HW3: Parallelizing Strassen's Matrix-Multiplication Algorithm via OpenMP

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 *leaf* matrices.

You are provided with a C++ program `strassen_omp.cpp` that includes code to compute the product of two matrices using the Strassen's algorithm as well the standard method.

To compile and execute the code, use the commands:

```
module load intel
icpx -o strassen_omp.exe -qopenmp strassen_omp.cpp
./strassen_omp.exe <k> <q>
```

where <k> and <q> are integer arguments that specify the size of the matrix (2^k) and the size of the leaf matrices (2^q). The output of a sample run is shown below.

```
./strassen_omp.exe 10 4
Matrix size = 1024, Leaf matrix size = 16, Strassen's (s) = 2.3122 s,
Standard = 3.3629 s, Error = 0
```

Note that in its present form, the code is not parallelized.

1. (75 points) Parallelize the code by inserting OpenMP directives to obtain a parallel implementation of Strassen's recursive algorithm.
2. (25 points) Determine the speedup obtained by your code on a single node of Grace using all available cores for matrixes of size 2^k for k = 10,…,14. Speedup should be computed as the speed improvement over Strassen's algorithm using only a single thread. Experiment with the size of the leaf matrix 2^q to determine which size(s) give you the maximum speedup. Summarize your findings in a document that includes speedup and efficiency graphs as well as your insights into the results you have obtained. Lastly, include a brief description of how to compile and execute the code on Grace.

Submission:

Upload two files to Canvas:

1. A **single zip file** consisting of the code you developed.
2. Submit a single PDF or MSWord document with your response for Problem.

Helpful Information:

1. You may use Grace for this assignment.
2. Load the compiler module prior to compiling your program. Use:
   `module load intel`
3. Compile C++ programs with OpenMP pragmas using `icpx` with the switch `-qopenmp`. For example, to compile `code.cpp` to create the executable `code.exe`, use
   `icpx -qopenmp –o code.exe code.cpp`
4. The run time of a code should be measured when it is executed in dedicated mode. Create a batch file and submit your code for execution via the batch system on the system.