UCLouvain EPL-INMA 18 juin 2025	**Exam : LINMA2710** Scientific Computing	First name : Last name : NOMA :

Important instructions:

- The exam duration is 2 hours.
- Please start your answer just below each question (when possible) and continue on a new sheet of paper if necessary. Put your name on all sheets.
- As mentioned in the course outline, only personal handwritten notes and annotated printouts of the material posted on the course Moodle page are allowed. All the rest, including electronic devices (laptop, smartphone, etc.), is forbidden.
- Do not use red ink.
- The size of the answer box is chosen with a quite conservative margin so do not worry if your answer is shorter than the box.

UCLouvain EPL-INMA 18 juin 2025	**Exam : LINMA2710** Scientific Computing	First name : Last name : NOMA :

1. Fill the gap between lines 3 and 21 in this C++ code. The code is meant to print "4.4", which is the area of a room of length 2.2 and breadth 2.0.

```
1
2  #include <iostream>
3  using namespace std;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 int main() {
22
23     Room room1;
24
25     room1.length = 2.2;
26     room1.breadth = 2.0;
27
28     cout << "Area of Room =  " << room1.calculate_area() << endl;
29
30     return 0;
31 }
```

2. Write a finite-volume scheme with only three (distinct) vertices for solving in C++ the heat equation on a disk

$$\Omega = \{x \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \leq R^2\}$$

with Dirichlet boundary conditions using a finite volume method. You can choose where to put the vertices; at least one should be on the boundary of Ω and one in the interior of Ω.

Comments: For this question, there is no length limitation, but concise answers (i.e., short and clear) are preferred. There is not a unique adequate way to answer the question, but the answer should at least contain: (i) a paragraph that relates to PDEs, (ii) a paragraph that introduces a numerical method, and (iii) a C++ code. Since you do not have access to a Voronoi library, feel free to restrict the vertices to a simple configuration. The C++ code should produce an array `U` that contains the numerical solution. In the C++ code, the problem description should be hard coded. Hence do not address input/output aspects.

3. Below is the implementation of the multiplication of a sparse matrix in Column-Sparse Compressed (CSC) format with a dense vector. This implementation uses OpenMP to parallelize the multiplication.

   Here is example explaining the CSC format. Consider the following 3×4 sparse matrix:

$$A = \begin{bmatrix} 0 & 4 & 0 & 8 \\ 2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 6 \end{bmatrix}$$

   The CSC (Compressed Sparse Column) representation is:

   - `nzval` = [2, 4, 5, 8, 6]
   - `rowval` = [1, 0, 2, 0, 2]
   - `colptr` = [0, 1, 3, 3, 5]

   Explanation:

   - `nzval` stores the nonzero values, column by column.
   - `rowval` stores the row index for each value in `nzval`.
   - `colptr` indicates the starting index in `nzval` for each column (with the last entry pointing one past the end).

```
private:
    std::vector<float> nzval;
    std::vector<size_t> rowval;
    std::vector<size_t> colptr;
    size_t m;
    size_t n;
public:
    std::vector<float> multiply(const std::vector<float>& x) const {
        if (x.size() != n)
            throw std::invalid_argument("Size mismatch");
        std::vector<float> y(m, 0);
        #pragma omp parallel for
        for (size_t j = 0; j < n; ++j) {
            for (size_t idx = colptr[j]; idx < colptr[j + 1]; ++idx) {
                size_t i = rowval[idx];
                float val = nzval[idx];
                y[i] += val * x[j];
            }
        }
        return y;
    }
```

   When trying your function, you sometime get 1) incorrect results and 2) degraded performance when using multiple threads.

(a) What is the mistake in the `multiply` function shown above causing these issues? Explain why it is a problem, especially in the context of parallel execution with OpenMP.

Solution: The line `y[rowval[i]] += nzval[i] * x[i]` is not atomic. This means that two threads could both read the same entry of `y` at the same time, each compute their own sum, and then both write their results back. For example, both threads might read the same initial value of `y`, add their respective contributions, and then write their results. In this case, whichever thread writes last will overwrite the result of the other, and the contribution from the first thread will be lost. This is a race condition: the final value in `y` may be missing the contribution from one of the threads.

Even if there are no incorrect results, performance issues can arise due to *false sharing*. False sharing occurs when multiple threads update elements of the result vector `y` that are located close together in memory (i.e., on the same cache line). When this happens, even though the threads are updating different elements, the cache coherence protocol will cause the cache line to bounce between the threads' cores, leading to unnecessary cache invalidations and degraded performance. This can significantly slow down the program, especially if many threads frequently update nearby entries of `y`.

(b) How can you fix the mistake? No need to write code, you can explain in words but then be precise.

Solution: One can add a line with `#pragma omp atomic` directive before the line where the result is updated. This will ensure that the update is atomic, i.e., no two threads can update the same element in the result vector at the same time.

