Major Project: Parallelizing Strassen's Matrix-Multiplication Algorithm

The product C of two matrices A and B of size n x n can be computed in $O(n^3)$ operations using the *standard algorithm* in which each element of the product is obtained by computing the inner product of a row vector of A and a column vector of B by employing 2n operations (additions and multiplications between scalars).

```
for i = 1, … , n
        for j = 1, … , n
                for k = 1, … , n
                                C_ij = C_ij + A_ik B_kj,
```

Volker Strassen proposed a recursive algorithm to compute the product in $O(n^{2.8})$ operations, and opened the door to the creation of several algorithms that reduced the complexity further. An interesting implication of this work is that the solution of a linear system of order n can be obtained in $O(n^{2.8})$.

Consider the following partitioning of the matrices into *equal sized* blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},\ C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, \tag{1}$$

which leads to the straightforward approach to compute C:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{11} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}, \tag{2}$$

that requires 8 multiplications and 4 additions among matrices of size n/2 x n/2.

Strassen's algorithm computes the product as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}, \tag{3}$$

where

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}), \qquad M_2 = (A_{21} + A_{22})B_{11}, \qquad M_3 = A_{11}(B_{12} - B_{22}),$$
$$M_4 = A_{22}(B_{21} - B_{11}), \qquad M_5 = (A_{11} + A_{12})B_{22},$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}), \qquad M_7 = (A_{12} - A_{22})(B_{21} + B_{22}). \tag{4}$$

Compared to the algorithm in (2), Strassen's approach requires computing only 7 matrix-multiplications, although the number of matrix-additions increases to 18. Since matrix-additions are $O(n^2)$ whereas multiplications are $O(n^3)$, a recursive algorithm using this strategy results in an asymptotic complexity of $O(n^{\log 7})$ which can be approximated by $O(n^{2.8})$.

Strassen's *recursive* algorithm applies the idea outlined in (3) and (4) to compute the multiplications for each $M_i, i = 1, \ldots, 7$ recursively. Furthermore, instead of going all the way down to 1 x 1 matrices, you can terminate the recursion after k' levels ($k > k' > 1$) when you reach matrices of size s x s ($s = n/2^{k'}$), and use the standard algorithm to compute matrix-multiplication among these terminal matrices.

1. (75 points) In this project, you have to develop a parallel implementation of Strassen's recursive algorithm for computing matrix-multiplications. You should choose **one** of the following strategies.
    a. Develop a shared-memory code using OpenMP, or
    b. Develop a CUDA-based code for the GPU.

    Your code should compute the product of two matrices of size n x n, where $n=2^k$, and k is an input to your program. Your code should also accept k' as an input to allow the user to vary the terminal matrix size.

2. (25 points) Develop a report that describes the parallel performance of your code. You will have to conduct experiments to determine the execution time for different values of k, and use that data to plot speedup and efficiency graphs. You should also experiment with different values of k' to explore its impact on the execution time.

    Discuss any design choices you made to improve the parallel performance of the code and how it relates to actual observations. Include any insights you obtained while working on this project.

    Lastly, include a brief description of how to compile and execute the code on platform you have chosen.

Submission:

Upload two files to Canvas:

1. A **single zip file** consisting of the code you developed.
2. The project report as a single PDF or MSWord.

Helpful Information:

OpenMP

1. Information for OpenMP code development and execution was listed in earlier assignments.

GPU

2. Information on compiling and running CUDA programs on a GPU for Grace is available in the HPRC user guide (See https://hprc.tamu.edu/wiki/Grace).
3. To develop code interactively, log on to one of the GPU-equipped login nodes on Grace; these nodes are named graceX.hprc.tamu.edu, where X is to be replaced by the numbers 1, 2, or 3 (See https://hprc.tamu.edu/wiki/Grace:Intro#Login_Nodes). If you use portal.hprc.tamu.edu to access Grace, you are randomly assigned a node where X can be 1-5. You will then need to use ssh to log into a GPU-equipped login node from the initial shell.
4. Check out https://hprc.tamu.edu/wiki/Terra:Compile:All#CUDA_Programming on how to compile and execute your code. Current recommendations are listed below.
    - Load the modules for compiler and CUDA prior to compiling your program. Use:

      ```
      module load intel CUDA
      ```

- Compile C programs using `nvcc`. For example, to compile `code.cu` to create the executable `code.exe`, use

  ```
  nvcc -ccbin=icc -o code.exe code.cu
  ```

5. The run time of a code should be measured when it is executed in dedicated mode. Sample batch files for Grace are available at https://hprc.tamu.edu/wiki/Grace:Batch#Job_File_Examples. To execute the code on a GPU-equipped node, you must use the submission options for GPUs.