Athanasios Papadopoulos
11029
atpapadop@ece.auth.gr

Homework Assignment 1
Connected Components of an Undirected Graph

2025-11-16

1 Introduction

A connected component (CC) of an undirected graph is a maximal set of vertices that are mutually reachable. This report studies parallel algorithms for computing CCs using a label-propagation (coloring) approach, chosen for its simplicity and parallel-friendly structure.
I implemented one sequential and three parallel versions of the algorithm using OpenMP, OpenCilk, and POSIX threads (Pthreads). A BFS-based CC routine and MATLAB's `conncomp` serve as correctness references. Experiments focus on two large graphs from the SuiteSparse Collection: `com-LiveJournal` and `com-Orkut`. The report emphasizes data structures, parallelization choices, and performance characteristics with respect to thread count and chunk size.

2 Graph Representation

All implementations use a compressed sparse row (CSR) 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 allows efficient traversal of large, irregular graphs.
The evaluated datasets are:

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

The code structure accommodates additional datasets with no modifications to the algorithmic core.

3 Connected Components Algorithms

3.1 Sequential Label Propagation

The label-propagation method assigns each vertex its own index as an initial label and repeatedly relaxes labels by taking the minimum among neighbors. Each iteration resembles a sparse matrix–vector multiplication where addition and multiplication are replaced with *min* and *copy*. The process converges when no labels change.
This algorithm converges relatively quickly in practice and is far more parallel-friendly than BFS-based approaches, since label updates are local and tolerate relaxed synchronization.

3.2 BFS Baseline

A BFS-based implementation is used as a correctness and performance reference. It processes components one by one, starting BFS from each unvisited vertex. While simple and deterministic, it is significantly slower on large irregular graphs and harder to parallelize efficiently.

4 Parallel Implementations

All parallel versions follow the same iterative label-relaxation structure but differ in how they distribute work and synchronize. A key feature of the algorithm is that it converges: many vertices stop changing early, so strict global ordering is unnecessary. Reducing synchronization overhead is essential for good performance although it does sacrifice strict repeatability.

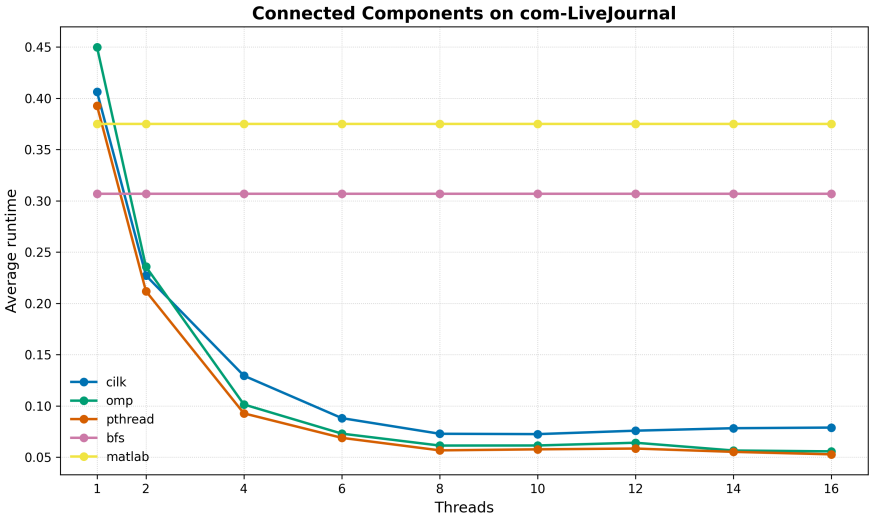


Figure 1: Runtime vs. threads for `com-LiveJournal`.

4.0.1 Chunking Strategy

Chunking is essential across all models. Smaller chunks improve load balance but increase scheduling overhead; larger chunks reduce overhead but risk imbalance. Because label propagation quickly stabilizes many vertices, dynamic chunk-based scheduling avoids wasting work on vertices that no longer change. While the exact chunk size matters less than its presence, enabling chunking at all yields substantial improvements.

