS&T website for running jobs on SCC: `http://www.bu.edu/tech/support/research/system-usage/running-jobs/`.

For the next part of this assignment, find out how to request an interactive shell with **four** cores. We'll need this to test out the most important of all codes: a parallel Hello World! While you should all know how to do this, here's a naïve hello world:

```
#include <stdio.h>

int main(int argc, char** argv)
{
printf("Hello world!\n");
return 0;
}
```

If you saved this to a file `hello.c`, you'd compile it with:

```
gcc hello.c -o hello
```

Next, let's consider a parallel version using OpenMP:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv)
{
int tid, nthreads;

#pragma omp parallel shared(nthreads) private(tid)
{
tid = omp_get_thread_num();
if (tid == 0)
nthreads = omp_get_num_threads();

#pragma omp barrier
printf("Hello world from thread %d of %d!\n", tid, nthreads);
}
return 0;
}
```

Which can be compiled by:

```
gcc omp_hello.c -fopenmp -o omp_hello
```

Note the additional `include` in the code and the added compile flag `-fopenmp`. There's more—we run this code by first defining the number of threads to use. This can be done by a bash environment variable:

```
export OMP_NUM_THREADS=4;
./omp_hello
```

You can see why it's important that we got a parallel interactive thread on SCC! OpenMP works largely through the `#pragma` statements. The line:

```
#pragma omp parallel shared(nthreads) private(tid)
```

instructs the compiler to run the code enclosed below in parallel on each core. The pragma explicitly calls out two variables, `nthreads` and `tid`, as shared and private, respectively. Each thread has its own `tid`, which makes sense—it's the numerical id of the thread! On the other hand, the number of threads will be the same on each thread. Thus, we mark `nthreads` as shared, and only have one thread call the function to find out how many threads there are.

One point of note: we're having *one* thread find out a piece of information that *every* thread eventually uses. That means we need all of the threads to sync up until that one thread has populated shared memory with `nthreads`. That's what the line

```
#pragma omp barrier
```

is for. After the barrier, we have every thread print its name, and carry on!

III Parallel Integration

For the first real problem, we're going to perform some multidimensional integrals. As a quick refresher, a single dimensional integral of a function $f(x)$ from -1 to 1 is written as:

$$\int_{-1}^{1} f(x)\ dx \tag{1}$$

On the other hand the two dimensional integral of a function $g(x, y)$ inside the square defined by $-1 < x < 1, -1 < y < 1$ can be written as:

$$\int_{x\in[-1,1]y\in[-1,1]} g(x, y)\ dA \tag{2}$$

I'm sure this is different from how you may have seen it in a multivariable calculus class. It's actually a non-trivial proof that this is equivalent, but a much clearer (and likely familiar) formulation is:

$$\int_{-1}^{1} dy \int_{-1}^{1} dx \; g(x,y) \tag{3}$$

Now that the function is written in this form, it becomes a bit easier to now see how we'll numerically approximate it:

$$\int_{-1}^{1} dy \int_{-1}^{1} dx \; g(x,y) \approx \int_{-1}^{1} dy \sum_{i=1}^{N} w_i g(x_i, y) \tag{4}$$

$$\approx \sum_{j=1}^{N} \sum_{i=1}^{N} w_i w_j g(x_i, y_j) \tag{5}$$

This approximation generalizes trivially to even higher dimensions.

III.1 Integrating a few more functions

This problem should seem familiar: we're going to integrate a few problems on the interval $[-1, 1]$, except in multiple dimensions. Write a main program `test_integrate_2d.c` that performs the following three integrals using the Trapezoidal rule and Gaussian integration for $N = 2$ to 24 in the sum:

$$\int_{-1}^{1} dy \int_{-1}^{1} dx \; x^8 + y^8 + (y-1)^3(x-3)^5 = ? \tag{6}$$

$$\int_{-1}^{1} dy \int_{-1}^{1} dx \; \sqrt{x^2 - y^2 + 2} = ? \tag{7}$$

$$\int_{-1}^{1} dy \int_{-1}^{1} dx \; e^{-x^2 - \frac{y^2}{8}} \cos(\pi x) \sin(\frac{\pi}{8} x) = ? \tag{8}$$

$$\tag{9}$$

Next, let's try a 4 dimensional integral:

$$P(m) = \int_{-1}^{1} dp_0 \int_{-1}^{1} dp_1 \int_{-1}^{1} dp_2 \int_{-1}^{1} dp_3 \frac{1}{\left(\sin^2(\pi p_0/2) + \sin^2(\pi p_1/2) + \sin^2(\pi p_2/2) + \sin^2(\pi p_3/2) + m^2\right)^2} \tag{10}$$

What is this integral anyway? Glad you asked! It is called a one loop Feynman diagram. It is related to the probability[1] in quantum mechanics of two particles (of mass m), starting at the same

[1] Emphasis on *related*, so don't be worried if it's greater than 1.

point, performing a random walk in space and time, and ending up eventually together at the same place at the same time! To get this random walk into this cute integral, we first put all of the space-time on an infinite 4D lattice and let the particle hop from point to point. Using the magic of Fourier transform we convert to velocities $v_i = p_i/m$b and energy $E = p_0$. Lots of applications in engineering materials and high energy physics need to do such integrals. Each time you add one particle you get another 4 dimensional integral. These are real computational challenges.

As a first step, integrate this for $m = 1$. Now if you want an accurate result from Gaussian integration there are a lot of sums, e.g. $N = 16$ implies $16^4 = 65536$ terms. You'll want to parallelize this integral. Performing an integral is an example of a *reduction*, where we are summing a series of numbers. Reductions can be finicky in parallel code due to race conditions—in a naïve implementation, multiple threads can try to accumulate a sum variable at the same "time". Thankfully, OpenMP can handle this. Here's an example of summing a one dimensional vector in OpenMP:

