Assignment 1: Parallel Matrix Processing with OpenMP

Version 3: 10 PM Feb 8

February 8, 2025

Due Dates

- **Due Date 1:** 11:59 PM, February 12, 2025
- **Due Date 2:** 11:59 PM, February 17, 2025

Introduction

Matrix multiplication between an $m \times n$ matrix A_{mn} and $n \times k$ matrix B_{nk}, $A_{mn}B_{nk}$ is an $m \times k$ matrix C_{mk}. It has applications in many fields, including scientific simulations, machine learning, and social media analysis.

Let a_{ij} be the jth element of the ith row of matrix A, b_{ij} be the jth element of the ith row of matrix B, and c_{ij} be the jth element of the ith row of the resultant matrix C where $C = A * B$.

$$c_{ij} = \sum_{p=0}^{n-1} a_{ip} \cdot b_{pj}, \quad 0 \leq i < m \text{ and } 0 \leq j < k$$

Problem Statement

You have to implement two functions to operate on sparse (Matrix with most of its elements as 0) matrices represented in a block-wise manner. The goal is to create a sparse matrix, represent it efficiently using non-zero blocks, and perform matrix exponentiation while applying specific modifications to the resulting matrix using OpenMP tasks to make the process parallel.

Function 1: Matrix Generation

The first function, `generate_matrix`, is responsible for creating a sparse matrix. Matrices in this context are large and contain mostly zero entries. To optimize storage and computation, only non-zero values are stored in a block-wise manner as explained below.

Signature:

```
map<pair<int,int>,vector<vector<int>>> generate_matrix(int n,int m,int b)
```

where,

- n: The size of the square matrix (i.e., the matrix has n rows and n columns) to generate. This matrix is divided into blocks.
- m: The size of each block. Each block is a smaller square sub-matrix of size $m \times m$, and the entire matrix is divided into $(n/m) \times (n/m)$ blocks. Assume n is divisible by m.
- b: The number of non-zero blocks in the matrix. A non-zero block has at least one non-zero element.

Task

1. Create a sparse matrix by randomly choosing blocks which are non-zero.
2. Ensure that the matrix has b such non-zero blocks. Let each block be identified by its row and column indices (i, j), meaning the top-left starting block is $(0, 0)$.
3. Populate the non-zero blocks with random int32 values. Each block is represented as an $m \times m$ matrix (or a vector of vectors in programming terms).
4. Each block should be generated by a separate OpenMP task (created inside generate_matrix).

Output

Return a `map` data structure where:

- The key is a pair (i, j) indicating the block's position in the matrix.
- The value is the $m \times m$ matrix containing the block's non-zero values.

For example, for $n = 6$, $m = 2$, and $b = 2$, the matrix is divided into 3×3 blocks. The function might create non-zero blocks at positions $(0, 1)$ and $(1, 0)$, and populate these blocks with random values.

Function 2: Matrix Multiplication (`matmul`)

The second function, `matmul`, performs the following operations on the matrix represented by the non-zero blocks generated by the first function. Note that the correctness of the `matmul` function will also be verified using test matrices during evaluation.

1. **Matrix Exponentiation:** Compute the k-th power of the matrix using its non-zero blocks. The result should be **in-place** to reflect the resulting k-th power.
2. **Row Statistic Computation:** This computation is required **only when** $k = 2$. For any other value of k, the function should return an empty vector. When $k = 2$, compute a statistic $S[i]$ for each row i, defined as follows:

   > Count P_i, the number of multiplications involving elements of row i that result in a non-zero product. $S[i] = P_i/B_i$, where B_i is the total number of elements (including zeros) of row i that exist in any non-zero blocks to which row i belongs. This is further explained in the example given at the end.

   Return S as a vector of floating point numbers (`vector<float>`).

   - Use OpenMP tasks: each task should multiply one non-zero block with another. These tasks should concurrently update $S[i]$. This occurs when elements in the j-th and k-th columns of row i participate in multiplications that lead to a non-zero product.

- To handle concurrent updates to the shared variable $S[i]$, you must implement proper synchronization mechanisms to ensure thread safety and correctness of the results.

Function Signature:

```
vector<float> matmul(map<pair<int,int>, vector<vector<int>>> &blocks,
                     int n, int m, int k)
```

The parameters are described as follows:

- `blocks`: A reference to the map containing the non-zero blocks of the matrix, as generated by the first function.
- `n`: The size of the matrix (same as in the first function).
- `m`: The size of each block (same as in the first function).
- `k`: The power to which the matrix should be raised.

Task

1. Pre-process the given matrix:
    - Examine each block in the matrix. If any element in a block is a multiple of 5, replace it with 0.
    - If a block becomes entirely zero after this operation, remove it from the map.
2. Perform matrix exponentiation:
    - Compute the k-th power of the resulting matrix using block-wise multiplication:
    - Ensure that the multiplication considers the positions of the blocks within the matrix, adhering to standard matrix multiplication rules.
    - Also update P_i (the number of multiplications involving elements of row i that result in a non-zero product) for updating $S[i]$.
3. Compute the statistic S[i] as mentioned above

Output

The input map, `blocks`, is modified in place to store the final representation of the matrix after exponentiation and modifications.

