Connected Components Detection in Large Scale Graphs

Gerasimou Dimitrios AEM: 10813 and **Stavrianidis Giannis** AEM: 11065

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

Thessaloniki, Greece
November 2025

Abstract

This project efficiently identifies connected components in large undirected graphs containing millions of nodes and edges. We implemented two algorithmic approaches—label propagation and union-find—using three parallel programming paradigms (OpenMP, POSIX Threads, and OpenCilk) to exploit multicore parallelism. Results demonstrate speedups ranging from 1.89× to 5.39× on 8 threads across seven diverse datasets, with OpenCilk achieving the best performance for label propagation. We employ compressed sparse column (CSC) format for memory-efficient graph representation and optimize performance through four key techniques: work chunking, partial pointer jumping, bitmap-based component counting, and thread-local variables.

1 Introduction

A connected component is a maximal subgraph in which every vertex is reachable from any other vertex. Identifying these components is fundamental in graph analysis, with applications in social network analysis, web graph clustering, and biological network studies. The challenge lies in processing graphs with hundreds of millions of nodes and edges within realistic memory and runtime constraints. This work focuses on parallel implementations that leverage modern multicore architectures to accelerate computation while maintaining correctness.

2 Graph Representation

We represent graphs using a custom Compressed Sparse Column (CSC) binary matrix structure that stores only the adjacency structure, discarding numerical values:

```c
typedef struct {
        size_t   nrows, ncols;    /* Matrix dimensions */
        size_t   nnz;             /* Number of edges */
        uint32_t *row_idx;        /* Row indices (length = nnz) */
        uint32_t *col_ptr;        /* Column pointers (length = ncols + 1) */
} CSCBinaryMatrix;
```

3 Algorithmic Approaches

3.1 Label Propagation (Variant 0)

The label propagation algorithm iteratively updates each vertex's label to the minimum label among its neighbors. Vertices in the same connected component eventually converge to the same label. This approach is naturally parallel-friendly since vertex updates within an iteration can be performed independently. We parallelize the neighbor inspection loop using work chunking—dividing graph columns into contiguous blocks and assigning each block to a thread. Convergence is detected when no vertex changes its label in an iteration. To optimize performance, we incorporate partial pointer jumping, where vertices can quickly adopt distant labels by following chains, significantly reducing iteration counts.

3.2 Union-Find (Variant 1)

The union-find approach uses a disjoint-set data structure with path halving to efficiently merge vertex sets as edges are processed. Each vertex initially belongs to its own set, and sets are merged when edges are encountered. We parallelize by dividing edges among threads for local merging, then combining parent pointers using thread-local variables and OpenMP reductions or Cilk reducers to avoid race conditions. While union-find is typically the fastest sequential algorithm, its work is less naturally divisible, resulting in smaller relative speedups compared to label propagation despite better absolute performance.

4 Parallelization Strategies

We implemented three parallel programming paradigms:

- **OpenMP:** Compiler-directive-based parallelism using: `#pragma omp parallel for` with `schedule(static, chunk_size)` for load-balanced work distribution
- **POSIX Threads:** Manual thread management with explicit work division, where each thread processes a contiguous block of columns
- **OpenCilk:** Work-stealing scheduler using `cilk_for` for dynamic load balancing, particularly effective for irregular workloads

Each paradigm offers distinct trade-offs. OpenMP provides ease of implementation with good performance. PThreads offer fine-grained control but requires manual synchronization. OpenCilk excels at load balancing through work stealing, making it particularly effective for irregular graphs.

5 Performance Optimizations

We implemented four key optimizations: **(1) Work chunking** groups columns into chunks to reduce scheduling overhead and improve cache locality. **(2) Pointer jumping** enables vertices to adopt distant labels by following parent chains, accelerating convergence. **(3) Bitmap-based counting** replaces quicksort with constant-time membership testing for component identification. **(4) Thread-local variables** with reductions eliminate atomic operations and false sharing, improving scalability.

6 Experimental Results

System Configuration: Intel Core i7-11800H (8 cores), 16GB RAM, 16GB Swap on Arch Linux.

