Athanasios Papadopoulos
11029
atpapadop@ece.auth.gr

Homework Assignment 1
Connected Components of an Undirected Graph

2025-11-19

1 Introduction

A connected component (CC) of an undirected graph is a maximal set of vertices that are mutually reachable. This report studies sequential and parallel methods for computing CCs using an iterative label-propagation (coloring) algorithm, chosen for its simplicity and suitability for parallel execution.
I implemented one sequential and three parallel variants using OpenMP, OpenCilk, and POSIX threads (Pthreads). A BFS-based CC algorithm and MATLAB's `conncomp` function serve as correctness references. Experiments are conducted on two large SNAP graphs (`com-LiveJournal` and `com-Orkut`) from the SuiteSparse collection. The report highlights data structures, parallelization strategies, and the impact of thread count and chunk size on performance.

2 Graph Representation

All implementations use a compressed sparse row (CSR) graph representation:

```
typedef struct {
  int32_t n;        // vertices
  int64_t m;        // edges
  int64_t *row_ptr; // offsets (size n+1)
  int32_t *col_idx; // adjacency (size m)
} CSRGraph;
```

CSR is compact, cache-friendly, and well-suited for large-scale irregular graphs.
The datasets used are:

- **com-LiveJournal**: $n = 3{,}997{,}962$, $m = 69{,}362{,}378$,
- **com-Orkut**: $n = 3{,}072{,}441$, $m = 117{,}185{,}083$.

The design generalizes easily to additional datasets without modifying the algorithmic core.

3 Connected Components Algorithms

3.1 Sequential Label Propagation

The sequential algorithm initializes each vertex with a unique label and repeatedly relaxes labels by taking the minimum among neighbors. Each iteration resembles a sparse matrix–vector multiplication where arithmetic operators are replaced by *min* and *copy*. The algorithm converges when no labels change. Because updates are local and independent, this approach parallelizes naturally.

3.2 BFS Baseline

A BFS-based CC implementation is included for correctness and timing comparison. It processes components one at a time by performing BFS from each unvisited vertex. This method is simple and deterministic but significantly slower on large, irregular graphs.

4 Parallel Implementations

The three parallel versions follow the same label-relaxation structure but differ in work partitioning and synchronization. Since many vertices stabilize early, strict global ordering is unnecessary, and reducing synchronization substantially improves performance, though at the cost of strict repeatability.

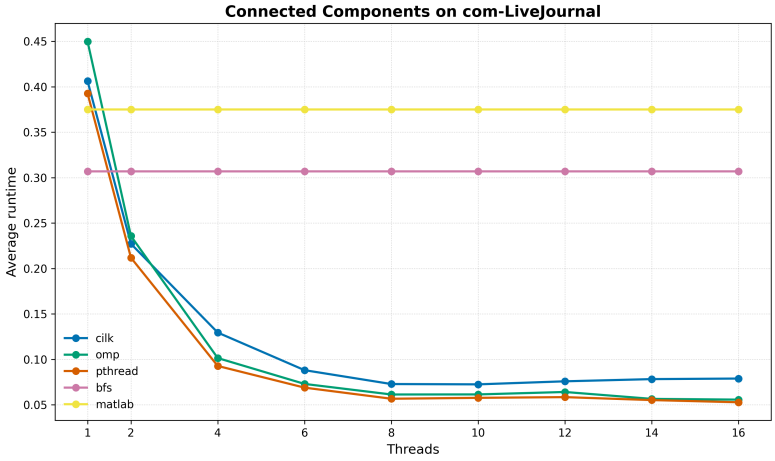

Figure 1: Runtime vs. threads for `com-LiveJournal`.

4.0.1 Chunking Strategy

Chunking is essential across models to prevent idle threads. Small chunks provide good load balance but incur scheduling overhead, while large chunks reduce overhead but risk imbalance. Because label values stabilize quickly, chunk-based dynamic scheduling avoids repeatedly processing vertices whose labels no longer change. Enabling chunking itself yields a significant improvement, with exact tuning being secondary factor.

4.1 OpenMP

The OpenMP version parallelizes the main loop with `#pragma omp parallel for` and uses an atomic label array for thread-safe updates. Dynamic scheduling (`schedule(dynamic,chunk_size)`) improves balance for graphs with skewed degree distributions. A reduction detects convergence.

4.2 OpenCilk

OpenCilk uses `cilk_for` loops, dividing vertices into manually sized chunks while relying on work stealing to distribute tasks. Work stealing mitigates some of the issues that arise from poorly chosen chunk sizes, though overhead is still slightly higher than in OpenMP or Pthreads for these graphs.

4.3 Pthreads

Pthreads exposes parallelism most explicitly: each thread repeatedly fetches a block of vertices from a shared atomic counter and processes them independently. A barrier synchronizes iterations, and a shared flag detects convergence. This design affords precise control over granularity and synchronization and achieved the best performance overall.

