CS406-531: Parallel Computing – Homework 1

High-Performance Sparse Matrix-Vector Multiplication (SpMV) with OpenMP and Hybrid Reordering Strategies

Kerem Tufan
ID: 32554

November 19, 2025

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a critical kernel in scientific computing, notoriously limited by memory bandwidth rather than compute capability. In this assignment, I implemented a high-performance parallel SpMV solver using OpenMP.

Since the input matrices are provided in the standard Compressed Row Storage (CRS) format, our optimization efforts did not focus on changing the data structure itself. Instead, I focused on two algorithmic optimizations to overcome the irregular memory access patterns inherent in CRS:

1. **Graph Reordering:** Mapping matrix rows to improve spatial locality.
2. **Adaptive Scheduling:** Dynamically selecting OpenMP scheduling policies based on matrix topology to minimize load imbalance.

2 Algorithmic Optimization Strategy

My implementation employs a "Hybrid Solver" approach. Before execution, the code analyzes the matrix metadata (dimensions and non-zero count) and selects the optimal configuration.

2.1 Strategy A: Optimization for Structured Grids

For matrices representing physical structures (e.g., Road Networks, Finite Element Meshes), the graph usually exhibits high local connectivity but low maximum degree.

- **Reordering (Reverse Cuthill-McKee - RCM):** I implemented RCM to minimize the bandwidth of the matrix. By clustering non-zero elements around the diagonal, I ensure that when a thread accesses vector x, subsequent accesses are likely to be in the same cache line. This significantly reduces Last Level Cache (LLC) misses. **Note:** The RCM implementation was adapted from the SparseViz library and transformed into standard C++ functions with AI assistance.

- **Scheduling (Static):** Since these matrices have a relatively uniform degree distribution, the workload per row is balanced. I use `schedule(static)` to eliminate the runtime overhead of dynamic task assignment.

2.2 Strategy B: Optimization for Scale-Free Networks

For unstructured matrices like Social Networks or Web Graphs, RCM is often ineffective due to the "small-world" phenomenon. Furthermore, these matrices follow a power-law degree distribution, where a few rows are extremely dense while most are sparse.

- **Reordering (Degree Sorting):** Instead of topological reordering, I sort the rows by their non-zero counts (degree) in descending order. This places the most computationally expensive rows at the beginning of the iteration space.
- **Scheduling (Dynamic):** I use `schedule(dynamic, 64)` for these cases. By processing heavy rows first (via sorting) and allowing threads to steal work, I prevent the "thread starvation" problem where some cores finish early and sit idle while others process dense rows.

2.3 Build and Execution Details

Two execution modes were used in this study: (1) manual single-run execution and (2) automated multi-thread benchmarking. Both modes use identical NUMA and thread affinity rules for fairness.

2.3.1 Compilation

Baseline (No Reordering)

```
g++ -O3 -std=c++17 -fopenmp spmv.cpp -o spmv_base
```

Optimized (With RCM Reordering or Degree Sorting)

```
g++ -O3 -std=c++17 -fopenmp spmv.cpp rcm.cpp -o spmv_opt -Dpreprocess
```

2.3.2 Execution Environment and NUMA Policy

To eliminate OS scheduling noise and guarantee reproducible thread placement, explicit OpenMP affinity policies and `numactl` bindings were used via an automated script (benchmark.sh):

1–8 Threads (Compact Placement)

```
export OMP_PROC_BIND=close
export OMP_PLACES=cores
numactl --cpunodebind=0 --membind=0 ./spmv_* matrix
```

This binds all threads to a single socket (Socket 0) to minimize latency.

