Project 3-B: Big Graph Processing

Last Updated: October 29, 2025

In this project, you will implement breadth-first search (BFS). A good implementation of this assignment will be able to run this algorithm on graphs containing hundreds of millions of edges on a multi-core machine in only seconds.

Contents

1 Due dates

- *Final submission & write-up:* 10:00 P.M. on Monday, December 8, 2025

2 Getting started

The assignment starter code is available on Github:

```
https://github.com/spe-fall25/project3-B_<your netid>.git
```

2.1 Background: Learning OpenMP

In this assignment we'd like you to use OpenMP for multi-core parallelization. OpenMP is an API and set of C-language extensions that provides compiler support for parallelism. You can

also use OpenMP to tell the compiler to parallelize iterations of `for` loops, and to manage mutual exclusion. It is well documented online, but here is a brief example of parallelizing a `for` loop, with mutual exclusion.

```
/* The iterations of this for loop may be parallelized by the compiler */
#pragma omp parallel for
    for (int i = 0; i < 100; i++) {
   /* different iterations of this part of the loop body may be  run in parallel on different cores */
    #pragma omp critical {
    /* This block will be executed by at most one thread at a time. */
    printf("Thread %d got iteration %lu\n", omp_get_thread_num(), i);
    }
}
```

Please see OpenMP documentation for the syntax for how to tell OpenMP to use different forms of static or dynamic scheduling. (For example, `omp parallel for schedule(dynamic 100)` distributes iterations to threads using dynamic scheduling with a chunk size of 100 iterations). You can think of the implementation as a dynamic work queue where threads in the thread pool pull off 100 iterations at once, like in these lecture slides.

Here is an example for an atomic counter update in OpenMP.

```
int my_counter = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
    if ( ... some condition ...) {
        #pragma omp atomic
        my_counter++;
    }
}
```

We expect you to be able to read OpenMP documentation on your own (Google will be very helpful), but here are some useful links to get you started:

- The OpenMP 3.0 specification: http://www.openmp.org/mp-documents/spec30.pdf.
- An OpenMP cheat sheet http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf.
- OpenMP has support for reductions on shared variables, and for declaring thread-local copies of variables.
- This is a nice guide for the `omp parallel_for` directives: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf

2.2 Background: Representing Graphs

The starter code operates on directed graphs, whose implementation you can find in `graph.h` and `graph_internal.h`. We recommend you begin by understanding the graph representation in these files. A graph is represented by an array of edges (both `outgoing_edges` and `incoming_edges`), where each edge is represented by an integer describing the id of the destination vertex. Edges are stored in the graph sorted by their source vertex, so the source vertex is implicit in the representation. This makes for a compact representation of the graph, and also allows it to be stored contiguously in memory. For example, to iterate over the outgoing edges for all nodes in the graph, you'd use the following code which makes use of convenient helper functions defined in `graph.h` (and implemented in `graph_internal.h`):

```
for (int i=0; i<num_nodes(g); i++) {
    // Vertex is typedef'ed to an int. Vertex* points into g.outgoing_edges[]
    const Vertex* start = outgoing_begin(g, i);
    const Vertex* end = outgoing_end(g, i);
    for (const Vertex* v=start; v!=end; v++)
    printf("Edge %u %u\n", i, *v);
}
```

2.3 Dataset

In this project, you will use a large graph dataset to test the performance. The dataset can be found at http://cs149.stanford.edu/cs149asstdata/all_graphs.tgz. You can download the dataset using `wget http://cs149.stanford.edu/cs149asstdata/all_graphs.tgz`, and then untar it with `tar -xzvf all_graphs.tgz`. **Be careful, this is a 3 GB download.**

Some interesting real-world graphs include:

- com-orkut_117m.graph
- oc-pokec_30m.graph
- soc-livejournal1_68m.graph

Your useful synthetic, but large graphs include:

- random_500m.graph
- rmat_200m.graph

There are also some very small graphs for testing. If you look in the `/tools` directory of the starter code, you'll notice a useful program called `graphTools.cpp` that can be used to make your own graphs as well.

3 Part 1: Parallel "Top Down" Breadth-First Search

Breadth-first search (BFS) is a common algorithm that might have seen in a prior algorithms class (See here and here for helpful references.) Please familiarize yourself with the function `bfs_top_down()` in `bfs/bfs.cpp`, which contains a sequential implementation of BFS. The code uses BFS to compute the distance to vertex 0 for all vertices in the graph. You may wish to familiarize yourself with the graph structure defined in `common/graph.h` as well as the simple array data structure `vertex_set` (`bfs/bfs.h`), which is an array of vertices used to represent the current frontier of BFS.

You can run bfs using:

```
./bfs <PATH_TO_GRAPHS_DIRECTORY>/rmat_200m.graph
```

where `<PATH_TO_GRAPHS_DIRECTORY>` is the path to the directory containing the graph files (see the "Dataset" section above).

When you run `bfs`, you'll see execution time and the frontier size printed for each step in the algorithm. Correctness will pass for the top-down version (since we've given you a correct sequential implementation), but it will be slow. (Note that `bfs` will report failures for a "bottom up" and "hybrid" versions of the algorithm, which you will implement later in this assignment.)

