Assignment 3: Processing Big Graphs on the Xeon Phi

Due: Wed Mar 8, 11:59PM EST

110 points total

Overview

In this assignment, which will be released in two parts, you will implement several parallel graph processing algorithms on the Xeon Phi processor (a 68-core, 256 thread) chip. (**However, our special access to a cluster of 100 Phi processors will not be available until next week, so you will begin your development and testing on the eight-core Xeon machines in GHC.**) A good implementation of this assignment will be able to run algorithms like breadth-first search on graphs containing hundreds of millions of edges in seconds. Once you have a good single node implementation that runs efficiently on multiple cores, in the second part of the assignment you will extend the implementation to distribute and process the graph on multiple machines. This will allow you to process graphs that don't fit in memory on a single machine.

Environment Setup

Early starters of this assignment should get started by running on the 8-core machines in the GHC cluster. (`ghc26/27/30-37/39-42/44/45`) These machines will suffice for basic development and performance testing. **However, final grading will be done on a cluster of Xeon Phi machines managed by Intel.** (Intel has generously allowed us to use this cluster, which contains 100 Phis.) We recommend that your initial development and testing be done on GHC, but once you are reading to test performance, please check out **How to Use the Intel Cluster**.

To get started:

Download the Assignment 3 starter code from the course Github page using:

```
git clone https://github.com/cmu15418/assignment3
```

Part 1: Parallel Graph Algorithms on a Multi-Core CPU

In this part of the assignment, you will implement two graph processing algorithms: **breadth-first search** (BFS) and a simple implementation of **page rank**.

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. It is well documented online, but here is a brief example of parallelizing a `for` loop, with mutual exclusion

You can also use OpenMP to tell the compiler to parallelize iterations of `for` loops, and to manage mutual exclusion.

```
/* The iterations this for loop may be parallelized */
#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 (e.g., `omp parallel for schedule(dynamic)`).

Here is an example for an atomic counter update.

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

```
}
```

As a participant in a 400-level course, 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.

Background: Representing Graphs

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

Part 1.1: Warm up: Implementing Page Rank (8 points)

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 `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 sychronization.

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

```
./pr /afs/cs/academic/class/15418-s17/public/asst3_graphs/com-orkut_117m.graph
```

You'll find a number of graphs in the course directory. Note that since some of these graphs are quite large (more than a GB in size), so during debugging/testing you might find it useful to locate them on the local disk in `/usr/tmp/`. Some interesting graphs include:

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

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% 16
```

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). Please pull the latest version of the starter code, and see the added comment.

Running code on the Intel Cluster

While its possible to debug and test your page rank code on GHC machines, final performance testing of code in this assignment will be done on the 68-core Xeon Phi processors (KNL). Please see our notes on **How to Use the Intel Cluster**.

Once you're up and running on the cluster, you can submit pagerank jobs to the KNL by running: `./submit`.

Note that from an interactive shell (e.g., GHC), you can also run the grading script via: `./pr_grader`.

Part 1.2: 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 `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 /afs/cs/academic/class/15418-s17/public/asst3_graphs/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 (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**, 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. (We will talk about compare and swap in detail in the second half of the course, but in this problem it's up to you to figure out how to use it.)

- 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.

Part 1.3: "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?

Part 1.4: Hybrid BFS (42 points)

Notice that in some steps of the BFS, the "bottom up" BFS is signficantly faster than the top-down version. In other steps, the top-down version is signficantly 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?

Running code on the Intel Cluster

While its possible to debug and test your page rank code on GHC machines, final performance testing of code in this assignment will be done on the 68-core Xeon Phi processors (KNL). Please see our notes on **How to Use the Intel Cluster**.

Once you're up and running on the cluster, you can submit pagerank jobs to the KNL by running: `./submit`.

Note that from an interactive shell (e.g., GHC), you can also run the grading script via: `./bfs_grader`.

Part 2: Scaling to Multiple Nodes (45 points)

In Part 1 of the assignment you used OpenMP to parallelize your graph processing code onto many cores in a single machine. Now in part 2 you will distribute your solution across a small cluster of machines. To communicate between these two machines, your code will send and receive messages using the **MPI** (Message Passing Interface) library. The starter code for part 2 of the assignment can be found in `/part2/pagerank` and `/part2/bfs`.

Part 2.1: Distributed PageRank (20 performance points)

In this part of the assignment you will extend your page rank implementation for multiple nodes. Take at look at function `pageRank()` in `part2/pagerank/pagerank.cpp`. This function is the same as in part 1, but now the representation of the graph has changed.

The graph is distributed, meaning that each node in the cluster is the "owner" of a subset of the graph's vertices (all the vertices assigned to a node have consecutive vertex ids in the the overall graph). You can think of this part of the assignment as mimicking a situation where you wish to process a graph that is so big that it cannot fit in memory on one node, so you are using many nodes to store the graph in the aggregate memory of the nodes of the cluster.

Take some time to understand the `DistGraph` structure defined in `part2/include/graph_dist.h`. After distributed graph construction, each node in the cluster is the owner of graph vertices in the range from `start_vertex` to `end_vertex`. The arrays `out_edges` and `in_edges` hold the vertex ids of outgoing and incoming edges local to the node.

- Although you will never call the function yourself, you may be interested in the implementation of `DistGraph::get_incoming_edges()`. In this method, all nodes exchange messages to initialize the `in_edges` array from the contents of `out_edges`, which are original spread across the cluster.
- You are welcome to modify the `DistGraph` structure to meet your needs.
- There is a method `DistGraph::setup()` that is a hook for you to place setup/initialization code that prepares the structure for future computation. For example, some implementations might optionally convert the `in_edges` and `out_edges` structures currently in `DistGraph` to a format more suitable for faster processing (e.g. closer to the graph format from Part 1.)

