Parallel Programming @ NYCU, Fall 2021

This is the webpage for the Parallel Programming course

Programming Assignment III: OpenMP Programming

Parallel Programming by Prof. Yi-Ping You

Due date: **23:59, Nov 18, Thursday, 2021**

The purpose of this assignment is to familiarize yourself with OpenMP programming.

Table of Contents

Get the source code:

```
$ wget https://nycu-sslab.github.io/PP-f21/HW3/HW3.zip
$ unzip HW3.zip -d HW3
$ cd HW3
```

1. Part 1: Parallelizing Conjugate Gradient Method with OpenMP

Conjugate gradient method is an algorithm for the numerical solution of particular systems of linear equations. It is often used to solve partial differential equations, or applied on some optimization problems. You may get more information on Wikipedia.

Please enter the `part1` folder:

```
$ cd part1
```

You can build and run the CG program by:

```
$ make; ./cg
```

In this assignment, you are asked to parallelize a serial implementation of the conjugate gradient method using OpenMP. The serial implementation is in `cg.c` and `cg_impl.c`. Please refer to the `README` file for more information about the code.

In order to parallelize the conjugate gradient method, you may want to use profiling tools to help you explain the performance difference before and after your modification. As it probably just works ineffectively if you simply add a parallel-for directive for each `for` loop, you may want to use profiling tools, such as `perf`, `time`, `gprof`, `Valgrind` (not limited to these tools), to profile your program to better understand how much the improvement is from the code snippets you have modified.

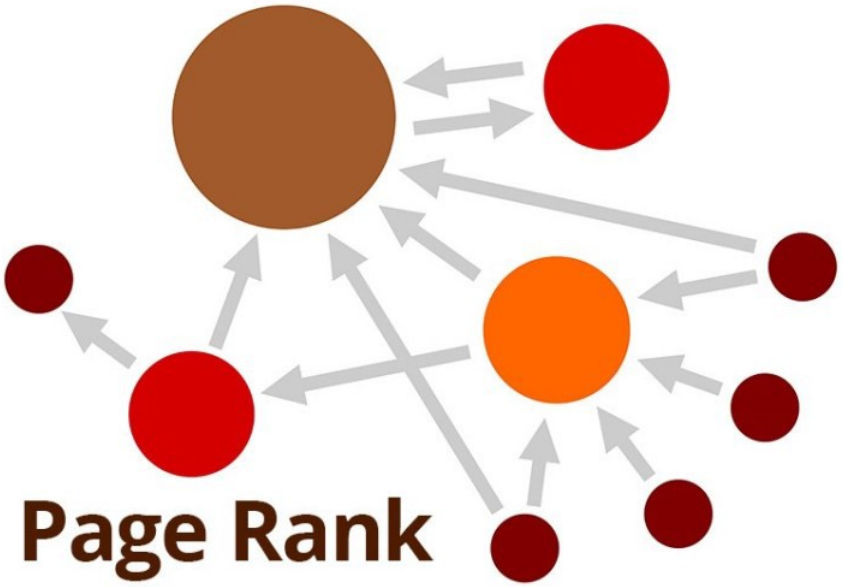
The following instructions may help you finish this part:

- Try to find the hot spots of the program.
- A possible improvement might be from more data parallelism, code refactoring, less page fault, or something else.

See the requirements to finish this part.

You can also run our grader via: `./cg_grader`, which reports the correctness of your program and a performance score.

Part 2: Parallelizing PageRank Algorithm with OpenMP



In this part, you will implement two graph processing algorithms: breadth-first search (BFS) and a simple implementation of page rank. A good implementation of this assignment will be able to run these algorithms on graphs containing hundreds of millions of edges on a multi-core machine in only seconds.

Please enter the `part2` folder:

```
$ cd part2
```

Read the following sections and accomplish the requirements of this part.

2.1 Background: Representing Graphs

The starter code operates on directed graphs, whose implementation you can find in `common/graph.h` and `common/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 `common/graph.h` (and implemented in `common/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);
}
```

2.2 Task 1: Implementing Page Rank

As a simple warm up exercise to get comfortable using the graph data structures, and to get acquainted with a few OpenMP basics, we'd like you to begin by implementing a basic version of the well-known page rank algorithm.

Please take a look at the pseudocode provided to you in the function `pageRank()`, in the file `page_rank/page_rank.cpp`. You should implement the function, parallelizing the code with OpenMP. Just like any other algorithm, first identify independent work and any necessary synchronization.

You can run your code, checking correctness and performance against the staff reference solution using:

```
./pr <PATH_TO_GRAPH_DIRECTORY>/com-orkut_117m.graph
```

If you are working on our workstation machines, we've located a copy of the graph and some other graphs at `/HW3/graphs/`. You can also download the graphs from `http://sslab.cs.nctu.edu.tw/~acliu/all_graphs.tgz`. (But be careful, this is a 3GB download. **Do not replicate the data on the workstations or download the zipped tar file to the workstations for conservation of storage space.**)

Some interesting real-world graphs include:

- com-orkut_117m.graph
- oc-pokec_30m.graph
- rmat_200m.graph
- soc-livejournal1_68m.graph

Some 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.

By default, the `pr` program runs your page rank algorithm with an increasing number of threads (so you can assess speedup trends). However, since runtimes at low core counts can be long, you can explicitly specify the number of threads to only run your code under a single configuration.

