Connected Components Detection in Large-Scale Graphs (Distributed-Memory)

Gerasimou Dimitrios AEM: 10813

Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki
050: Parallel and Distributed Systems

Thessaloniki, Greece
December 2025

Abstract

We evaluate connected components (CC) detection on massive sparse graphs using a hybrid **MPI + OpenMP** implementation targeting distributed-memory HPC systems. Graphs are stored in a custom binary **CSC** format and distributed by contiguous column blocks across MPI ranks, while intra-rank work is parallelized with OpenMP.

Results show that performance is strongly workload and layout dependent. On **com-Friendster** (65,608,366 vertices, 3,612,134,270 nonzeros), splitting a 128-core node into multiple MPI ranks reduces runtime from 100.5 s (1×128) to 29.2 s (16×8), a 3.4× improvement. Using a balanced per-node configuration (2 ranks/node, 64 threads/rank), strong scaling reaches **4.59× speedup (relative to 1 node)** at 8 nodes (1,024 cores), achieving 564.7 Medge s^{-1}. In contrast, on **mawi_201512020330** (226,196,185 vertices, 480,047,894 nonzeros), the best runtime occurs on a single node (5.83 s), while scaling to 8 nodes slows to 10.54 s (0.55× speedup relative to 1 node), indicating that communication and synchronization overheads dominate for this workload in the current implementation.

1 Introduction

Connected components (CC) partition a graph into maximal sets of mutually reachable vertices and are a core primitive in graph analytics (community structure, connectivity testing, preprocessing for other algorithms). On modern machines, CC can be accelerated with shared-memory parallelism; however, real-world graphs may exceed single-node memory capacity or be limited by memory bandwidth and NUMA effects, motivating distributed-memory designs.

This report studies a hybrid **MPI + OpenMP** CC implementation. The emphasis is practical: (i) how graph distribution affects performance, (ii) how intra-node layout (ranks vs threads) changes runtime even at fixed cores, and (iii) when scaling out to more nodes helps or hurts.

2 Graph Representation and Distribution

Graphs are represented as a binary **Compressed Sparse Column (CSC)** structure. Only adjacency is stored; any non-zero numeric values are treated as 1.

```c
typedef struct {
        uint32_t  nrows, ncols;  /* vertices */
        size_t    nnz;           /* edges / nonzeros */
        uint32_t *col_ptr;       /* ncols + 1 */
        uint32_t *row_idx;       /* nnz */
} CSCBinaryMatrix;
```

For distributed execution, each MPI rank owns a contiguous range of columns (a *column-block* partition). This design is convenient for CSC because it (i) preserves sequential access patterns in `col_ptr/row_idx`, (ii) keeps per-rank adjacency compact, and (iii) provides simple ownership rules for vertex labels.

3 Hybrid Distributed CC Method

The implementation follows an iterative label-propagation style structure adapted to distributed memory:

1. **Initialization:** each vertex label is initialized to its global id.
2. **Local relaxation:** each rank scans its local CSC columns and performs label relaxations in parallel (OpenMP).
3. **Boundary exchange:** updates affecting non-local vertices are exchanged between ranks (MPI).
4. **Convergence check:** a global reduction determines whether any label changed in the iteration.

Dataset	Vertices	Nonzeros (nnz)
com-Friendster	65,608,366	3,612,134,270
mawi_201512020330	226,196,185	480,047,894

Table 1: Datasets used in evaluation.

In this model, scalability is largely driven by the ratio of useful local work (label relaxations over local adjacency) to distributed overhead (boundary messages and global synchronizations). Graph structure and partitioning determine how frequently non-local vertices are touched and how quickly labels converge.

4 Experimental Setup

All experiments were executed on the **Aristotelis** HPC cluster (system tag: `rome`) using AMD EPYC 7662 CPUs (64 cores/socket, 256 GiB memory per node). Benchmarks were run on 19 December 2025. Reported times correspond to mean runtime as shown in figures.

5 Results

5.1 Single-node OpenMP scaling (1 MPI rank/node)

Figure 1 evaluates intra-node scaling using a single MPI rank and varying OpenMP threads. Friendster improves up to 32 threads and then degrades sharply at 64–128 threads, consistent with increasing NUMA traffic, memory bandwidth contention, and reduced locality at high thread counts. MAWI shows mild gains up to 64 threads and also degrades at 128 threads.

