

**Spring 2022: Advanced Topics in Numerical Analysis:  
High Performance Computing  
Assignment 1 (due Feb. 14, 2022 before class)**

0. **Instructions for submitting this homework** Upload a PDF with your answers to the questions to NYU brightspace. Additionally, upload a zip archive with your code. In total, you will upload 2 attachments.
1. **Describe a parallel application and the algorithms used.** Find and examine an application problem for which high-performance computing has been used. Pick a problem from your own research, or find a problem elsewhere. Prepare a 1–2 page description of the problem and describe where and how successful high-performance computing has been/is used. 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) How well does the algorithm perform? Does it “scale”?

If you are looking for an application, take a look at the papers from one of the previous Supercomputing conferences.<sup>1</sup> Alternatively, take a look at the National Science Foundation (NSF)-funded supercomputing centers. These centers provide computing resources for open research under the Extreme Science and Engineering Discovery Environment (XSEDE)<sup>2</sup> and the website usually has science stories with links to papers. Here are some direct links to US-based computing centers.<sup>34567</sup>

2. **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/pehersto/HPCSpring2022/>. 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.
3. **Write a program to solve the Laplace equation in one space dimension.** For a given function  $f : [0, 1] \rightarrow \mathbb{R}$ , we attempt to solve the linear differential equation

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

---

<sup>1</sup><http://supercomputing.org/history.php> and choose “Conference Proceedings”.

<sup>2</sup>XSEDE: <https://www.xsede.org/>

<sup>3</sup>Oak Ridge National Laboratory: <https://www.olcf.ornl.gov/>

<sup>4</sup>National Energy Research Scientific Computing Center (NERSC): <https://www.nersc.gov/>

<sup>5</sup>San Diego Supercomputing Center: <http://www.sdsc.edu/>

<sup>6</sup>Nasa Advanced Supercomputing Division: <http://www.nas.nasa.gov/>

<sup>7</sup>Texas Advanced Computing Center (TACC): <https://www.tacc.utexas.edu/>

for a function  $u$ . In one space dimension<sup>8</sup>, 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):

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

where

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

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

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

If you are unfamiliar with these methods, please take a look at the Section 11.2 in *Matrix Computations* by Golub and Van Loan.

---

<sup>8</sup>The generalization of (1) to two and three-dimensional domains  $\Omega$  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.

- (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  $\mathbf{f}$  is a vector of all ones.
- (b) After each iteration, output the norm of the residual  $\|A\mathbf{u}^k - \mathbf{f}\|$  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.,  $\mathbf{u}^0$  is the zero vector.
- (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.