5 Experimental Setup and Results

Experiments were conducted on a 16-core machine, with all binaries compiled using `-O3`. Each executable performs multiple timed runs and logs results for Python-based post-processing.
The evaluation consists of two main experiments:

- **Thread-scaling experiment:** runtimes are measured for thread counts from 1 to 16 using a fixed chunk size of 4096 across all parallel methods.
- **Chunk-size sweep (Pthreads only):** chunk sizes from 1 to 16384 are tested across multiple thread counts to study the effect of granularity on performance.

5.1 Thread Scaling

Figures 1 and 2 show scaling results. All parallel methods scale well up to roughly 12 threads, with diminishing returns thereafter. On the denser `com-Orkut` graph, OpenCilk's scaling degrades more noticeably, whereas Pthreads and OpenMP remain stable.

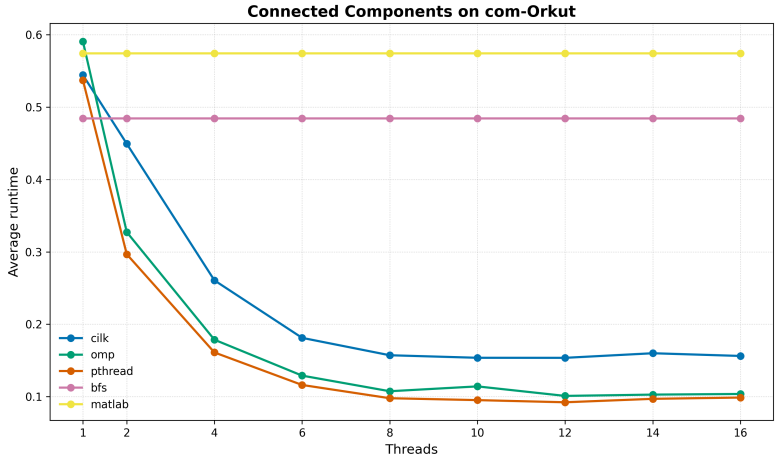

Figure 2: Runtime vs. threads for `com-Orkut`.

(a) `com-LiveJournal` (16 threads)

Method	Min [ms]	Mean [ms]
BFS	303.8	306.8
MATLAB	375.0	365.4
OpenMP	54.9	55.8
OpenCilk	72.5	78.9
Pthreads	51.2	52.8

(b) `com-Orkut` (12 threads)

Method	Min [ms]	Mean [ms]
BFS	481.1	484.5
MATLAB	560.0	574.0
OpenMP	100.7	101.0
OpenCilk	149.4	153.6
Pthreads	90.8	92.2

Table 1: Runtime summaries for both graphs.

Tables 1a and 1b summarize performance for the best-performing thread counts (16 for LiveJournal, 12 for Orkut).
Pthreads achieves the lowest overall runtimes, with OpenMP following closely and OpenCilk lagging behind on both datasets. Relative to the sequential BFS baseline, both Pthreads and OpenMP deliver approximately a fivefold speedup on `com-LiveJournal`, and this improvement increases further on the denser `com-Orkut` graph. In contrast, OpenCilk achieves only a three- to fourfold improvement, and its relative advantage decreases as graph density grows.

These patterns align with the scaling behavior observed in the thread-scaling plots: for the denser `com-Orkut` graph, Pthreads and OpenMP maintain strong scaling efficiency, whereas OpenCilk's performance degrades more noticeably. This suggests that the explicit work distribution mechanisms in Pthreads and OpenMP handle heavy-degree vertices more effectively than Cilk's work-stealing runtime, whose overhead becomes increasingly visible on denser graphs.

5.2 Chunk Size Sensitivity

Chunk-size effects were evaluated on `com-LiveJournal` using Pthreads, with chunk sizes from 1 to 16384 and thread counts from 1 to 16. Figures 3–4 show both the 3D performance surface and its projection.
Small chunk sizes increase scheduling overhead, whereas very large chunk sizes risk load imbalance. This sensitivity is tied to graph density, which affects the amount of work per vertex, and thus per chunk. Chunk sizes between 2048 and 4096 generally perform best across thread counts. More importantly, enabling chunking itself matters far more than the exact value as shown in Table 2, turning off chunking entirely (chunk size $= 1$) leads to substantial slowdown. This effect is strongest in Pthreads and in OpenMP when using `schedule(dynamic,chunk_size)`. OpenCilk mitigates the issue somewhat via work stealing but still suffers some performance drop when chunking is absent.

Another thing to note is that, looking at Figure 4 and Table 2, we can conclude that, due to the somewhat chaotic behaviour of performance relative to chunk size, choosing the appropriate value is not always straightforward. In this experiment, for example, it appears that the performance of Chunk sizes 2048, 8192 and 16384 for 8 Threads matches that of Chunk size 4096 for 16 Threads.

