Marionet Summer School
GraphGrind Lab

Hans Vandierendonck

July 20th

This document guides you through some experiments and programming exercises that will make you acquainted with GraphGrind.

You will need graph datasets. These have been copied to your machine in the directory `/scratch/graphs`. You need to unzip these files before the can be used:

```
$ gunzip /scratch/graphs/LiveJournal_dir.gz &
$ gunzip /scratch/graphs/orkut_undir.gz &
```

This will take a short while. Best to run them in the background while you setup GraphGrind. The LiveJournal graph is smaller than the Orkut graph. The former is a directed graph, while Orkut is undirected. Whenever you use the Orkut graph, make sure to use the "`-s`" option, otherwise you will run out of memory!

GraphGrind is available from `https://github.com/hvdieren/GraphGrind-ICPP`. Download it using git:

```
$ git clone https://github.com/hvdieren/GraphGrind-ICPP graphgrind
```

In the following we will assume that the top-level directory of the repository was named `graphgrind`. To undertake this lab, move to the directory `graphgrind/tutorial`.

1 Compiling GraphGrind Programs

The GraphGrind framework is implemented in C++ header files only. Algorithms are implemented in C++ source code files with extension `.C`. GraphGrind uses Cilkplus for managing parallel execution. As such, you need a Cilk-enabled C++ compiler. The Intel C compiler and GCC both support Cilk. To use GCC on the lab machines, execute this command:

```
$ scl enable devtoolset-6 bash
```

To compile, step into the directory `graphgrind/tutorial` and follow these steps (these commands are assuming a bash shell):

```
$ make PageRank
```

This will build the PageRank program under `graphgrind/tutorial/PageRank`. A similar approach may be taken for other algorithms.

You can now run the program with a command line of this structure:

```
$ CILK_NWORKERS=1 ./PageRank [options] /path/to/graph
```

The environment variable `CILK_NWORKERS` controls the number of threads used to execute the program. By default, as many threads are started as there are processing cores on your system. This is 4 for the lab machines.

The available command line options are:

Option	Description	Default
-c *n*	Partition the graph *n*-ways	384
-v (vertex—edge)	Partition the graph by balancing the number of vertices or edges	edge
-rounds *n*	Average execution time of the algorithm over *n* runs	3
-r *vid*	Start BFS search at vertex *vid*	0
-s	Optimise space using the symmetry of the graph (undirected)	0

2 Exploration

1. **Parallelism:** Compile and run the `PageRank` program using 1 core:

   ```
   $ CILK_NWORKERS=1 ./PageRank -rounds 5 -c 1 /path/to/LiveJournal_dir
   ```

   Then execute it using as many cores as your computer has:

   ```
   $ CILK_NWORKERS=4 ./PageRank -rounds 5 -c 4 /path/to/LiveJournal_dir
   ```

   Note that it is important to adjust the value of the `-c` to the same as the number of threads.

   What you should see is that the parallel execution is faster than the single-core execution, although it won't be as much faster as the number of cores in your computer.

2. **Graph partitions:** The parameter `-c` controls how many partitions are created. Its primary function is to control memory locality. By default the code is compiled such that each partition is executed by a single thread. As such, with fewer partitions than threads, some of the threads will be underutilised.

   Run the `PageRank` program with 1, 2, 4, 8, 16 and 32 partitions and observe how the execution time changes.

   GraphGrind prints the percentage of vertices (destinations) in each partition and the percentage of edges. Check how the partitions are balanced when you change the number of partitions.

3. **Vertex vs. edge orientation:** PageRank is an edge-oriented algorithm. By default, GraphGrind balances the number of edges during partitioning. Add the parameter `-v vertex` to the command line and observe the impact on the composition of the partitions. How does the performance impact of the partitioning method vary with the number of partitions created?

3 Breadth-First Search

1. In the directory `graphgrind/tutorial`, review the source code in `BFS.C`. GraphGrind uses three versions of the update method in the class `BFS_F`: one for single-threaded execution, one thread-safe version using atomic updates, and a third version that enables the compiler to retain values in registers during a CSC traversal. Satisfy yourself that they implement the same functionality.

2. Compile the program and execute it using as many threads as available on your computer (4 threads on the lab machines).

   ```
   % CILK_NWORKERS=4 ./BFS -rounds 10 -c 8 -r 100 /path/to/LiveJournal_dir
   ```

3. By default, graphgrind applies edge partitioning. Change the partitioning method to vertex-balanced partitioning:

   ```
   % CILK_NWORKERS=4 ./BFS -rounds 10 -c 8 -r 100 -v vertex /path/to/LiveJournal_dir
   ```

   and observe the performance impact. Can you explain this result?

4. In BFS, vertices converge quickly. In fact, the parent of each vertex is updated at most once (no updates are made for unreachable vertices). Mute the convergence check (which is responsible for skipping edges during a pull traversal) by making it always return true. Recompile the program and run it again. How is the overall performance affected?

5. The theory states that vertex-balanced partitioning is effective when when most edges are skipped during `EdgeMap`. We know this is the case for BFS because of Beamer's SC'12 paper on direction-optimization.

   Mute the convergence check (which is responsible for skipping edges during a pull traversal) by making it always return true. Recompile the program and run it again with edge-balanced and vertex-balanced partitioning and compare their performance. Which one is most efficient now? How was the overall performance affected?

4 Connected Components

1. The file `graphgrind/tutorial/Components.C` is a skeleton program for the connected components problem. Study the presented code. For convenience, only one version of the update method needs to be implemented (the non-atomic and cached versions point to the atomic one for convenience).

2. Complete the methods `update(intT s, intT d)` and `cond(intT d)` to implement the connected components problem. The update method should implement `IDs[d] = min(IDs[d], prevIDs[s])` using the atomic operation `writeMin(intT *ptr, intT val)`.

   The method `writeMin` is defined in `graphgrind/utils.h`. It returns true if it could update the memory location pointed to by `ptr` with the value of `b`. It returns false when `*ptr` was already less than `b`, or another thread updated `*ptr` concurrently.

   What should the `cond(intT d)` method return to indicate convergence?

   Compile the program and run it on the orkut graph (the algorithm is applicable only to directed graphs):

   ```
   $ make Components.C
   $ CILK_NWORKERS=4 ./Components -rounds 10 -c 4 -s /path/to/orkut_undir
   ```

   The program should report 187 distinct components. If not, there will be an error in the program logic. Record the execution time.

3. Edit the file `graphgrind/tutorial/Components.C` again and now make the following optimization: in the update method, allow to propagate the current label to other vertices within the same round.

   After making this change, recompile the program and check correctness. Does this speed up the calculation significantly? Can you argue why it is correct?

4. Edit the file `graphgrind/tutorial/Components.C` once more and implement the versions of the update method for single-threaded execution and the version with the caching optimisation. Set the variable `use_cache` to true to enable caching. Validate the correctness. Can you observe a performance difference?