4.1 OpenMP

The OpenMP version parallelizes the main vertex loop with `#pragma omp parallel for` and uses an atomic label array (`_Atomic int32_t`) for safe updates. Dynamic scheduling with a tunable chunk size improves load balance on graphs with skewed degree distributions. A reduction detects whether any label changed during an iteration.

4.2 OpenCilk

The OpenCilk implementation uses `cilk_for` loops, with vertices divided into manually sized chunks. Work stealing provides automatic balancing, though interaction with the simple per-vertex relaxation loop results in slightly higher overhead than OpenMP on these datasets.

4.3 Pthreads

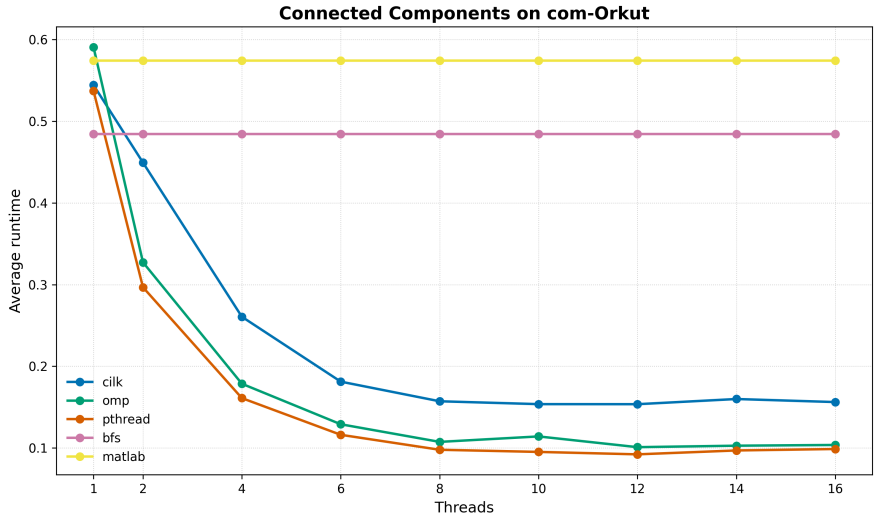
The Pthreads implementation exposes the parallelization more explicitly. Threads repeatedly fetch blocks of vertices from a shared atomic counter and process them independently. A barrier synchronizes iterations, and a shared atomic flag detects convergence. This model gives the most control over work granularity and synchronization, and it produced the best performance overall.

5 Experimental Setup and Results

Experiments were run on a 16-core machine with binaries compiled using `-O3`. Each executable performs multiple timed runs and logs results for Python-based post-processing.
Thread-scaling experiments use thread counts from 1 to 16 with a chunk size of 4096 (which performed well across both graphs and various thread counts). Additional sweeps vary the chunk size to evaluate its effect.

5.1 Thread Scaling

Figures 1 and 2 show runtime as a function of thread count for both graphs.
All parallel methods improve rapidly up to about 12 threads, after which returns diminish. Differences between models become more apparent on the denser `com-Orkut` graph: OpenCilk scaling degrades more noticeably, while OpenMP and especially Pthreads remain robust.



Figure

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 numerical results. For each graph, the best-performing thread count (16 for LiveJournal, 12 for Orkut) is reported.
Pthreads achieves the best performance on both datasets, followed closely by OpenMP. OpenCilk consistently trails behind. MATLAB and BFS are substantially slower.

5.2 Chunk Size Sensitivity

To study chunk size, I ran a sweep on `com-LiveJournal` using Pthreads, measuring runtime across chunk sizes from 1 to 16384 and thread counts from 1 to 16. Figures 3 and 4 show the 3D performance surface and its projection while Table 2 shows a summary of the mean runtime for various chunk size-thread cobinations.
Small chunk sizes incur scheduling overhead, while very large chunk sizes can produce load imbalance. I theorize that this is largely dependent on the density of the graph as that directly correlates with how much work each chunk requires from the thread. Nevertheless, chunk sizes between 2048 and 4096 perform consistently well across thread counts. More importantly, enabling chunking in the first place is more impactful than tuning its exact value, as can be see in Table 2, where for chunk size 1 (effectively no chunking) performance plummets.