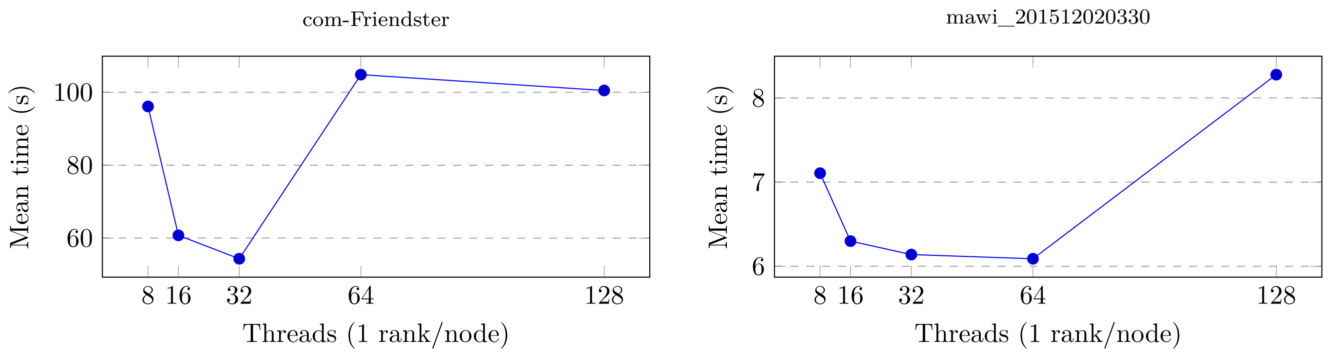

Figure 1: OpenMP-only scaling on 1 node using 1 MPI rank (time vs threads).

5.2 Intra-node layout: splitting 128 cores into MPI ranks

Keeping a node fixed at 128 total cores, Figure 2 shows the effect of splitting the node into multiple MPI ranks. Friendster benefits substantially: runtime drops from 100.5 s (1×128) to about 29 s for 2–16 ranks with fewer threads/rank. MAWI exhibits a weaker optimum near 2 ranks × 64 threads.

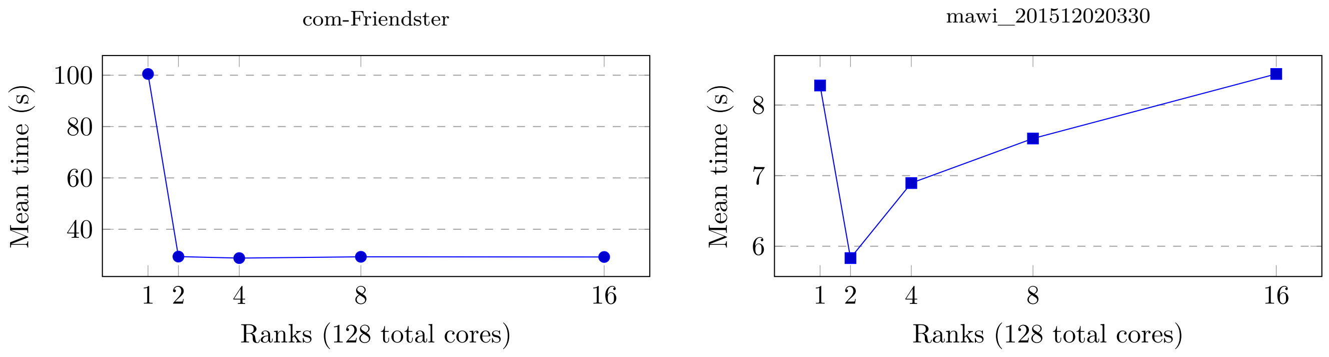

Figure 2: Effect of rank splitting at 128 total cores (1 node): mean time vs ranks/node.

5.3 Strong scaling with the recommended per-node configuration

For strong scaling we use the best-performing per-node configuration from benchmarking: (2 MPI ranks/node, 64 threads/rank). Figure 3 reports speedup relative to the 1-node baseline, alongside an ideal linear reference. Friendster scales to 8 nodes (4.59× relative to 1 node), while MAWI shows slowdowns (speedup < 1) as nodes increase.