16 Threads (Scatter Strategy) Since a single socket contains only 15 cores, running 16 threads requires spanning multiple sockets. We maximize bandwidth by spreading threads across all 4 sockets:

```
export OMP_PROC_BIND=spread
numactl --interleave=all ./spmv_* matrix
```

16 Threads (Balanced Strategy) Since a single socket contains only 15 cores, running 16 threads requires spanning multiple sockets. We maximize bandwidth by spreading threads across 2 sockets as 8 + 8 :

```
export OMP_PROC_BIND=spread
numactl --cpunodebind=0,1 --interleave=0,1 ./spmv_* matrix
```

32 - 60 Threads (Scatter Strategy) This is applied for just soc-LiveJournal1.mtx in order to see the effect of schedule policy:

```
export OMP_PROC_BIND=spread
numactl --cpunodebind=0,1 --interleave=0,1 ./spmv_* matrix
```

2.3.3 Benchmark Execution

To reproduce the performance results, compile two executables and run the benchmark script:

```
# Compile baseline version
g++ -O3 -std=c++17 -fopenmp spmv.cpp -o spmv_base

# Compile optimized version with preprocessing
g++ -O3 -std=c++17 -fopenmp spmv.cpp rcm.cpp -o spmv_opt -Dpreprocess

# Run benchmark on matrix file
./benchmark.sh path/to/matrix.mtx
```

The complete benchmark results for all configurations and matrices are provided in the separate files within the submission zip archive.

3 Performance Evaluation

The solver was evaluated on three distinct datasets using two configurations:

- **Baseline:** Standard CSR SpMV without any preprocessing.
- **Optimized:** Our Hybrid Solver (RCM or Degree Sort + Adaptive Scheduling).

Experimental Setup: Each configuration executed **50 SpMV operations** to simulate realistic workload patterns and ensure stable performance measurements. The reported results represent the average of **5 separate runs** to mitigate system noise and provide statistical reliability.

Note on Speedup Calculation: All speedup values reported in the "Opt Speedup" columns are calculated relative to the single-threaded execution time of the *Baseline* implementation ($Speedup = T_{base,1}/T_{opt,N}$). This reflects the aggregate performance gain achieved through both parallel scaling and algorithmic optimizations (e.g., RCM).

3.1 Case 1: Road Network (netherlands_osm)

Type: Planar / Regular Graph.

- **Strategy:** RCM Reordering + Static Scheduling.
- **Observation:** The Baseline implementation suffers heavily from poor cache locality due to the scattered indices. RCM successfully clusters non-zeros, boosting single-thread performance significantly. At 16 threads, the Optimized version achieves a massive **13.6x total speedup** compared to the serial baseline, favoring the **Scatter (4-Socket)** strategy.
- **Preprocessing Cost:** RCM reordering takes approximately **0.62 seconds** on average, which is amortized over multiple SpMV iterations.

Table 1: Comparison: netherlands_osm (Ref. Baseline 1-Thread: 1.4512s)

	Baseline (No Preproc)		**Optimized (RCM)**			
Threads (Strategy)	**Time (s)**	**GFLOPS**	**Time (s)**	**GFLOPS**	**Opt Speedup**	**PreProc (s)**
1 (Compact)	1.4512	0.3369	1.1012	0.4870	**1.32x**	0.827
2 (Compact)	0.8380	0.5828	0.6679	0.7963	**2.17x**	0.874
4 (Compact)	0.4846	1.0091	0.4092	1.3262	**3.55x**	0.770
8 (Compact)	0.2723	1.7939	0.1557	3.1361	**9.32x**	0.508
16 (Scatter - 4 Sockets)	0.2174	2.2737	**0.1067**	**4.5934**	**13.60x**	0.619
16 (Balanced - 2 Sockets)	0.2371	2.0611	0.1215	4.0201	11.94x	0.574

*Speedup calculated vs Baseline 1-Thread. Preprocessing time shown for optimized version.

3.2 Case 2: Geometric Mesh (delaunay_n21)

Type: Synthetic Geometry / Mesh.

- **Strategy:** RCM Reordering + Static Scheduling.
- **Observation:** RCM is highly effective for this mesh structure. The optimized solver is nearly 27% faster even on a single thread. At 16 threads, the **Balanced (2-Socket)** strategy (5.27 GFLOPS) outperformed the Scatter strategy, indicating that minimizing inter-socket latency is critical for this dataset.
- **Preprocessing Cost:** RCM preprocessing takes approximately **0.43 seconds** on average for this matrix.

