

# PDS Project 2

ΑΙΜΙΛΙΟΣ ΜΑΥΡΟΘΑΛΑΣΣΙΤΗΣ

ΑΠΘ

amavroth@ece.auth.gr

AEM : 10983

**Abstract**—We extend a shared-memory OpenMP implementation for iterative graph processing to a distributed-memory MPI environment, targeting large sparse graphs that do not fit in a single node’s memory. Starting from an OpenMP baseline, we introduce row-wise matrix partitioning across MPI ranks, hybrid MPI+OpenMP parallelism, and global label synchronization using collective communication. The implementation is evaluated on the Aristotelis high-performance computing cluster using the Slurm workload manager. We study the impact of node count and synchronization frequency on performance, focusing on runtime, convergence behavior, and scaling efficiency. Experimental results on a real-world network dataset show that frequent global synchronization yields the best performance, while increasing the number of nodes beyond a small scale provides limited benefit due to communication and synchronization overheads. These findings highlight the practical trade-offs between computation and communication in hybrid MPI+OpenMP graph algorithms.

## I. From OpenMP baseline to MPI (and how we ran it on Slurm)

### A. Baseline (Phase 1): shared-memory OpenMP code

Our starting point was a single-node OpenMP implementation that processes a sparse adjacency matrix loaded from a MATLAB .mat file. The algorithm follows an iterative label-propagation scheme and terminates when no labels change.

Parallelism is applied over matrix rows using `#pragma omp parallel for`, with a reduction on a global changed flag to detect convergence. This OpenMP version serves as the correctness baseline and single-node performance reference for the distributed-memory experiments.

### B. Phase 2: distributed-memory MPI extension

In Phase 2 we extended the baseline to run on distributed memory using MPI. The goal was to handle graphs that do not comfortably fit in one node’s memory, and to scale computation across multiple nodes.

The main changes compared to the OpenMP version were:

a) MPI initialization and rank roles: We added standard MPI startup (`MPI_Init`), retrieved `world_size` and `world_rank`, and ensured only rank 0 prints usage/errors (to avoid duplicated output).

b) Matrix loading + distribution: Rank 0 loads the sparse matrix from the .mat file and extracts the CSC arrays. Then:

- Dimensions (`nrows`, `ncols`, `nnz`) are broadcast to all ranks with `MPI_Bcast`.

- Rows are partitioned across ranks using a block distribution with remainder handling (nearly equal chunk sizes).
- Rank 0 sends each rank its local row-pointer segment and corresponding column-index slice (via `MPI_Send`); other ranks receive them (`MPI_Recv`) and locally rebase row pointers so indexing starts from 0.

c) Hybrid parallelism: MPI + OpenMP: Inside each MPI rank we keep OpenMP parallelism on the local row range:

- The local update step parallelizes over local rows.
- Each local row maps to a global row index (`global_row = row_start + local_row`).

d) Global label consistency (synchronization): For simplicity and correctness, we kept a full label array `A_1` on every rank (same global indexing). After a batch of local updates, all ranks synchronize labels using:

`MPI_Allreduce(A_1, MPI_MIN)`

This effectively propagates the best (minimum) label information globally.

e) K-sync optimization knob: To reduce communication overhead, we introduced `K_sync`: instead of synchronizing after every local iteration, each rank performs up to `K_sync` local iterations before calling `MPI_Allreduce`. This provides a simple trade-off:

- small `K_sync`  $\Rightarrow$  more frequent synchronization (more communication, potentially fewer outer rounds),
- large `K_sync`  $\Rightarrow$  fewer synchronizations (less communication, but potentially more stale information between ranks)
- Or at least we thought so. We couldn’t show that larger `K` offers fewer outer rounds but we are confident that if we could run the tests in a larger amount of nodes we would be able to provide a graph of this relationship.

Convergence is checked once per outer synchronization step using an `MPI_Allreduce` logical-or over a local “changed” flag. Rank 0 reports total runtime and number of connected components (via the number of distinct final labels).