```
double sum = 0.0; #pragma omp parallel for shared(v1) reduction (+:sum)
for(int i = 0; i < SIZE; i++)
sum = sum + v1[i];
```

In this example, `sum` is the variable we accumulate into and `v1` is the array we're summing over. All of the threads access `v1`, which is why it's noted as shared. We also need to tell OpenMP what variable we're reducing into, thus the part `reduction (+:sum)`, where `sum` is the variable name (which could be anything, such as `wheelbarrel`, and the `+` indicates what operation we're performing (it could be `-`, `*`, etc).

For a four dimensional integral, you should have four nested `for` loops (unless you want to try to solve this by recursion, in which case you'll still have an outer `for` loop). Include an appropriate `pragma` to parallelize the outer loop. Try this for a range of m: for large m, the probability of them ever meeting is small, while for lighter particles, there is a much higher chance they'll meet... and the integral becomes more difficult. Scan over m from $\frac{1}{\sqrt{10}}$ to 10 in steps of $10^{-1/6}$ (so 10 different values).[2] You'll see some inconsistency as a function of N for very small m—what's the reason for this? Don't be worried about it, ultimately. While it's not perfect, you should see a scaling $P(m) \sim -\log(m)$ at small m: make a plot showing this. Remember to set your axes properly to make it clear!

Write a main program `test_integrate_feynman.c` where you perform this integral in parallel, and create a plot `integral_scaling.pdf` (or whatever appropriate extension your version of `gnuplot` supports) showing the log behavior described above.

IV Solving the one dimensional Laplace equation

Let's start with solvers by beginning with the simplest Jacobi, Gauss Seidel and Red/Black (or even/odd) iterations. The use of relaxation parameters. The next part is a gentle introduction to Multigrid. The project on linear algebra will explore the use of these and comparison with the classic Conjugate Gradient method.

[2] In the original version of the assignment, we had you scan over m from 0.0001 to 100, steps of $\sqrt{10}$, which gave poor results for small m.

Let us consider first a 1D heat condition problem. As describe above the equation we can consider a variable and a function The Laplace equation for a function $\phi(x)$ of a real variable $x \in [a, b]$ and put it on a grid with spacing h:

$$-\frac{d^{\phi}(x)}{dx^2} = b(x) \rightarrow -\frac{2\phi[i] - \phi[i-1] - \phi[i+1]}{h^2} = b[i] \tag{11}$$

Let's take this to model temperature in the cold night with heater in the midle of the 1D room. The walls are set to temperatures T_1 and T_2. For example we can take grid from $i = -N, \cdots N$ and set the walls to absolute zero $T_1 = 0$ and $T_2 = 0$ with a single heater in the middle of the room! (I guess you are in outer space on linear room. Now x is an integer! The matrix `A T = b` equation is

$$T[x] - \frac{1}{2}[T[x-1] + T[x+1]] = \alpha h^2 \delta_{x,0} \tag{12}$$

with $T[N] = T[-N] = 0$. Actually we know the solution!

$$T[x] = (N \pm x)\alpha h^2/(2N) \quad \text{for} \quad \pm x \geq 0 \tag{13}$$

Since this is exact, we can even look at for $h \rightarrow 0$. But now we need to go back to real x's and let the room have width $W = 2Nh$. So $T(x) = \alpha(W - |x|)/2W$ is the exact solution to

$$-\frac{d^2 Ti(x)}{dx^2} = (\alpha/2)\delta(x) \tag{14}$$