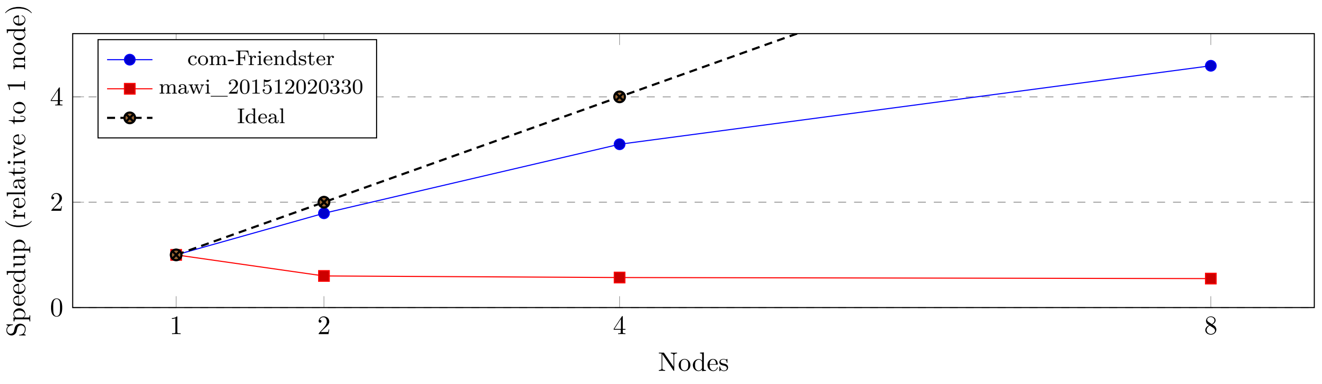

Figure 3: Strong scaling speedup using 2 ranks/node and 64 threads/rank.

Dataset	Best 1-node time	8-node speedup (vs 1 node)	8-node throughput
com-Friendster	28.77 s to 29.35 s	4.59×	564.7 Medge s^{-1}
mawi_201512020330	5.83 s	0.55×	45.5 Medge s^{-1}

Table 2: Summary of key outcomes. Throughput is reported in million nonzeros processed per second.

6 Analysis and Discussion

Friendster: layout-sensitive, then scales well. With 1 rank/node, Friendster degrades beyond 32 threads. However, splitting the same 128 cores into multiple ranks stabilizes runtime around 29 s. A likely explanation is improved NUMA locality and reduced shared-memory contention: smaller per-rank working sets and more localized memory allocation can reduce cross-socket traffic and bandwidth pressure. Scaling out is then effective because the per-rank workload remains large enough to amortize distributed overhead.

MAWI: insufficient work per rank to amortize overhead. MAWI has far fewer nonzeros per vertex than Friendster: roughly 480,047,894/226,196,185 ≈ 2.12 nnz/vertex, compared to Friendster's 3,612,134,270/65,608,366 ≈ 55.1 nnz/vertex. With less local work per iteration, the fixed costs of boundary exchanges and global convergence checks become dominant. As a result, distributing MAWI across more nodes increases synchronization and communication relative to computation, producing a net slowdown in the current design.

Overall, the results support a practical rule: *hybrid tuning is workload-dependent.* The ranks-per-node choice can be as important as the node count, and scaling out is beneficial only when computation per rank is large enough to hide distributed overhead.

7 Conclusions

We evaluated a distributed-memory connected components implementation using MPI + OpenMP on two large graphs. Friendster benefits strongly from intra-node rank splitting and achieves **4.59**× speedup at 8 nodes relative to the 1-node baseline under a balanced layout (2 ranks/node, 64 threads/rank). MAWI achieves its best runtime on one node and slows down when distributed, suggesting that overheads dominate for this workload in the current implementation.

Future Directions

- Reduce communication volume (ghost vertices, compact update encoding, or selective boundary propagation).
- Improve partitioning quality (edge-balanced or communication-aware decompositions beyond simple column blocks).
- Reduce global synchronization frequency (e.g., fewer/all-reduce checks or more asynchronous progress).
- Add instrumentation: break down time into compute, exchange, and global synchronization to localize bottlenecks.

Code Availability: Source code and benchmark results are available at: `https://github.com/dimgerasimou/pds-hw2-mpi-connected-components`.