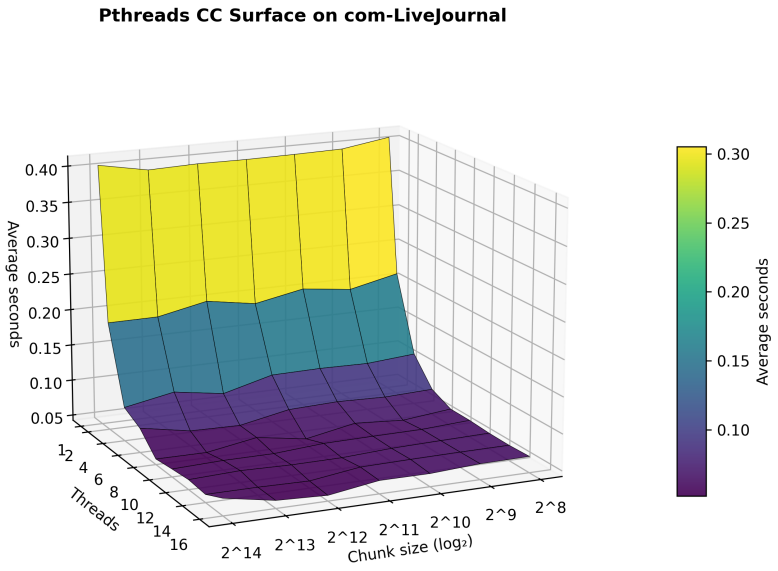

Figure 3: Pthreads runtime surface for `com-LiveJournal`.

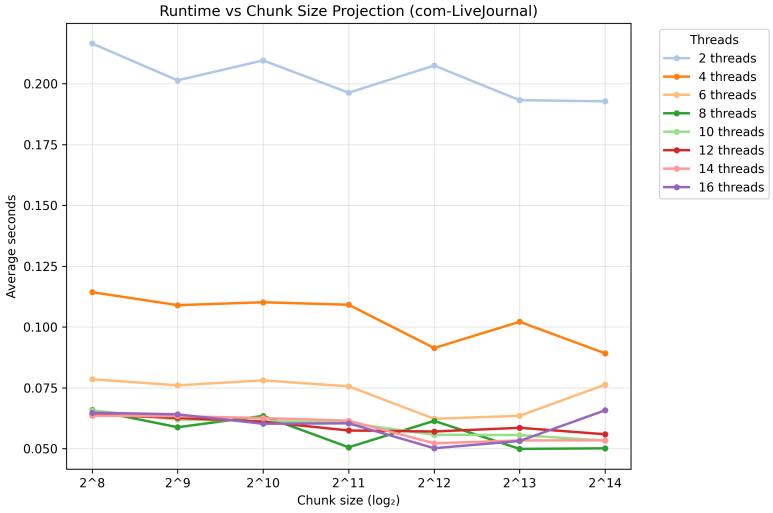

Figure 4: Runtime vs. chunk size (projection) for `com-LiveJournal`.

6 Conclusions

Label propagation is well-suited to parallel execution due to its locality, tolerance to relaxed synchronization, and naturally convergent behavior. Among the evaluated implementations, Pthreads consistently delivered the strongest performance, with OpenMP following closely. OpenCilk scales correctly but was less effective on the tested graphs. Chunking proved essential across all methods: although the best chunk size varies, simply enabling chunk-based dynamic work distribution provides the largest performance gain. MATLAB's `conncomp` and the BFS baseline served as useful correctness references but were significantly slower on large graphs. Overall, the dominant performance factors were reducing synchronization and using dynamic, chunk-based work distribution. While fine-grained tuning can produce incremental benefits, the main improvements arise from adopting relaxed synchronization and appropriate parallel work partitioning across all models.

Future Work

Future improvements could include NUMA-aware scheduling to better utilize multi-socket systems, as current implementations do not control memory locality. Larger datasets, especially on the scale of billions of edges, would help stress-test scalability further. Finally, while chunk-size sweeps were performed for Pthreads, a full parameter sweep across OpenMP and OpenCilk would allow identifying true optimal settings for each runtime system.

Table 2: Pthreads runtime (*ms*) vs. chunk size for `com-LiveJournal`.

Chunk size\Threads	6T	8T	10T	12T	14T	16T
1	494.996	487.756	424.581	391.883	361.023	348.066
256	78.561	65.909	65.749	63.958	63.562	64.806
512	76.047	58.834	61.970	62.650	63.368	64.140
1024	78.086	63.475	61.449	61.118	62.599	60.252
2048	75.639	50.595	60.577	57.489	61.528	60.435
4096	62.285	61.438	55.730	57.065	52.235	50.209
8192	63.528	49.931	55.622	58.597	53.410	53.174
16384	76.331	50.161	53.299	55.940	53.426	65.790

Code availability. All code, plotting scripts, and instructions are available at:

`https://github.com/AtPapadop/CC`