Points to Note

- The matrix size n is always divisible by the block size m.
- The matrix is sparse, meaning that only a small fraction of blocks are non-zero. so b << n.
- In the function generate_matrix, we will check that the returned blocks are indeed non-zero (That is they have some non-zero numbers within them).
- The values within non-zero blocks are integers, and their range is from 0 to 256.
- The operations should be efficient, leveraging the sparsity of the matrix to avoid unnecessary computations. We will measure the runtime of "generate_matrix" and "matmul" individually for performance scores.
- The numbers given will all be non-negative. (greater than or equal to 0)

Starter Code

The starter code is provided here. It includes a template.cpp file and a basic checker code. Please note that this is not the final checker; it performs only a few fundamental checks to help verify the correctness of your function. Additionally, the checker is limited to testing for k = 2 .

Example

Let $n = 9$, $m = 3$, and $b = 3$.
Thus, the matrix will be of size 9×9, with 3 blocks of size 3×3 as non-zero

elements.

Example Matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 3 & 1 \\ 2 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix is represented by the following map:

- key: {0,0}
  value: $[[1, 0, 2], [0, 3, 0], [2, 0, 4]]$
- key: {0,2}
  value: $[[0, 0, 0], [0, 3, 1], [0, 0, 5]]$
- key: {2,0}
  value: $[[0, 0, 0], [0, 3, 0], [0, 1, 5]]$

Now, the matrix representation of $\mathbf{A} \times \mathbf{A}$ is:

$$\mathbf{A} \times \mathbf{A} = \begin{pmatrix} 5 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 19 & 5 & 0 & 0 & 0 & 0 & 9 & 3 \\ 10 & 5 & 45 & 0 & 0 & 0 & 0 & 0 & 20 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 9 & 3 \\ 10 & 3 & 20 & 0 & 0 & 0 & 0 & 3 & 26 \end{pmatrix}$$

The map representation of $\mathbf{A} \times \mathbf{A}$ is updated as follows:

- key: {0,0}
  value: $[[5, 0, 10], [0, 19, 5], [10, 5, 45]]$

- key: {0,2}
  value: $[[0, 0, 10], [0, 9, 3], [0, 0, 20]]$
- key: {2,2}
  value: $[[0, 0, 0], [0, 9, 3], [0, 3, 26]]$
- key: {2,0}
  value: $[[0, 0, 0], [0, 9, 0], [10, 3, 20]]$

The statistic to be returned is the number of multiplications involving elements of row i that result in a non-zero product. We then divide this quantity by the number of elements in blocks containing non-zero elements in the row being considered.

Let's consider row 1 of $\mathbf{A} \times \mathbf{A}$:

$$\begin{pmatrix} 5 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 10 \end{pmatrix}$$

1. The number of multiplications that contributed to 5 are 2 (1×1 and 2×2).
2. The number of multiplications that contributed to 10 are 2 (1×2 and 2×4).
3. The number of multiplications that contributed to 10 is 1 (2×5).

The sum-total is $2 + 2 + 1 = 5$.

Now, we divide this by the number of non-zero elements in the row. There are 6 highlighted elements. Therefore, the statistic $S[0]$ is:

$$S[0] = \frac{5}{6}$$

Report

Your report should list your optimizations done in a text file named `readme.txt`, and a CSV file showing your timings for the matrix multiplication function across different matrix sizes on varying number of cores.

Format for Final Scalability Analysis: The performance table records timings (in milliseconds, up to two decimal points) for varying cores with increasing matrix sizes, in CSV format. Below is the formatted version for clarity:

b	2 cores	4 cores	8 cores	16 cores	32 cores	40 cores
2^6						
2^8						
2^{10}						
2^{12}						

Here b is the non-0 input blocks. Keep n = 100,000, k = 2 and m = 16 across all experiments. You are required to maintain the same order of columns in your csv file called 'data.csv'. If the guidelines are not followed, the autograding script will award you 0 marks.

Submission Instructions

Submit the following files on Gradescope:

```
|-- template.cpp
|-- readme.txt
|-- data.csv
```

Note that if you submit more or less than these 3 files you will get a 0. Please stick to the given file names.

Code Guidelines

1. You are not allowed to use *pragma omp for* construct anywhere in the assignment code, not even in comments. We will search for the string "pragma omp for" and if found, your assignment will not be evaluated and will get a 0. You are only supposed to use OpenMP task constructs. We will use the correctness and runtime of the 2 functions for evaluation.

2. Whenever you create an OpenMP task, make sure to add an `if(black_box())` clause to the task pragma. The `black_box()` function is provided by us, and you can assume that it always returns `true`. Failure to include

the `if` clause in every task you create will result in your assignment not being evaluated.

Below is an example of how to use the `if` clause with `black_box()`:

```
#pragma omp task shared(A, B, C) depend(in:B) if(black_box())
    mult_block(A, B, C);
```

Note that the code you intend to execute in parallel will only run in parallel if the `black_box()` function returns `true`. Otherwise, it will run serially.

Execution Instructions

Module Requirements

Before running the code, you must load **ONLY** the following modules:

1. `module load compiler/gcc/9.1.0`
2. `module load compiler/gcc/9.1/mpich/3.3.1`

Run `module purge` before loading any modules so that just the two modules are loaded.

Important Notes

- Do **NOT** modify `check.cpp` or `check.h`.
- Your implementation should be written in `template.cpp` only.
- Do not modify any function definitions in `template.cpp`.
- In `check.cpp`, you may only modify the values of the following variables:
  - `n`
  - `m`
  - `b`
  - `k`

Testing Your Code

check.cpp is a preliminary autograder to verify if your code works correctly. The final autograder will include additional checks on top of the existing ones.

To compile and run the autograder, use these commands:

```
g++ -fopenmp -std=c++17 check.cpp template.cpp -o check
./check
```