COL380 Assignment 4 Report: Parallel Matrix Multiplication

Parallelization Strategy

Our implementation focuses on efficiently multiplying a sequence of sparse matrices across multiple nodes with multi-core CPUs and GPUs. We employ a three-level parallelization approach:

1. MPI (Distributed Memory Parallelization)

- **Matrix Distribution**: Input matrices are initially read in parallel with each MPI process handling a subset of matrices.
- **Work Distribution**: We use a master-worker pattern where rank 0 coordinates the multiplication sequence and distributes matrix multiplication tasks to worker processes.
- **Asynchronous Communication**: Non-blocking MPI operations are used to overlap computation and communication.
- **Multiplication Order Optimization**: A distributed algorithm finds an efficient multiplication order based on matrix sparsity patterns to minimize overall computation.

2. OpenMP (Shared Memory Parallelization)

- **Block-Level Parallelism**: Matrix multiplication is parallelized at the block level using OpenMP threads.
- **Dynamic Scheduling**: `#pragma omp for schedule(dynamic, 32)` is used to handle load imbalance due to sparse block distribution.
- **Thread-Local Storage**: Each thread maintains local block storage to reduce lock contention.
- **Memory Prefetching**: Precomputation of connectivity information reduces access latency.

3. CUDA (GPU Acceleration)

- **Custom CUDA Kernels**: We implemented specialized kernels for sparse block multiplication:
  - `blockMultiplyKernel`: For single block-pair multiplication
  - `batchBlockMultiplyKernel`: For processing multiple block multiplications simultaneously
- **Shared Memory Optimization**: Block data is loaded into shared memory to reduce global memory access.
- **Multiple CUDA Streams**: 8 concurrent streams enable overlapping of computation and memory transfers.
- **Memory Coalescing**: Memory access patterns are designed to maximize coalescing for better throughput.
- **Loop Unrolling**: `#pragma unroll` directives optimize inner multiplication loops.

Performance Analysis

Matrix Reading Times (Medium Test Case with 5 matrices, block size 4)

MPI Processes	Matrix Reading Time (s)	Total Execution Time (s)
1	0.113806	0.731605
2	0.042409	0.808786
3	0.086719	0.765587
4	2.219400	2.941370

Test Results

Test Case	Matrices	Dimensions	Block Size	Sparsity	NP=1 (s)	NP=2 (s)	NP=4 (s)
Small	3	100–200	16×16	10%	0.099	0.123	4.904
Medium	10	500–1000	32×32	5%	0.268	0.290	2.496
Large	25	800–1500	32×32	3%	0.615	0.750	2.897
Very Large	50	1000–2000	32×32	2%	0.814	0.995	3.066
Medium Sample	5	∼1024	4×4	¡10%	0.588	0.736	2.902

Key Points

- Single process (NP=1) generally gives the fastest execution across all cases.
- Increasing to 2 or 4 processes does not improve performance; in fact, total time increases.
- For NP=4, matrix reading becomes a major bottleneck due to file system contention.
- Communication and I/O overheads dominate for the given matrix sizes and sparsity levels.

Implementation Highlights

- **Adaptive Computation Strategy**:
  - Small blocks ($\leq 8 \times 8$) are processed on CPU to avoid the overhead of transferring data to the GPU.
  - Batching of multiple block multiplications between the same matrix locations improves GPU utilization.
  - Computations are conditionally executed only if there is potential for a non-zero result, based on sparsity patterns.
- **Memory Optimizations**:
  - Host memory allocations use pinned memory (`cudaMallocHost`) to accelerate host-to-device data transfers.
  - CUDA buffers are pre-allocated and reused throughout the execution to minimize allocation overhead.
  - Sparse matrices are represented with hash-based unordered maps for efficient lookup and access.
- **Algorithmic Optimizations**:
  - A greedy heuristic is used to determine a multiplication order that minimizes intermediate matrix sizes.
  - Precomputed block connectivity reduces unnecessary calculations.
  - 8 concurrent CUDA streams allow overlapping of computation with memory transfers for better device utilization.

Conclusion

Our hybrid parallel implementation effectively combines MPI, OpenMP, and CUDA for sparse matrix multiplication. However, performance analysis shows that for the tested problem sizes, a single MPI process achieves the best results due to minimal communication overhead and efficient local parallelism using OpenMP and CUDA.

For relatively small matrices with moderate sparsity, the costs associated with communication, file I/O contention, and GPU sharing across MPI processes outweigh the benefits of distributed parallelization. Future work should explore optimizing I/O operations and minimizing inter-process communication to better leverage multiple processes for larger or more complex problem instances.