### C. Running on Aristotelis with Slurm (final test script)

For the final set of tests we used the following Slurm setup (Rome partition), compiling on-the-fly and then running with mpiexec.

a) Where we change nodes / MPI ranks / threads:

The resource scaling knobs are all in the #SBATCH header:

- Nodes: #SBATCH -nodes=2. This is the main control for multi-node scaling (we change this to 1, 2, 4, ...depending on the experiment).
- MPI ranks per node: #SBATCH -ntasks-per-node=1. Total MPI ranks become SLURM\_NTASKS = nodes × ntasks-per-node. If we want more ranks per node, we increase this value.
- OpenMP threads per rank: #SBATCH -cpus-per-task=32. We bind OpenMP to exactly the requested cores using:

```
export OMP_NUM_THREADS=
$SLURM_CPUS_PER_TASK
```

So changing -cpus-per-task directly changes the number of threads each MPI rank uses.

b) Partition and time limits: We ran on -partition=rome and set a short wall time (-time=00:01:00) for quick iteration during testing.

c) Compilation step inside the job: Inside the script we compile with OpenMP + MPI enabled:

```
mpic++ -O2 -fopenmp ... -lmatio -lhdf5
```

We explicitly point include/library paths using MATIO\_ROOT and HDF5\_ROOT, then set LD\_LIBRARY\_PATH so the executable can find the shared libraries at runtime.

d) Runtime parameters and reporting:

We print basic runtime context: hostname, NODES=\$SLURM\_JOB\_NUM\_NODES, NTASKS=\$SLURM\_NTASKS, and OMP\_THREADS=\$OMP\_NUM\_THREADS, then run:

```
mpiexec -np $SLURM_NTASKS
./mpi_mat_prog MAT_FILE Problem A 2
```

In our case the input file was mawi\_201512020330.mat and the last argument (2) was the chosen synchronization frequency parameter for the MPI algorithm (our K\_sync in the report).

e) What this specific configuration means (the one we used): With -nodes=2 and -ntasks-per-node=1, the job runs with 2 MPI ranks total (one rank on each node). With -cpus-per-task=32 and OMP\_NUM\_THREADS=\$SLURM\_CPUS\_PER\_TASK, each MPI rank uses 32 OpenMP threads. So overall the job uses 2 ranks × 32 threads = 64 CPU threads across the two nodes.

## II. Results

This section summarizes the performance sweep extracted from our results table (runtime in seconds and number of outer synchronization rounds). Unless stated otherwise, runs use OMP\_THREADS=128 and one MPI rank per node (so NTASKS=NODES). We varied the number of nodes (1–7) and the synchronization frequency parameter K\_sync (1–5).

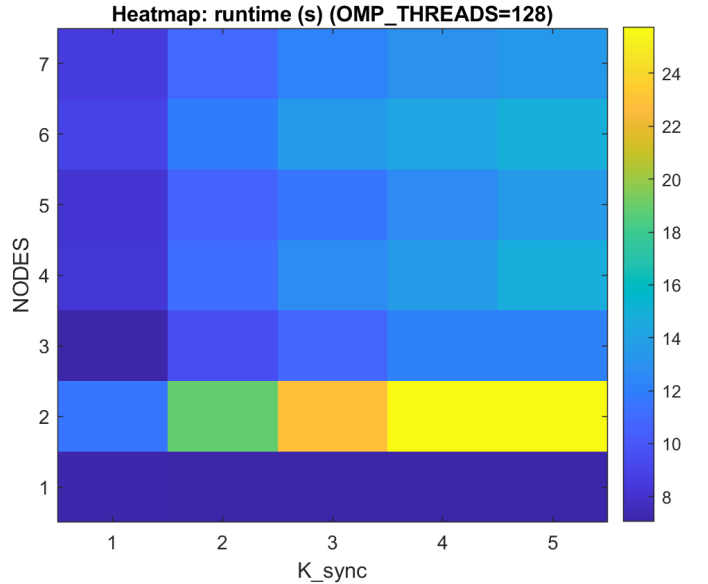


Figure 1. Runtime heatmap vs nodes and K\_sync (OMP\_THREADS=128).

a) Heatmap overview (Fig. 1): The heatmap highlights a very clear global trend: for every multi-node configuration tested, K\_sync=1 gives the lowest runtime. Increasing K\_sync always increases runtime, meaning that in this workload the extra local work between synchronizations does not translate into faster overall convergence. The 2-node row is also a visible outlier, showing substantially higher runtimes than the rest of the sweep, especially for larger K\_sync.