whatever `\delta(x)` means. Let's see how it works by just substituting it in!

$$-\int_{-W}^{W} \frac{d^2 T(x)}{dx^2} = -\frac{dT(x)}{dx^2}\Big|_{-W}^{W} = 2W \equiv \alpha \int_{-W}^{W} \delta(x) \tag{15}$$

Lets use this as the definition! (Physicists and engineers cheat this way all the time! Actually it is not cheating it is smart. [3])

IV.1 Coding up a one-dimensional Laplace solver

Write a program to solve the 1D finite difference heat equation described above using the Jacobi, Gauss-Seidel, and Red/Black iterative solvers. We practically wrote this for you in the 2D case—you just have to modify the boundary terms and turn it into a 1D problem. Remember, each algorithm is nearly identical, with only a few changes in the iterative loop! You will want to relabel the arrays defined above as `double T[2N +1]` with indices $T[i]$ with $i = 0, ..., 2N$ so now the boundary values are $T[0] = T[2N] = 0$ $i = 0, i = 2N$. The source now lives at the mid-point, $i = N$.

The program should use all three algorithms, starting with the initial guess $T[i] = 0$ for all i, and print to file the value of the residual $r = b - AT$ at each iteration, that is:

$$r[i] = b[i] - AT[i] = b[i] - (T[i] - \frac{1}{2}[T[i-1] + T[i+1]]) \tag{16}$$

[3] See comment in in very interesting book *The history of Pi* by Petr Beckman that "What especially outranged the mathematicians was not so much that electrical engineers continued to use it (e.g. delta functions) but that it would almost always supply the correct result"

The program should stop iterations when each method reach single precision convergence:

$$\frac{\sqrt{\sum_{i=1}^{2N-1} r[i] * r[i]}}{\sqrt{\sum_{i=1}^{2N-1} b[i] * b[i]}} < 10^{-6} \tag{17}$$

The goal is to compare how each converges using the residual relative to each other for large N and to compare the scaling of each to reach this cut-off as a function of N. For simplicity, assume $\alpha h^2 = 1$ (but a smart programmer would leave this as a variable that can be set at compile or run-time).

As a next step, parallelize the code! As we discussed, Gauss-Seidel can't be parallelized well, so we'll skip that. Instead, just parallelize the Jacobi and Red/Black implementations using the appropriate OpenMP pragmas. Don't forget to parallelize the reductions when you check for convergence, too! Name your parallelized source file `relaxation_parallel.cpp`. While you don't need to provide another plot, make sure it does indeed give identical results to the non-parallel code!

The deliverables for this exercise are:

- The source files `relaxation.cpp` and `relaxation_parallel.cpp` as described above.
- Three plots of the residual as a function of iteration number, one for each method. You should do this for $N = 10000$. Think about your y-axis—what is more instructive, a linear or a logarithmic plot?
- One plot which shows the iteration count for all three methods as a function of N. Vary N from 10 to 10000. Again, think about your axes—should they be linear or log?

V Submitting Your Assignment

This assignment is due at 11:59 pm on Tuesday, February 21. Please e-mail a **tarball** containing the assignment to the class e-mail, bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

```
tar -cvf evan_weinberg_asgn4.tar [directory with files]
```

You may want to include other files. **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.