Index	Dataset	Nodes	Edges	Result	V0 Best (×)	V1 Best (×)
1	dictionary28	52,652	178,076	17,903	3.90 (OpenMP)	2.59 (OpenMP)
2	hollywood-2009	1,139,905	113,891,327	44,508	4.73 (OpenCilk)	2.60 (OpenMP)
3	kmer_V1r	214,005,017	465,410,904	9	3.14 (OpenCilk)	2.73 (OpenCilk)
4	kmer_A2a	170,728,175	360,585,172	5353	2.40 (OpenMP)	2.75 (OpenCilk)
5	com-LiveJournal	3,997,962	69,362,378	1	5.11 (OpenCilk)	3.82 (OpenMP)
6	mawi	226,196,185	480,047,894	3,971,144	1.89 (OpenMP)	2.11 (OpenCilk)
7	com-Orkut	3,072,441	234,370,166	1	5.39 (OpenCilk)	5.10 (OpenMP)

Table 1: Dataset characteristics and best speedup achieved by each algorithm variant on 8 threads.
V0 = Label Propagation, V1 = Union-Find. Result column shows number of connected components found.

Figures 1 and 2 compare implementation speedups across datasets for variants 0 and 1 respectively, while Figure 3 examines strong scaling behavior.

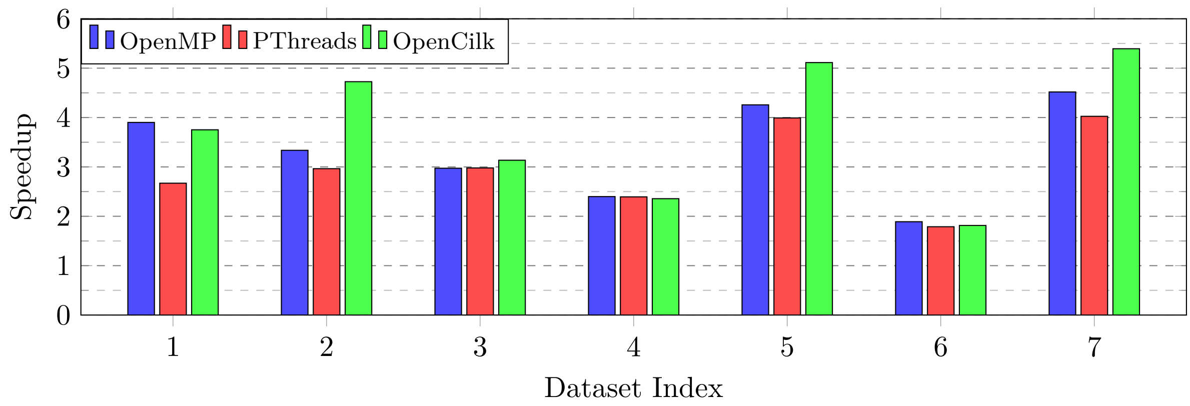

Figure 1: Speedup comparison across seven datasets for label propagation (Variant 0) using OpenMP, PThreads, and OpenCilk on 8 threads.

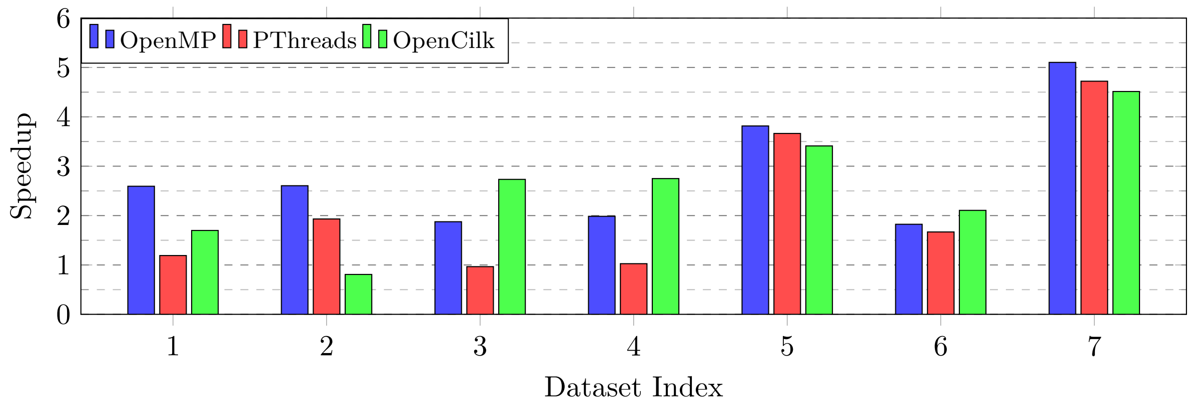

Figure 2: Speedup comparison across seven datasets for union-find (Variant 1) using OpenMP, PThreads, and OpenCilk on 8 threads.

