High Performance Computing
Anthony Trubiano
Assignment 1 (due Feb. 11, 2019)

Note: All computations were performed with an Intel Core i7-8750H Processor with base frequency 2.20 GHz. The maximum main memory bandwidth is 41.8 GB/s. At 16 double precision operations per cycle, the theoretical max flop rate would be about 35.2 GFlops/s.

1. **Matrix-matrix multiplication.** We will experiment with a simple implementation of a matrix-matrix multiplication, which you can download from the `homework1` directory in https://github.com/NYU-HPC19/. We will improve and extend this implementation throughout the semester. For now, let us just assess the performance of this basic function. Report the processor you use for your timings. 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.

	Flag -O0		Flag -O3	
N	Flop Rate	Memory Rate	Flop Rate	Memory Rate
200	0.215	3.44	1.16	18.55
220	0.214	3.43	1.14	18.23
240	0.208	3.33	1.11	17.77
260	0.214	3.42	1.14	18.17
280	0.214	3.43	1.13	18.10
300	0.214	3.43	1.13	18.06
320	0.179	2.86	1.10	17.61
340	0.214	3.43	1.12	18.02

Table 1: The measured flop rate (GFlops/s) and memory access rate (GB/s) for the matrix-matrix multiplication code, MMult0.cpp, ran on $N \times N$ matrices, using two different compiler optimization flags.

   From the table, we see a relatively consistent estimate of the flop rate and memory access rate. Using the O3 flag, we get a flop rate of about 3% of the theoretical max and a memory access rate of about half of the max bandwidth. This is expected since the computation is memory limited. We note a dip at $N = 320$.

2. **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[1], this so-called *boundary value problem* can be solved analytically by integrating f twice. In higher dimensions, the analogous problem usually cannot be

[1] 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.

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 $\{x_i = ih : i = 0, 1, \ldots, N, N+1\} \subset [0,1]$, with $h = 1/(N+1)$, and approximate $u(x_i) \approx u_i$ and $f(x_i) \approx f_i$, for $i = 0, \ldots, N+1$. 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 $\boldsymbol{u}^k$, $k = 1, 2, \ldots$. 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

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

If you are unfamiliar with these methods, please take a look at the Wikipedia entries for the Jacobi[2] and the Gauss-Seidel[3] methods.

(a) Write a program in 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 $\|A\boldsymbol{u}^k - \boldsymbol{f}\|$ on a new line, and terminate the iteration when the initial residual is decreased by a factor of 10^6 or after 5000 iterations. Start the iteration with a zero initialization vector, i.e., $\boldsymbol{u}^0$ is the zero vector.

[2] http://en.wikipedia.org/wiki/Jacobi_method
[3] http://en.wikipedia.org/wiki/Gauss-Seidel_method

(c) Compare the number of iterations needed for the two different methods for different numbers $N = 100$ and $N = 10,000$. Compare the run times for $N = 10,000$ 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.

A C++ code that uses either Jacobi or Gauss-Seidel (comment out whichever you are not using before compiling) iterations to solve the problem $-u''(x) = 1$ on the interval $[0, 1]$ with zero Dirichlet boundary conditions was written. The code takes in the number of interior grid points, N, and the max number of iterations as input. Iterations terminate when the residual has been reduced by a factor of 10^6 or after 5000 iterations.

Running the codes, I find neither algorithm reduces the residual to within tolerance in 5000 iterations (is this an error? I imagine no, since the condition number is pretty large). For $N = 100$, the Jacobi method terminates in 28347 iterations while the Gauss-Seidel method terminates after 14174 iterations, twice as fast. Increasing to $N = 10000$, the residual isn't even reduced by a factor of 10 after 100000 iterations. This is not surprising, since the condition number of this matrix grows like N^2, which is huge here.

Next, we set $N = 10000$ and compute the time taken (in seconds) for 100 iterations of each method, using two different compiler flags. The results are summarized in the following table.

Method	Flag -O0	Flag -O3
Jacobi	0.028	0.017
Gauss-Seidel	0.029	0.017

Table 2: Time taken (in seconds) to perform 100 iterations using each iterative method an a square matrix of size $N = 10000$, for different compiler flags.

We see that both methods take a similar amount of time run, and using the the 03 optimization flag gives nearly a 2x speed up. We should expect this, as both methods perform the same number of flops. In terms of memory, Gauss-Seidel can re-write elements to the same vector, so it is half as expensive as Jacobi, and also converges faster.