In this part of the assignment your job is to parallelize top-down BFS. You'll need to focus on identifying parallelism, as well as inserting the appropriate synchronization to ensure correctness. We wish to remind you that you should not expect to achieve near-perfect speedups on this problem (we'll leave it to you to think about why!).

Tips/Hints:

- Always start by considering what work can be done in parallel.
- Some parts of the computation may need to be synchronized, for example, by wrapping the appropriate code within a critical region using `#pragma omp critical` or `#pragma omp atomic`. However, in this problem you should think about how to make use of the simple atomic operation called `compare and swap`. You can read about GCC's implementation of compare and swap, which is exposed to C code as the function `__sync_bool_compare_and_swap`. If you can figure out how to use compare-and-swap for this problem, you will achieve much higher performance than using a critical region.
- Updating a shared counter can be done efficiently using `#pragma omp atomic` before a line like `counter++;`.
- Are there conditions where it is possible to avoid using `compare_and_swap`? In other words, when you *know* in advance that the comparison will fail?
- There is a preprocessor macro `VERBOSE` to make it easy to disable useful print per-step timings in your solution (see the top of `bfs/bfs.cpp`). In general, these printfs occur infre-

quently enough (only once per BFS step) that they do not notably impact performance, but if you want to disable the printfs during timing, you can use this `#define` as a convenience.

4 Part 2: "Bottom Up" BFS

Think about what behavior might cause a performance problem in the BFS implementation from Part 1.2. An alternative implementation of a breadth-first search step may be more efficient in these situations. Instead of iterating over all vertices in the frontier and marking all vertices adjacent to the frontier, it is possible to implement BFS by having *each vertex check whether it should be added to the frontier!* Basic pseudocode for the algorithm is as follows:

```
for each vertex v in graph:
   if v has not been visited AND
   v shares an incoming edge with a vertex u on the frontier:
       add vertex v to frontier;
```

This algorithm is sometimes referred to as a "bottom up" implementation of BFS, since each vertex looks "up the BFS tree" to find its ancestor. (As opposed to being found by its ancestor in a "top down" fashion, as was done in Part 1.2.)

Please implement a bottom-up BFS to compute the shortest path to all the vertices in the graph from the root (see `bfs_bottom_up()` in `bfs/bfs.cpp`). Start by implementing a simple sequential version. Then parallelize your implementation.

Tips/Hints:

- It may be useful to think about how you represent the set of unvisited nodes. Do the top-down and bottom-up versions of the code lend themselves to different implementations?
- How do the synchronization requirements of the bottom-up BFS change?

5 Part 3: Hybrid BFS

Notice that in some steps of the BFS, the "bottom up" BFS is significantly faster than the top-down version. In other steps, the top-down version is significantly faster. This suggests a major performance improvement in your implementation, if you could dynamically choose between your "top down" and "bottom up" formulations based on the size of the frontier or other properties of the graph! If you want a solution competitive with the reference one, your implementation will likely have to implement this dynamic optimization. Please provide your solution in `bfs_hybrid()` in `bfs/bfs.cpp`.

Tips/Hints:

- If you used different representations of the frontier in Parts 1.2 and 1.3, you may have to convert between these representations in the hybrid solution. How might you efficiently convert between them? Is there an overhead in doing so?

You can run our grading script via: `./bfs_grader <path to graphs directory>`, which will report correctness and a performance points score for a number of graphs.

6 Grading and Hand-in

Along with your code, we would like you to hand in a clear but concise high-level description of how your implementation works as well as a brief description of how you arrived at your solutions. Specifically address approaches you tried along the way, and how you went about determining how to optimize your code (For example, what measurements did you perform to guide your optimization efforts?).

Aspects of your work that you should mention in the write-up include:

1. Include your name and Netid at the top of your write-up.
2. Run bfs_grader and insert a copy of the score table in your solutions.
3. Describe the process of optimizing your code:

- In Part 1 (Top Down) and 2 (Bottom Up), where is the synchronization in each of your solutions? Do you do anything to limit the overhead of synchronization?
- In Part 3 (Hybrid), did you decide to switch between the top-down and bottom-up BFS implementations dynamically? How did you decide which implementation to use?
- Why do you think your code (and the staff reference) is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)

7 Points Distribution

The 85 points on this assignment are allotted as follows:

- 70 points: BFS performance
- 15 points: Write-up

8 Hand-in Instructions

Please submit your work using Gradescope.

1. Please submit your writeup as a PDF.
2. To submit your code, run `sh create_submission.h` to generate a `tar.gz` file and submit it to Gradescope. We only look that your `bfs/bfs.cpp` and `bfs/bfs.h` file, so do not change any other files. Before submitting the source files, make sure that all code is compilable and runnable! We should be able to simply make, then execute your programs in the `/bfs` directories without manual intervention.

Our grading scripts will rerun the checker code allowing us to verify your score matches what you submitted in your writeup. We may also run your code on other datasets to further examine its correctness.