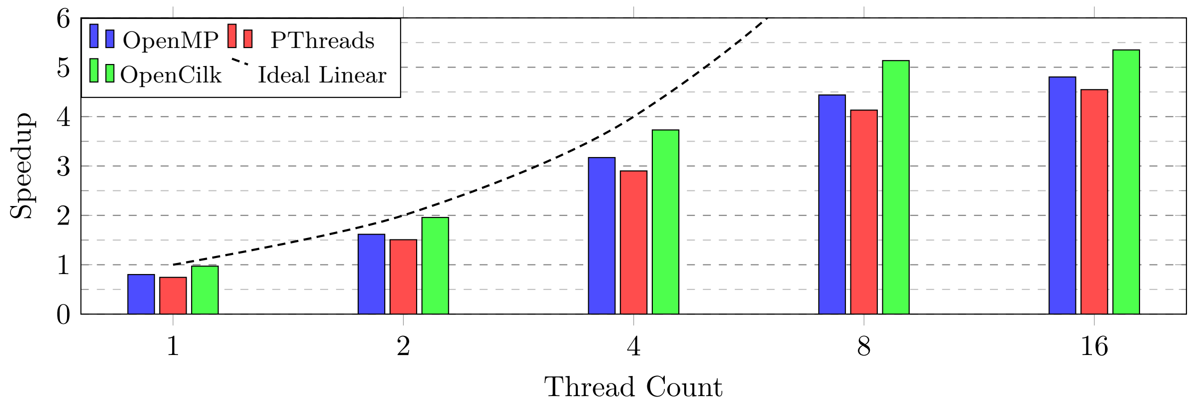

Figure 3: Strong scaling analysis for label propagation on LiveJournal dataset (1-16 threads). Dashed line indicates ideal linear speedup.

7 Analysis and Discussion

7.1 Key Findings

- **Algorithm Behaviour:** Label propagation (V0) consistently achieves higher speedups (3.14×–5.39×) compared to union-find (V1), with efficiencies ranging from 39–67%. Union-find shows better absolute performance but lower parallelization benefits due to more complex data dependencies.
- **Implementation Comparison:** OpenCilk demonstrates superior performance for label propagation on most datasets, benefiting from its work-stealing scheduler. OpenMP achieves competitive results

with simpler code. PThreads consistently shows slightly lower performance due to static work partitioning.

- **Dataset Characteristics:** Social networks (Orkut, LiveJournal) exhibit excellent speedups (5.11×–5.39×) due to well-balanced component structures. Large sparse graphs (MAWI, Kmer) show modest speedups (1.89×–3.14×), likely due to memory bandwidth limitations and irregular access patterns that reduce parallel efficiency.

7.2 Notable Observations

The Hollywood dataset exhibits surprising OpenCilk union-find slowdown (0.81×), likely from excessive work-stealing overhead on irregular edge distributions. Efficiency metrics reveal none exceed 68%, indicating memory bandwidth, rather than computation, becomes the bottleneck—consistent with the sub-linear scaling in Figure 3, where speedup plateaus beyond 4 threads.

7.3 Strong Scaling Analysis

Figure 3 demonstrates strong scaling behavior for label propagation on LiveJournal. All implementations achieve near-linear speedup up to 4 threads (OpenCilk: 3.6×, OpenMP: 3.4×), but show diminishing returns at 8 threads (OpenCilk: 5.1×, efficiency 64%) and saturation at 16 threads. This plateau reflects Amdahl's Law limitations and memory bandwidth saturation. The gap between actual and ideal linear speedup (dashed line) quantifies parallelization overhead, with OpenCilk's work-stealing providing 15-20% advantage over static partitioning at higher thread counts.

7.4 Correctness Verification

All implementations were verified by comparing component counts across sequential and parallel versions. Identical results across all implementations confirm correctness, with deterministic outcomes being the main goal, despite non-deterministic thread scheduling.

8 Conclusions

This project successfully demonstrates parallel connected components detection across diverse graph datasets using multiple programming paradigms. OpenCilk with label propagation achieves the best results on social network graphs (up to 5.39× speedup with 67% efficiency), while showing that memory-bound operations limit scalability on larger sparse graphs. The implemented optimizations— work chunking, pointer jumping, bitmap-based counting, and thread-local variables—prove essential for achieving good parallel performance.

8.1 Future Directions

Several promising directions for future work include:

- Exploring distributed-memory implementations using MPI for graphs exceeding single-node capacity
- Investigating GPU acceleration using CUDA or OpenCL for massive parallelism
- Implementing hybrid algorithms that adaptively switch between label propagation and union-find based on graph characteristics
- Optimizing memory access patterns through graph reordering and compression techniques
- Conducting deeper analysis of cache performance using hardware counters (cache misses, memory bandwidth utilization)

Code Availability: Complete source code, build instructions, and benchmark scripts are available at https://github.com/gianst2004/pardis2025