```
./pr %GRAPH_FILENAME% 8
```

Your code should handle cases where there are no outgoing edges by distributing the probability mass on such vertices evenly among all the vertices in the graph. That is, your code should work as if there were edges from such a node to every node in the graph (including itself). The comments in the starter code describe how to handle this case.

You can also run our grader via: `./pr_grader <PATH_TO_GRAPH_DIRECTORY>`, which reports the correctness of your program and a performance score for four specific graphs.

You are highly recommended to read the paper, "Direction-Optimizing Breadth-First Search", which proposed a hybrid method combining top-down and bottom-up algorithms for breadth-first search, before you continue the following sections. In Sections 2.3, 2.4, 2.5, you will need to finish a top-down, bottom-up, and hybrid method, respectively.

2.3 Task 2: Parallel Breadth-First Search ("Top Down")

Breadth-first search (BFS) is a common algorithm that you've almost certainly seen in a prior algorithms class.

Please familiarize yourself with the function `bfs_top_down()` in `breadth_first_search/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` (`breadth_first_search/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
```

(as with page rank, `bfs`'s first argument is a graph file, and an optional second argument is the number of threads.)

When you run `bfs`, you'll see the 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 a top-down BFS. As with page rank, 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 part of the computation may need to be synchronized, for example, by wrapping the appropriate code within a critical region using `#pragma omp critical`. However, in this problem you can get by with a single 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.
- 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 `breadth_first_search/bfs.cpp`). In general, these printfs occur infrequently 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.

2.4 Task 3: "Bottom-Up" BFS

Think about what behavior might cause a performance problem in the BFS implementation from Part 2.3. 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 &&
       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 2.3.)

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 `breadth_first_search/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?

2.5 Task 4: 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 `breadth_first_search/bfs.cpp`.

Tips/Hints:

- If you used different representations of the frontier in Parts 2.3 and 2.4, 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?

3. Requirements

3.1 Part 1 Requirements

You will modify only `cg_impl.c` to improve the performance of the CG program, such as inserting OpenMP pragmas to parallelize parts of the program. Of course, you may add any other variables or functions if necessary.

3.2 Part 2 Requirements

You will modify only `bfs.cpp` and `page_rank.cpp` to implement and parallelize the algorithms with OpenMP.

4. Grading Policy

NO CHEATING!! You will receive no credit if you are found cheating.

We will judge your code with the following aspects:

- Correctness: Your parallelized program should run faster than the original (serial) program.
- Scalability: We will evaluate your program with 2, 3, 4 or more threads. Your program is expected to be scalable.
- Performance: We will compare your code with TA's solution. You should get a closer or better performance than TA's implementation.

Total of 110%:

- Part1 (30%): Refer to `cg/grade.c`.
- Part2 (80%): `page_rank` counts for 16% and `breadth_first_search` counts for 64%. Refer to `page_rank/grade.cpp` and `breadth_first_search/grade.cpp`.

5. Evaluation Platform

Your program should be able to run on UNIX-like OS platforms. We will evaluate your programs on the workstations dedicated for this course. You can access these workstations by `ssh` with the following information.

The workstations are based on Ubuntu 20.04 with Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz processors. `g++-10`, `clang++-11`, and `libomp5` have been installed.

IP	Port	User Name	Password
140.113.215.195	37072 ~ 37080	{student_id}	{your_passwd}

Login example:

```
$ ssh <student_id>@140.113.215.195 -p <port>
```

You can run the graders (built from the source code) on the workstation. **We will use the same grader to judge your code**:

```
$ ./cg_grader
$ ./pr_grader  /HW3/graphs/
$ ./bfs_grader  /HW3/graphs/
```

6. Submission

All your files should be organized in the following hierarchy and zipped into a `.zip` file, named `HW3_xxxxxxx.zip`, where `xxxxxxx` is your student ID.

Directory structure inside the zipped file:

- `HW3_xxxxxxx.zip` (root)
  - `cg_impl.c`

- `bfs.cpp`
- `page_rank.cpp`

Zip the files with the following command:

```
$ zip HW3_xxxxxxx.zip cg_impl.c bfs.cpp page_rank.cpp
```

You can use the testing script `test_hw3` to check your answer *for reference only*. Run `test_hw3` in a directory that contains your `HW3_XXXXXXX.zip` file on the workstation. `test_hw3` checks if the zip file is correct, and runs each graders.

> Be aware that `test_hw3` runs **very heavy work** which results in **high CPU usage**, we recommend you *first* use `./cg_grader`, `./pr_grader`, `./bfs_grader` to check your programs at local. **Only use `test_hw3` before your submission.**

```
$ test_hw3
```

You will get *NO POINT* if your ZIP's name is wrong or the ZIP hierarchy is incorrect. (Updated!!)

Be sure to upload your zipped file to new E3 e-Campus system by the due date.

7. References

- OpenMP Official Website
- OpenMP5 SPEC
- OpenMP Tutorial
- Clang11 OpenMP Support
- Linux TIME Manual
- 在 Linux 上使用 Perf 做效能分析(入門篇)
- Wikipedia: Parallel breadth-first search
- 使用 Compare and Swap 做到 Lock Free
- 並行程式設計: Lock-Free Programming
- Atomics 操作