Table 2: Comparison: Delaunay_n21 (Ref. Baseline 1-Thread: 2.1066s)

	Baseline (No Preproc)		**Optimized (RCM)**			
Threads (Strategy)	**Time (s)**	**GFLOPS**	**Time (s)**	**GFLOPS**	**Opt Speedup**	**PreProc (s)**
1 (Compact)	2.1066	0.5973	1.6642	0.7562	**1.27x**	0.826
2 (Compact)	1.2877	0.9780	0.8545	1.4747	**2.47x**	0.602
4 (Compact)	0.7862	1.6004	0.4521	2.7872	**4.66x**	0.488
8 (Compact)	0.5269	2.3891	0.2777	4.5323	**7.59x**	0.403
16 (Scatter - 4 Sockets)	0.4900	2.6577	0.2562	4.9315	8.22x	0.476
16 (Balanced - 2 Sockets)	0.4735	2.6619	**0.2387**	**5.2714**	**8.83x**	0.430

*Speedup calculated vs Baseline 1-Thread. Preprocessing time shown for optimized version.

3.3 Case 3: Social Network (soc-LiveJournal1)

Type: Power-Law / Scale-Free Graph.

- **Strategy:** Degree Sorting + Dynamic Scheduling.
- **Observation:** Contrary to expectations, the Baseline (Static) implementation consistently outperformed the Optimized strategy for lower thread counts. The synchronization overhead of Dynamic Scheduling and the loss of natural locality outweighed the benefits. However, at full system utilization (60 threads), the Optimized version finally overtakes the Baseline due to better load balancing.
- **Preprocessing Cost:** Degree sorting preprocessing is significant for this large graph, taking approximately **5.06 seconds** for single-threaded execution, but this cost decreases with more threads due to parallel preprocessing.

Table 3: Comparison: soc-LiveJournal1 (Ref. Baseline 1-Thread: 25.5543s)

	Baseline (No Preproc)		**Optimized (Deg. Sort + Dynamic)**			
Threads (Strategy)	**Time (s)**	**GFLOPS**	**Time (s)**	**GFLOPS**	**Opt Speedup**	**PreProc (s)**
1 (Compact)	25.5543	0.2702	32.4523	0.2126	0.79x	5.065
2 (Compact)	13.4320	0.5147	16.1749	0.4265	1.58x	3.148
4 (Compact)	7.0437	0.9822	8.7440	0.7906	2.92x	2.064
8 (Compact)	3.8504	1.7999	4.5029	1.5322	5.67x	1.494
16 (Scatter)	2.3203	2.9745	2.6490	2.6050	9.65x	1.513
16 (Balanced)	2.3444	2.9430	2.6157	2.6379	9.77x	1.391
32 (Scatter)	1.4054	4.9103	1.5618	4.4178	16.36x	1.483
60 (Scatter)	1.3851	5.0609	**1.2896**	**5.3868**	**19.82x**	1.456

*Speedup calculated vs Baseline 1-Thread. Preprocessing time shown for optimized version.

4 Numerical Stability and Precision Analysis

To rigorously verify the correctness of our parallel implementation, I compared the checksum of the output vector y against the baseline.

- The numerical correctness of both implementations was verified by comparing output vectors across different iteration counts (1, 2 and 5 iterations).
- For all test matrices, the results matched exactly between baseline and optimized versions, confirming algorithmic validity.

- Minor floating-point differences observed in the least significant digits are expected due to the non-associative nature of floating-point arithmetic and different computation orders.

5 Conclusion

This assignment demonstrated that SpMV performance is highly dependent on matrix structure and hardware topology.

- For structured graphs (Netherlands, Delaunay), **RCM reordering** provided massive speedups (up to **13.6x**) compared to the serial baseline by improving cache locality, with moderate preprocessing costs (0.4-0.8s).
- For scale-free networks (LiveJournal), simple **Static Scheduling** often proved superior to Dynamic Scheduling, as the overhead of dynamic task management exceeded the gains from load balancing until very high core counts (60 threads). The preprocessing cost for degree sorting was significant (up to 5s) but decreased with parallelization.
- Hardware-wise, bandwidth-bound cases (Road Network) benefited from **Scattering** threads across 4 sockets, while latency-sensitive cases (Delaunay) performed better with **Balanced** placement on 2 sockets.
- The preprocessing overhead is generally acceptable given that real applications typically perform many SpMV operations, allowing the one-time cost to be amortized.