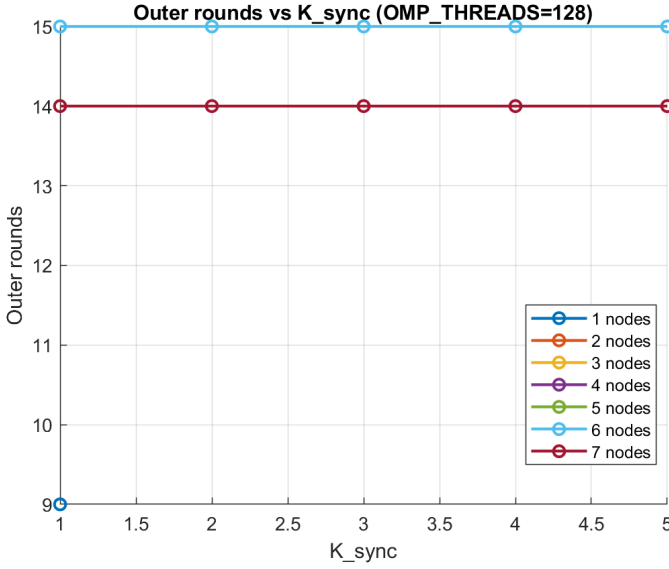


Figure 2. Outer synchronization rounds vs  $K_{\text{sync}}$  (OMP\_THREADS=128).

b) Convergence in outer rounds (Fig. 2): (3-5-7 nodes all do 14 outer rounds and 2-4-6 all do 15 outer rounds) The outer-rounds plot explains why larger  $K_{\text{sync}}$  performs poorly: for each fixed node count, the number of outer synchronization rounds is constant across  $K_{\text{sync}}$ . In our data, the 1-node run converged in 9 outer rounds (we only measured  $K_{\text{sync}}=1$  for 1 node), while all multi-node runs required either 14 rounds (3/5/7 nodes) or 15 rounds (2/4/6 nodes), independent of  $K_{\text{sync}}$ . Therefore, increasing  $K_{\text{sync}}$  does not reduce the number of expensive global synchronization steps; it mostly adds additional redundant computation between them.

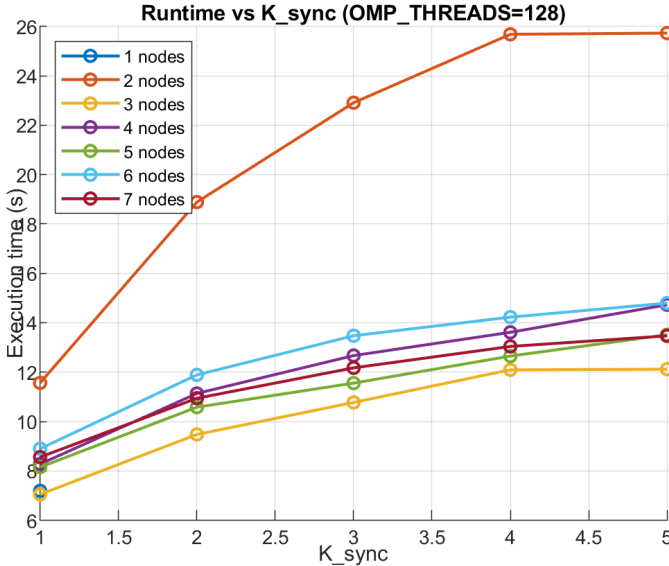


Figure 3. Runtime vs  $K_{\text{sync}}$  for different node counts (OMP\_THREADS=128).

c) Runtime vs  $K_{\text{sync}}$  (Fig. 3): The per-node curves show a near-monotonic runtime increase as  $K_{\text{sync}}$  grows. The slowdown from  $K_{\text{sync}}=1$  to  $K_{\text{sync}}=5$  is significant: for nodes 3–7 it is about  $1.57\times$ – $1.78\times$  (e.g., 3 nodes: 7.05s  $\rightarrow$  12.12s; 4 nodes: 8.28s  $\rightarrow$  14.73s), while for 2 nodes it is even worse at about  $2.22\times$  (11.57s  $\rightarrow$  25.73s). Practically, the best setting in this sweep is always the smallest tested value,  $K_{\text{sync}}=1$ .

