CS 531 HW 5

Hritvik

1 Introduction

The goal of this assignment is to implement a parallel sparse matrix–vector multiplication (SpMV) using a combination of MPI and OpenMP and to evaluate its performance on a realistic sparse matrix. The target operation is

$$y \leftarrow Ax,$$

where A is a large sparse matrix in Matrix Market format and x is a dense vector. The provided test case is `cant/cant.mtx`

Goals:

- a correct parallel SpMV implementation,
- verification against a reference result vector, and
- timing experiments using different MPI and OpenMP configurations, including a hybrid run.

2 Implementation

2.1 MPI

The matrix is distributed across MPI ranks in blocks of rows. Let P denote the number of ranks. Each rank p is assigned a contiguous range of rows $[r_{\min}^{(p)}, r_{\max}^{(p)})$ that partitions $[0, n)$.

Rank 0 performs:

- it scans the global COO arrays,
- assigns each nonzero to the rank that owns its row index, and
- sends the corresponding local COO arrays to each rank using MPI.

After this step, every rank holds only the entries whose row index lies in its local row range. The input vector x is broadcast to all ranks in the "Vec Bcast" phase using `MPI_Bcast`.

Each rank also allocates a local segment of the output vector y, covering its assigned row range. A global result vector is produced at the end using `MPI_Gatherv` (the "Res Reduce" and "Store" phases).

2.2 OpenMP

Within each MPI rank, the SpMV kernel is parallelized using OpenMP. The computation is essentially

$$y_i \mathrel{+}= a_{ij} x_j$$

for each local nonzero (i, j, a_{ij}).

Because multiple nonzeros can contribute to the same output row i, I used simple row-level locking:

- an array of OpenMP locks is allocated, one per local row, in the "Lock Init" phase;
- the OpenMP parallel region loops over local COO entries; and
- each update acquires the lock for row i, performs the `y[i] += a * x[j]` update, and then releases the lock.

The timing reported under "COO SpMV" measures only the kernel region — that is, the OpenMP-accelerated loop over local nonzeros — and excludes I/O and MPI setup.

2.3 Hybrid

The hybrid configuration uses both MPI and OpenMP: the matrix is distributed over multiple ranks, and each rank uses multiple threads. The core kernel is the same as in the OpenMP-only case; the only difference is that each rank now owns fewer rows and fewer nonzeros.

For the hybrid experiment, I used:

- 4 MPI ranks on a single node, and
- 7 OpenMP threads per rank,

so that the total number of threads is $4 \times 7 = 28$.

3 Correctness

The correctness of the implementation was verified by comparing the computed result vector `ans_*.mtx` against the provided reference `cant/ans.mtx` For example:

```
diff cant/ans.mtx ans_4.mtx
diff cant/ans.mtx ans_omp1.mtx
diff cant/ans.mtx ans_omp2.mtx
...
diff cant/ans.mtx ans_4ranks_omp7.mtx
```

In all cases, the only differences reported by `diff` were in the last decimal place of some entries. A representative discrepancy is:

```
< -256.3818732256
---
> -256.3818732255
```

which corresponds to an absolute difference of 10^{-10}.

This behavior is expected: changing the order of floating-point operations (additions and multiplications across different processes and threads) can cause roundoff differences at that level

4 Performance results

In all runs, the matrix and vector are loaded from disk using the same code path, and the reported "COO SpMV" time is taken from the "Module Time" output.

4.1 OpenMP scaling (1 MPI rank)

Table 1 shows the performance when running with a single MPI rank and varying the number of OpenMP threads. Only the "COO SpMV" kernel time is reported.

MPI ranks	OMP threads	COO SpMV time (s)
1	1	0.032 072
1	2	0.033 013
1	4	0.032 865
1	8	0.032 430
1	28	0.033 470

Table 1: SpMV kernel time for `cant.mtx` with 1 MPI rank and varying numbers of OpenMP threads.

The kernel time stays essentially flat around 0.032–0.033 seconds for 1–28 threads. This is consistent with SpMV being largely memory-bandwidth-bound: adding more threads does not significantly reduce the time once the memory is saturated. Additionally, the use of row-level locks adds some synchronization overhead that can offset potential speedups from increased parallelism.

4.2 MPI and hybrid configurations

Table 2 compares a pure-MPI configuration (multiple ranks, one thread per rank) to a hybrid configuration that combines MPI and OpenMP.

Configuration	MPI ranks	OMP threads/rank	COO SpMV time (s)
Pure MPI (single node)	4	1	0.093 811
Hybrid (single node)	4	7	0.002 158

Table 2: SpMV kernel time for pure-MPI and hybrid MPI+OpenMP runs on `cant.mtx`.

The 4-rank MPI-only configuration is noticeably slower than the 1-rank OpenMP runs (Table 1). This is expected in this implementation: distributing the matrix across multiple ranks introduces communication and setup overhead (the "Mat Scatter" and reduction steps), while the core computational work per rank becomes smaller.

The hybrid configuration with 4 ranks and 7 threads per rank shows a very small reported kernel time (≈ 2 ms). Because the absolute runtime of the kernel is so short, small sources of measurement noise and timer granularity can produce significant relative variation. The main takeaway is that the hybrid configuration is capable of using both MPI and OpenMP without sacrificing correctness, and in this particular case the kernel time falls below the cost of I/O and setup.

5 Analysis

- The implementation is numerically correct across all tested parallel configurations, with differences from the reference output appearing only at the last decimal place due to floating-point roundoff.
- For this matrix and code, increasing the number of OpenMP threads beyond one does not significantly improve performance. The combination of memory-bandwidth limits and row-level locking overhead likely explains the nearly flat timing.
- Pure-MPI parallelization on a single node is slower for this specific setup because of the overhead of scattering the matrix and gathering the result, compared to the relatively cheap local computation.
- The hybrid configuration demonstrates that the MPI+OpenMP approach runs correctly and can yield very low kernel times