TÉCNICO
LISBOA

Assignment 3

High Performance Computing Systems

Sistemas de Computação de Elevado Desempenho (SCED)

2020/2021

Overview

In this assignment, you will implement two graph processing algorithms: breadth-first search[1] (BFS) and a simple implementation of page rank[2]. 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.

Deadlines and Submission

You will submit the work developed for Lab3 via Fenix (until 29th of November at 23h59, i.e., Sunday).

The submission should be made in a single `zip` file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for all programs (to keep submission small, please do a `make clean` in program directories prior to creating the archive, and remove any residual output, etc.)

When handed in, all codes must be compilable and runnable out of the box on the `adriana` machine!

Your report should have at most 10 pages and should be well structured. For each part (problem) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

Environment Setup

For this assignment, you will need to run (the final version of) your code on `adriana` machine (hostname: `adriana.inesc-id.pt`). This machine contains four-core 3.5 GHz Intel Core i7 processor (although dynamic frequency scaling can take it to 3.9 GHz when the chip decides it is useful and possible to do so). For the curious, a complete specification for this CPU can be found at https://ark.intel.com/.../intel-core-i7-3770k-processor-8m-cache-up-to-3-90-ghz.html.[3]

The access to `adriana` machine is performed through `ssh`, by using the command:

```
ssh HPCS_g<group_number>@adriana.inesc-id.pt
```

where you should substitute *`<group_number>`* with your group number (e.g., the username for the group number 3 is `HPCS_g3`). The passwords for each group have the flowing format `g<group_number>hpcs` (again, the password for the group number 3 is `g3hpcs`).

IMPORTANT: Since the machine is a shared resource, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups (or even other researchers from INESC-ID). To check the users logged in the machine, you can use the **`who`**

[1] https://en.wikipedia.org/wiki/Breadth-first_search
[2] https://en.wikipedia.org/wiki/PageRank
[3] https://ark.intel.com/content/www/us/en/ark/products/65523/intel-core-i7-3770k-processor-8m-cache-up-to-3-90-ghz.html

command. It is highly recommended to schedule the utilization of adriana among yourselves to avoid the simultaneous use of the machine.

Note: For grading purposes, we expect you to report on the performance of code run on the adriana machine. However, for development, you may also want to run the programs in this assignment on your own machine.

To get started:

1. The assignment starter code is available on the course webpage (section: Labs). wget it, unzip it and start having fun with it.

Background: Learning OpenMP

In this assignment we'd like you to use OpenMP[4] 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. Besides on our slides from theoretical classes, OpenMP is well documented online and 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.

Background: Representing Graphs

The starter code operates on directed graphs, whose implementation you can find in the common folder, files 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);
}
```

[4] http://openmp.org/wp/

Part 1: Warm-Up: 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[5] algorithm.

Please take a look at the pseudocode provided to you in the function `pageRank()`, in the file `pagerank/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_GRAPHS_DIRECTORY>/com-orkut_117m.graph
```

If you are working on `adriana` machine, we've located a copy of the graphs directory at **`/extra/sced_lab3`**. Some interesting real-world graphs include:

- `com-orkut_117m.graph`
- `soc-pokec_30m.graph`
- `rmat_200m.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 (such as tiny, grid4x4 etc.)

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 grading script via: `./pr_grader <PATH_TO_GRAPHS_DIRECTORY>` which will report correctness and a performance points score for a number of graphs.

[5] https://en.wikipedia.org/wiki/PageRank

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

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

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 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. 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[6], 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 `bfs/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.

[6] http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

Part 3: "Bottom-up" BFS

Think about what behavior might cause a performance problem in the BFS implementation from Part 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 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?

Part 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 `bfs/bfs.cpp`.

Tips/Hints:

- If you used different representations of the frontier in Parts 2 and 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?

Important aspects of your work that you should mention in the report:

For bfs, describe the process of optimizing your code:

- In Part 2 (Top Down) and 3 (Bottom Up), where is the synchronization in each of your solutions? Do you do anything to limit the overhead of synchronization?
- In Part 4 (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 is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)