Spring 2023: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 1, due Feb, 12

This assignment requests that you do some background reading and tell us about it in a mini presentation. Sign up for your presentation day on this Google-doc (you need to login with your NYU account to edit the document). For the simple warmup coding problems, please hand in your results and listings of your solutions on Brightspace. For future homework assignments we will use Git, but for now Brightspace will do.

1. **Describe a parallel application, a computing myth or a background topic in class.** Find and examine an application problem for which high-performance computing has been used, or a background topic related to HPC. Please see the list of topic suggestions on Brightspace under *Reading/proposals for mini presentations* for suggestions; as soon as you have picked a topic, enter your name and the topic (e.g., Application paper from SC18 on solar wind modeling, or computing myth #5)in the above Google document. Each topic can only picked once, so first come, first serve. You can also pick a topic from your own research or find one elsewhere, but please check with me if you do that. Prepare 2-3 slides of the problem. For applications and papers, describe where and how successful high-performance computing has been/is used in a ~ 3 minute class presentation (I will cut you off after 4 minutes!). For applications, please consider to include the following:

   (a) What's the application problem being solved?

   (b) Why does the problem require large/fast computation?

   (c) What are the underlying algorithms?

   (d) If the application uses a supercomputer, where is that computer on the Top500 list (`http://www.top500.org/`)? Say a few words about the kind of architecture.

   (e) What is being said about the performance of the algorithm?

   Make an effort in creating your 2-3 slides and in your presentation–we all want to learn something. To avoid connecting and disconnecting of computers, drop your presentation as PDF into this Google-folder. To avoid a mess in the directory, please use `LASTNAME_FIRSTNAME.pdf` as name for your slides (and please leave them there after your presentation, this is part of your submission for the first assignment).

2. **Matrix-matrix multiplication.** We will experiment with a simple implementation of a matrix-matrix multiplication, which you can download from `https://github.com/NYU-HPC23/homework1/`. We will improve and extend this implementation throughout the semester. For now, let us just assess the performance of this basic function (this is the implementation with computational intensity of 2, i.e., about 2 flops per slow memory access). Report the processor you use for your timings[1]. For code compiled with different optimization flags (`-O0` and `-O3`) and for various (large) matrix sizes, report

   - the flop rate,
   - and the rate of memory access.

[1]On Linux you can use the command `cat /proc/cpuinfo`, on a Mac you can use the Apple menu or the command `sysctl -n machdep.cpu.brand_string`.

3. **Write a program to solve the Laplace equation in one space dimension.** For a given function $f : [0,1] \to \mathbb{R}$, we attempt to solve the linear differential equation

$$-u'' = f \text{ in } (0,1), \text{ and } u(0) = 0, u(1) = 0 \tag{1}$$

for a function u. In one space dimension[2], this so-called *boundary value problem* can be solved analytically by integrating f twice. In higher dimensions, the analogous problem usually cannot be solved analytically and one must rely on numerical approximations for u. We use a finite number of grid points in $[0,1]$ and finite-difference approximations for the second derivative to approximate the solution to (1). We choose the uniformly spaced points $\{x_i = ih : i = 0, 1, \dots, N, N+1\} \subset [0,1]$, with $h = 1/(N+1)$, and approximate $u(x_i) \approx u_i$ and $f(x_i) \approx f_i$, for $i = 0, \dots, N+1$. Using Taylor expansions of $u(x_i - h)$ and $u(x_i + h)$ about $u(x_i)$ results in

$$-u''(x_i) = \frac{-u(x_i - h) + 2u(x_i) - u(x_i + h)}{h^2} + \text{h.o.t.},$$

where h.o.t. stands for a remainder term that is of higher order in h, i.e., becomes small as h becomes small. We now approximate the second derivative at the point x_i as follows:

$$-u''(x_i) \approx \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}.$$

This results in the following finite-dimensional approximation of (1):

$$A\boldsymbol{u} = \boldsymbol{f}, \tag{2}$$

where

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}, \qquad \boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}, \qquad \boldsymbol{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}.$$

Simple methods to solve (2) are the Jacobi and the Gauss-Seidel method, which start from an initial vector $\boldsymbol{u}^0 \in \mathbb{R}^N$ and compute approximate solution vectors $\boldsymbol{u}^k$, $k = 1, 2, \dots$. The component-wise formula for the Jacobi method is

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \sum_{j \neq i} a_{ij} u_j^k\right),$$

where a_{ij} are the entries of the matrix A. The Gauss-Seidel algorithm is given by

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \sum_{j<i} a_{ij} u_j^{k+1} - \sum_{j>i} a_{ij} u_j^k\right).$$

[2]The generalization of (1) to two and three-dimensional domains Ω instead of the one-dimensional interval $\Omega = [0,1]$ is the *Laplace equation*,

$$\begin{aligned} -\Delta u &= f \text{ on } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned}$$

which is one of the most important partial differential equations in mathematical physics.

If you are unfamiliar with these methods, please take a look at the Wikipedia entries for the Jacobi[3] and the Gauss-Seidel[4] methods.

(a) Write a program in C/C++ that uses the Jacobi or the Gauss-Seidel method to solve (2), where the number of discretization points N is an input parameter, and $f(x) \equiv 1$, i.e., the right hand side vector $\boldsymbol{f}$ is a vector of all ones.

(b) After each iteration, output the norm of the residual $\|A\boldsymbol{u}^k - \boldsymbol{f}\| := \left(\sum_i ((A\boldsymbol{u}^k)_i - f_i)^2\right)^{1/2}$ on a new line, and terminate the iteration when the initial residual is decreased by a factor of 10^4 or after 5000 iterations. Start the iteration with a zero initialization vector, i.e., $\boldsymbol{u}^0$ is the zero vector.

(c) Compare the number of iterations needed for the two different methods for different numbers $N = 10$, $N = 1000$ and $N = 100,000$. Compare the run times for $N = 100,000$ for 100 iterations using different compiler optimization flags (`-O0` and `-O3`). Report the results and a listing of your program. Specify which computer architecture you used for your runs. Make sure you free all the allocated memory before you exit[5]. Note that for large N, these algorithms need many iterations!

[3]http://en.wikipedia.org/wiki/Jacobi_method

[4]http://en.wikipedia.org/wiki/Gauss-Seidel_method

[5]In the next class I'll show you how to check memory leaks using `valgrind`, which I find a useful tool.