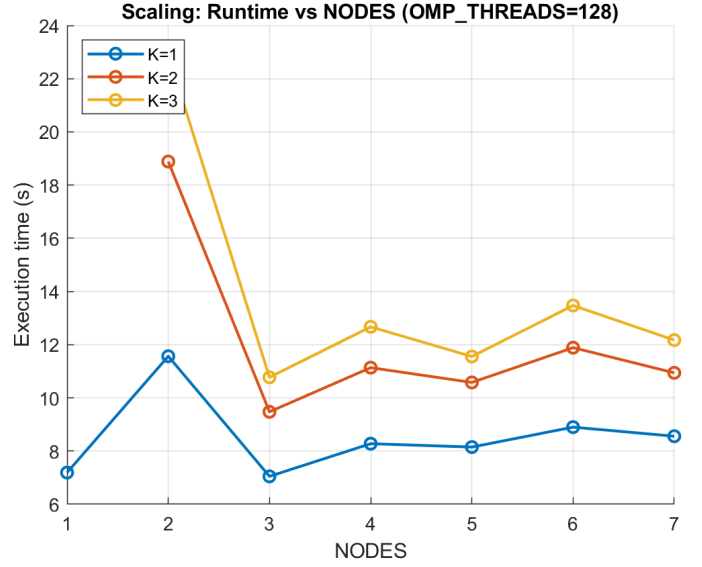


Figure 4. Scaling: runtime vs nodes for selected  $K_{\text{sync}}$  values (OMP\_THREADS=128).

d) Scaling with nodes (Fig. 4): Strong-scaling behavior is limited: runtime does not keep decreasing as nodes increase. For  $K_{\text{sync}}=1$ , the baseline 1-node runtime is 7.19s. The best observed runtime overall is 7.05s at 3 nodes (only about 2% faster than 1 node). Beyond 3 nodes, runtimes rise and then plateau around 8–9s (e.g., 7 nodes: 8.56s, which is  $\sim 19\%$  slower than 1 node). The 2-node point is an anomaly (11.57s), far worse than both 1 and 3 nodes; this suggests that at small node counts the run-to-run system effects and/or communication overhead can dominate and produce non-monotonic scaling.

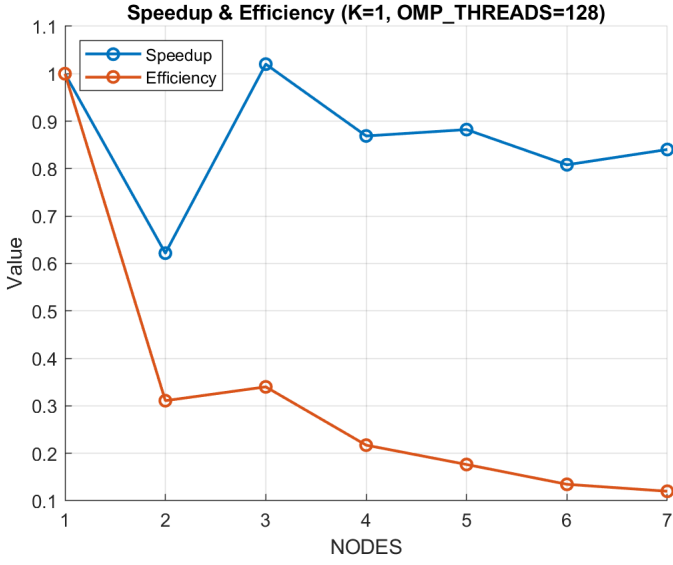


Figure 5. Speedup and efficiency vs nodes for  $K_{\text{sync}}=1$  (OMP\_THREADS=128).

e) Speedup and efficiency for  $K_{\text{sync}}=1$  (Fig. 5).: Using the 1-node,  $K_{\text{sync}}=1$  time (7.19s) as the reference, speedup is mostly below 1 (i.e., slowdown) except at 3 nodes. Specifically: 2 nodes achieve speedup 0.62 (efficiency 0.31), 3 nodes achieve the best speedup 1.02 (efficiency 0.34), and by 7 nodes speedup drops to 0.84 (efficiency 0.12). This pattern indicates that the implementation is limited by synchronization/communication costs (global coordination each outer step), so the benefits of distributing the local work are quickly outweighed as node count grows, especially since each rank is already heavily threaded (OMP\_THREADS=128).

f) Main takeaway.: For this dataset and the tested configuration, the best-performing region is small  $K_{\text{sync}}$  (1) and a modest node count (best observed at 3 nodes). Larger  $K_{\text{sync}}$  consistently increases runtime without improving the number of outer rounds, and scaling beyond 3 nodes yields diminishing returns or slowdown, consistent with synchronization/communication dominating the overall cost.