Once you understand the distributed graph format, your job is to implement `pageRank()` in `part2/pagerank/pagerank.cpp`. The complex part of the implementation is ensuring that the necessary per-vertex score updates are communicated between the nodes in each iteration of the algorithm. To be performant, your implementation should take care to communicate only the necessary information. (Hint: why is it not necessary to communicate each partition of the graph to all other nodes?)

How to run distributed Page rank on the Intel Cluster:

Please see our notes on **How to Use the Intel Cluster**.

On the Intel cluster `./submit` will run your current implementation on 16 KNL nodes via the job queue. This 16-node configuration is the configuration we will use for grading.

How to run MPI Programs on the GHC machines:

Development and initial testing can also be done on the GHC machines to avoid the Intel cluster job queue. In order to add support for MPI compilation to the GHC machines, add the following to your `~/.cshrc`:

```
setenv PATH $PATH\:/usr/lib64/openmpi/bin
```

or to your `~/.bashrc`:

```
export PATH=$PATH:/usr/lib64/openmpi/bin
```

On ghc machines `mpirun -np <num processes> ./pr_dist [options]` will run `pr_dist` on a graph distributed across `<num processes>`.

Part 2.2: Distributed BFS (25 points)

In this part of the assignment, you will create a distributed implementation of BFS. Take a look at the function `bfs()` in `part2/bfs/bfs.cpp`. Distributed BFS uses the same `DistGraph` structure as distributed page rank, so after initialization, each node of the cluster is responsible for different graph vertices. This makes BFS quite challenging to implement, since as the BFS evolves the frontier might move from node to node or might contain a set of vertices that a distributed across nodes. For example, consider processing vertex `v_i` maintained on node 0 during a BFS step. This action might add neighbor vertex `v_j` to the frontier, but `v_j` may be located on node 10. This means that in the next BFS step, node 10 should be processing `v_j`. You will need to determine how to orchestrate this communication after each BFS step.

You are allowed to implement BFS however you wish, but we want you to carry out processing related to each vertex on the node owning the vertex. In a top-down BFS implementation, this means that if vertex `v_i` is owned by node 0, then the work to traverse `v_i` and adding its neighbors to the frontier should be done on node 0. (E.g., you are not permitted to copy the entire graph to one node, perform your part 1 BFS, and then copy the results back to all nodes.)

Below are a few comments on the structure of the start code. However, we remind you that your solution does not need to follow this structure:

- We provide you an implementation of a distributed vertex list structure (`DistFrontier` in `frontier_dist.h`). The code should be relatively self-explanatory, but the key point is that `DistFrontier` organizes the list of vertices on the frontier by their owning node. Since BFS terminates when the frontier is empty, you will need to implement the method `DistFrontier::is_empty()`. (It currently returns `true`.)
- In `part2/bfs/bfs.cpp`, there are two functions left for you to implement: The first is `bfs_step()`, which as you'd expect carries out a step of a BFS using the distributed graph and distributed frontier structures. The second is `global_frontier_sync()` which is called after each step of the BFS to propogate new frontier information across cluster nodes.

How to run Distributed BFS on the Intel Cluster:

On the Intel cluster `./submit` will run your current implementation on 16 KNL nodes via the job queue. This 16-node configuration is the configuration we will use for correctness grading.

How to run Distributed BFS on GHC:

On ghc machines `mpirun -np <num processes> ./bfs_dist [options]` will run `bfs` on a graph distributed across `<num processes>`

Grading:

As you will quickly realize, implementing graph algorithms like BFS efficiently in a distributed memory environment is significantly tricker than in shared memory. In fact, for the graphs available in this class, a fast parallel single node shared memory implementation of BFS will be hard to beat in the distributed setting, since there is high overhead to communicating between machines. (However, one motivation for a distributed implementation is processing large graphs that don't fit on a single machine.) As such Part 2.2. is graded on your ability to correctly implement a distributed BFS on several test graphs, rather than quantitative performance of solution against a reference. (However we reserve the right to dock a few points if a solution is grossy inefficient.)

Grading and Handin

Along with your code, we would like you to hand in a clear, 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 both partners names and andrew id's at the top of your write-up.
2. Replicate the part 1 BFS/pagerank, and part 2 pagerank score table generated for your solution.
3. For part 1-bfs, describe the process of optimizing your code:
    - Where is the synchronization in each your solutions? Do you do anything to limit the overhead of synchronization?
    - Did you decide to dynamically switch between the top-down and bottom-up BFS implementations? 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 imabalance? communication/synchronization? data movement?)
4. For part 2-bfs, describe how you implemented the distributed bfs.
5. Briefly describe how you arrived at your final solution. What other approaches did you try along the way. What was wrong with them?

Points Distribution

The 110 points on this assignment are alloted as follows:

- Part 1:
    - 8 points: pagerank performance
    - 42 points: BFS performance
- Part 2:
    - 20 points: Pagerank performance

    - 25 points: BFS

- Writeup: 15 points

Hand-in Instructions

Please submit your work using Autolab.

1. **Please submit your writeup as the file `writeup.pdf`.**
2. **Please submit your code under the folder `code`.** Just submit your full assignment 3 source tree. To keep submission sizes small, please do a `make clean` in the program directories prior to creating the archive, and remove any residual output images, etc. 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 `/bfs`, `/pagerank`, `/part2/pagerank/`, and `part2/bfs` directories without manual intervention.
3. **If you're working with a partner, form a group on Autolab.** Do this before submitting your assignment. One submission per group is sufficient.

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