6 Conclusions

Label propagation is a naturally parallel algorithm for identifying connected components, and it benefits substantially from relaxed synchronization and dynamic, chunk-based work distribution. Among the parallel models, Pthreads delivered the best performance, with OpenMP close behind and OpenCilk somewhat less effective on these inputs.
The most important performance factors were reducing synchronization and introducing chunking to avoid idle threads, while the exact chunk size played only a secondary role. MATLAB's `conncomp` and the BFS implementation were useful for correctness and reference but significantly slower for large graphs.

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

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

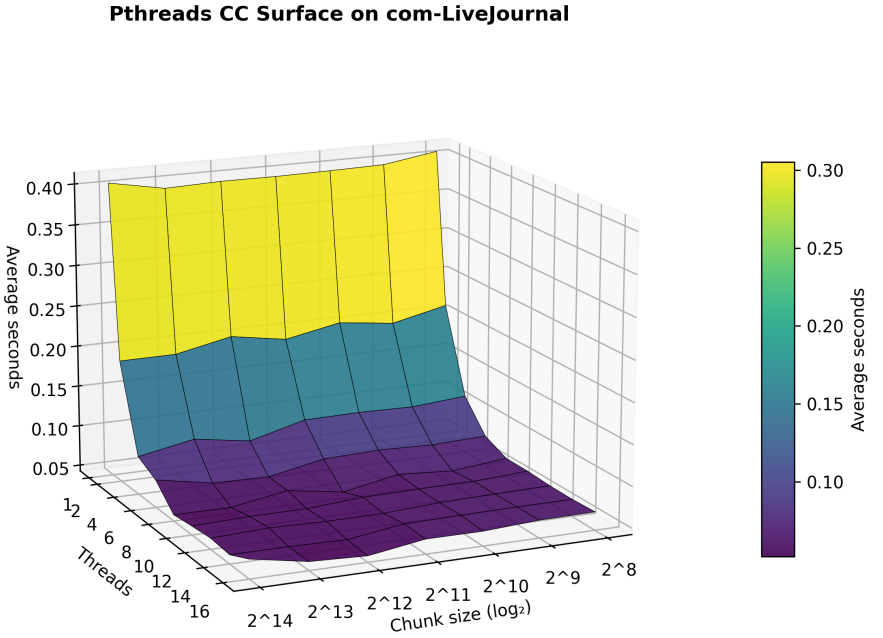


Figure 3: Pthreads runtime surface for `com-LiveJournal`.

Table 2: Pthreads runtime 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

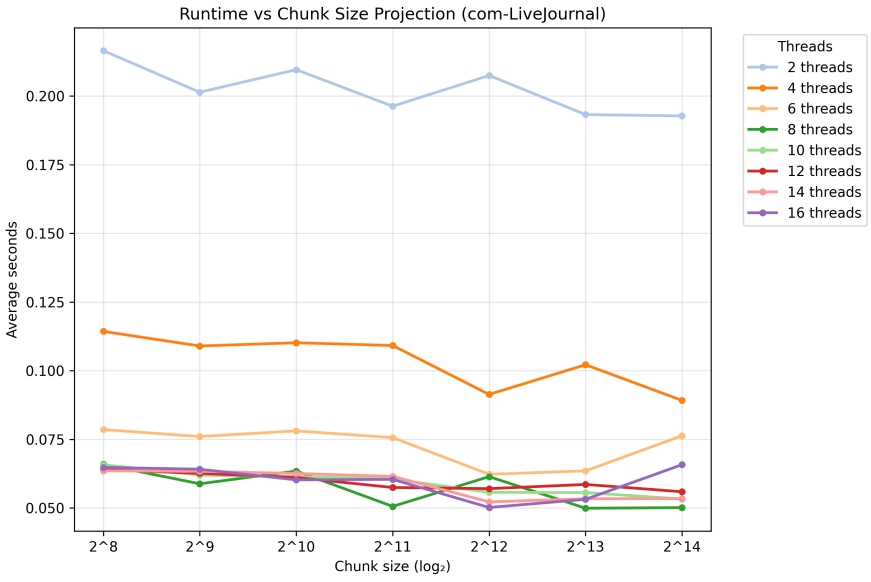


Figure 4: Runtime vs. chunk size (projection).