Alternatively, one can create separate result vectors for each thread and then merge them at the end. This will ensure that each thread updates its own result vector and there is no race condition.

The first solution will be faster if there is a large number of columns compared to the number of threads, hence only a few collision will occur. Otherwise, the second solution will be faster.

The advantage of the second solution (using separate result vectors for each thread and merging at the end) is also that it does not suffer from false sharing. Since each thread writes to its own private result vector, there are no concurrent updates to nearby memory locations by different threads, so cache line contention and the associated performance penalty from false sharing are avoided.

4. You now want to distribute the matrix multiplication over multiple compute nodes of a cluster. The sparse matrix is large and should be distributed over the compute nodes.

   Then, the matrix-vector product will be computed for **many** different vectors. So we want connect the compute nodes so as to minimize the cost of the matrix-vector product, not the cost of distributing the matrix.

   You have access to d compute nodes and enough material to connect d pairs of compute nodes together. Using these connections, the unidirectional communication of k bytes takes a time equal to $\alpha + \beta k$. Let γ be the time taken by a compute node to execute one arithmetic operation and z be the number of nonzero entries in the whole sparse matrix.

   (a) How would you interconnect the compute nodes to minimize the time of the matrix-vector product ? Which part of the matrix would be stored in each compute node?

Solution: To minimize the time of the matrix-vector product, we should interconnect the compute nodes in a **star topology** (also called a hub-and-spoke topology), where one central node is connected directly to all other nodes. This is because, for each matrix-vector product, all nodes need access to the entire input vector, and the result vector may need to be gathered. The star topology allows the central node to broadcast the input vector to all other nodes in a single communication step, minimizing the communication time compared to a ring or tree, given a fixed number of connections.

Each compute node should store a subset of the matrix's rows (i.e., a **row-wise partitioning**). Specifically, the n rows of the matrix are divided among the d nodes, so that each node stores approximately n/d rows and all the nonzero entries in those rows. This way, each node can compute its portion of the output vector independently once it has the full input vector.

Alternatively, we could interconnect the compute nodes in a **tree topology**, where each node is a vertex of the tree and the edges represent direct communication links. In this case, the d connections are used to form a spanning tree with d nodes and $d-1$ edges (and possibly one extra redundant connection).

For the matrix distribution, we would use a **column-wise partitioning**: each compute node stores a subset of the matrix's columns, along with all the nonzero entries in those columns. That is, if the matrix has m rows and n columns, each node is responsible for approximately n/d columns and all their nonzero entries.

(b) Assuming the interconnection topology you choose in the first part is used. Suppose a dense vector is loaded in the memory of the first compute node. Which MPI collectives would you use to compute the product of that vector with the sparse matrix and retrieve the result on the first compute node? What would be the time complexity of this operation in terms of $\alpha, \beta, \gamma, d, m, n$ and z (you may not need all these 7 variables in your answer).

Solution: The appropriate MPI collectives are:

- `MPI_Bcast` to broadcast the (partial, for column-wise partitioning) input vector from the first node to all other nodes.
- Each node computes its local matrix-vector product for its assigned rows.
- `MPI_Gather` to collect the (partial, for row-wise partitioning) result vectors from all nodes back to the first node.

For the **star topology** (row-wise partitioning), the time complexity is:

$$T_{\text{star}} = \mathcal{O}(\alpha + \beta n + \gamma(z/d + n) + \alpha + \beta m/d)$$

where

- the first $\alpha+\beta n$ is for broadcasting the input vector of length n (assuming node one can send the data to all other nodes in parallel since it has distinct connections to each node),
- $\gamma(z/d + n)$ is the total computation time for z nonzeros, if d is large and gets close to z, the number of nonzeros per node is small but we still need to iterate through `colptr` which has length m so we need to use $z/d + m$ not just z/d,
- and the second $\alpha + \beta m/d$ is for gathering the result vector of size n.

For the **tree topology** (column-wise partitioning), the time complexity is:

$$T_{\text{tree}} = \mathcal{O}(\alpha \log_2 d + \beta(n/2 + n/4 + n/8 + \cdots) + \gamma(z + n)/d + (\alpha + \beta m) \log_2 d)$$

where

- the first $\alpha \log_2 d + \beta n$ is for scattering the input vector (total size n). The root needs to send half of the vector to each child so communicates vector of length $n/2$. Then each child communicate half of the vector so length $n/4$ etc... In total, we have a sum of $\mathcal{O}(n/2 + n/4 + n/8 + \cdots) = \mathcal{O}(n)$.
- $\gamma(z + n)/d$ is the computation. In case the matrix is super sparse and $z \ll n$, we still need to iterate over `colptr` so we need $z + n$.
- $(\alpha + \beta n) \log_2 d$ is for the communication of the result vector. This time, each communication is with the full result of length m.