COL380 A4:- Sparse Matrix Multiplication using CUDA, OpenMP and MPI

Yash Bansal (2022CS51133)

1 Implementation Details

This section describes the detailed process for multiplying sparse matrices using CUDA, OpenMP, and MPI.

1. **Matrix Structure:** We define a matrix using the following structure:

```
struct mat_struct {
    int exist;
    int height;
    int width;
    int num_blocks;
    vector<uint64_t> vec;
    unordered_map<pair<int, int>, int, pair_hash> mat_map;
};
```

   Here, `vec` stores all the non-zero blocks of the matrix concatenated into a single vector. `mat_map` maps the (row, column) location of each block to its corresponding index in `vec`.

2. **MPI Distribution:** Given N matrices, we distribute the workload across all MPI nodes. Matrices are assigned in batches to different nodes in a round-robin fashion.

3. **Matrix Reading:** On each node, matrices are read in parallel using an OpenMP `parallel for` loop to improve I/O efficiency.

4. **Matrix Multiplication on Node:** Each node multiplies its assigned matrices serially using CUDA, following the method outlined below.

5. **Preparation for CUDA Multiplication:**

   - For two matrices to be multiplied, we iterate over all block locations in their respective `mat_map`.
   - We identify all valid block pairs that need to be multiplied and store them in a vector.
   - A new `mat_map` for the resulting matrix is created, and its `vec` is resized appropriately to store the results.

6. **Integrated Vector Creation:** For each identified block pair:

   - We concatenate three components into an integrated vector: the block from matrix A, the block from matrix B, and an empty space reserved for the result.

This integrated vector is created for all block pairs.

7. **Memory Transfer to CUDA:** The integrated vector is transferred to the device (GPU) memory using `cudaMalloc`.
8. **CUDA Kernel Execution:**
   - Each CUDA block is responsible for multiplying one block pair.
   - Within each block, we launch `block_size` $\times$ `block_size` threads, where each thread computes one element of the resulting block.
   - Threads compute their assigned element via a simple dot-product operation (single loop).
   - The computed results are stored directly into the reserved empty space in the integrated vector.
9. **Copying Results Back:** After the CUDA kernel finishes, the integrated vector (now containing the multiplied blocks) is copied back to the host.
10. **Final Matrix Construction:** We aggregate all the computed blocks, update the sparse matrix structure accordingly, and return the final matrix.
11. **Communication Between MPI Processes:** Each MPI process serializes its partial matrix into a vector and sends it to the root process.
12. **Final Multiplication at Root:** The root process receives all partial matrices and multiplies them serially using the same CUDA multiplication method as above.
13. **Matrix Dumping:** Finally, the resulting matrix is saved to a file for further use or analysis.

2 Performance Analysis

The execution times were obtained, where each data point represents an average of five runs of the code.

Test Case	Number of Cores	Number of Nodes	Time (seconds)
Medium	4	1	0.465
Medium	8	1	0.489
Medium	4	4	1.040
Medium	4	8	1.200
Large	4	1	150.240
Large	8	1	158.322
Large	4	4	143.234
Large	4	8	117.421

Table 1: Execution time comparison for different test cases, number of cores, and nodes.

The observations from these runs are as follows:

- For medium-sized test cases, increasing the number of nodes leads to a higher execution time. This is primarily due to the communication overhead involved in transferring data between nodes. Additionally, since the matrices are relatively small, the use of OpenMP does not provide significant performance benefits.

- For larger matrices, OpenMP provides a slight improvement in performance. Moreover, increasing the number of nodes reduces the overall execution time, as multiple nodes perform matrix multiplications in parallel using CUDA simultaneously.