Project: Map Reduce and Hadoop
Unity Id : akagrawa

Problem 1: The goal of this assignment is to make sure you know how to use debugger and profiler tools. You are asked to write a C/C++ code for calculating the degree of each vertex in the graph G with *n* vertices and *m* edges. It will be a serial program but it must mimic the map/reduce API. The graphs will be provided to you for testing. The graphs will be stored in ASCII files: one file per graph. The file format is the following: (a) the first line stores *n* and *m*, the number of vertices and edges, resp.; (b) every other line is for an edge defined by its vertex id pair; (c) each vertex id is an integer between 0 and *n*-1.

Code: Please refer README.md for the code description and commands for the execution.

Approach: Implemented simple Mapper and reducer function in order to mimic map-reduce framework. The mapper function will read all the files from the input folder and produce an intermediate file with <Key,Value> pair which in this case will be vertex as key and value as 1. This intermediate file is then fed input to the reducer function which will combine these key-value pairs using internal map(std::map) function. The final reduced output is then tored in the output file and the intermediate file is deleted.

Input Files: The graph data-set given for amazon. dblp and youtube is used for the performance evaluation. All the graph sizes (small, medium, large, original) are included for the processing and performance.

Output Metrics: For all the input graphs the program written in C++ generates the following output metrics.

```
######## Map Reduce Job Begins ############
 ------ Mapper Job Initiated --------
Mapper job done for file = : youtube.graph.small
Mapper job done for file = : youtube.graph.medium
Mapper job done for file = : dblp.graph.small
Mapper job done for file = : youtube.graph.original
Mapper job done for file = : amazon.graph.original
Mapper job done for file = : dblp.graph.original
Mapper job done for file = : dblp.graph.large
Mapper job done for file = : amazon.graph.medium
Mapper job done for file = : youtube.graph.large
Mapper job done for file = : amazon.graph.large
Mapper job done for file = : dblp.graph.medium
Mapper job done for file = : amazon.graph.small
```

```
------ Mapper Job Ends ------
For Mapper Jobs 37000000 clicks =(37.000000 seconds).

------- Reducer Job Initiated  -------
For Reducer Jobs 21660000 clicks =(21.660000 seconds).

For Total Map-Reduce Jobs 58660000 clicks = (58.660000 seconds).
######## Map Reduce Ends ##########
```

(1) *Demonstrate that you know how to use the profiler:*
Profile the performance of your program using the GNU *gprof* profiler.
(a) Using small, medium, large size graphs provided to you, write a short summary from the profiler. Specifically comment on the cost of performing the I/O versus computation. Which functions take most of the computational time? Are there any functions that take longer (e.g., say 80% of the total time)?

GNU gprof profile is used for profiling the above process. The steps are precisely noted in the README.md file. All the input graphs of small, medium, large and original sizes are taken for computation and hence observed through profilers.
In short, the code is compiled using -pg command and hence while executing it gmon.out file is generated. This gmon.out file is then used by various profilers to produce different set of outputs.

1) **gprof a.out gmon.out -p >> performance.txt**

This provides a flat profile indicating many metrics described below by each functions.

Flat profile:

Each sample counts as 0.01 seconds. Sample output is given below. Please refer the complete flat profile inside files folder.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
88.72	5.74	5.74	1	5.74	5.84	VertexDegree::**reducer**(char**)
9.74	6.37	0.63	1	0.63	0.63	VertexDegree::**mapper**(char**)
1.08	6.44	0.07	1134890	0.00	0.00	frame_dummy
0.46	6.47	0.03	1	0.03	0.03	std::_Rb_tree<std::string, std::pair<std::string const, int>, std::_Select1st<std::pair<std::string const, int> >, std::less<std::string>,

std::allocator<std::pair<std::stringconst,int>>>::_M_erase(std::_Rb_tree_node<std::pair<st
d::string const, int> >*)

0.00 6.47 0.00 1 0.00 0.00

_GLOBAL__sub_I__ZN12VertexDegree6mapperEPPc

%time the percentage of the total running time of the program used by this function.

%cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this

ms/call function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this

ms/call function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

2) gprof a.out gmon.out -q >> performance.txt

Above command outputs the call graph describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children. Sample output is given below. Please refer the complete call graph inside files folder.

granularity: each sample hit covers 4 byte(s) for 0.15% of 6.47 seconds

index % time self children called name spontaneous>

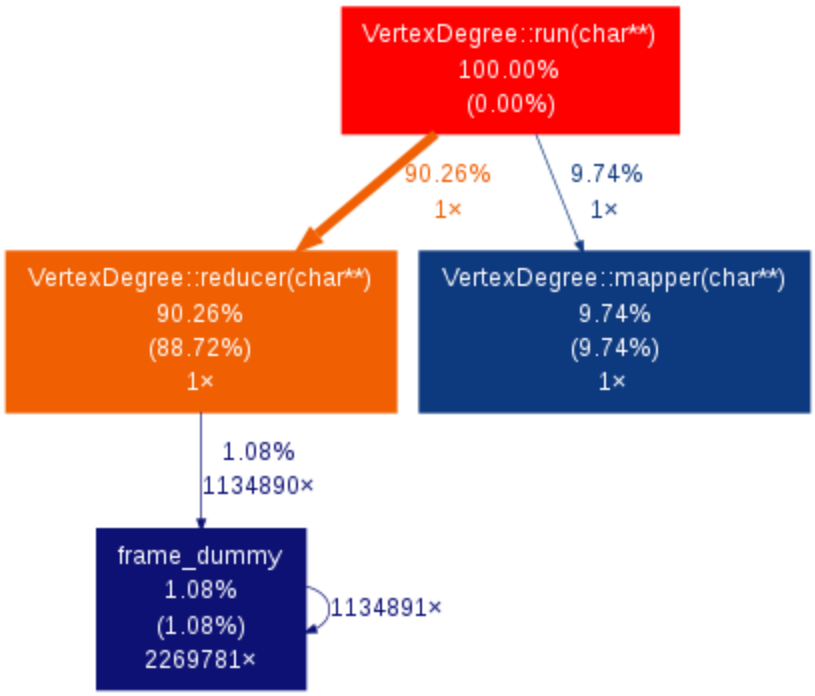
[1] 100.0 0.00 6.47 VertexDegree::run(char**) [1]
5.74 0.10 1/1 VertexDegree::reducer(char**) [2]
0.63 0.00 1/1 VertexDegree::mapper(char**) [3]

5.74 0.10 1/1 VertexDegree::run(char**) [1]
[2] 90.3 5.74 0.10 1 VertexDegree::reducer(char**) [2]
0.07 0.00 1134890/1134890 frame_dummy [4]
0.03 0.00 1/1 std::_Rb_tree<std::string, std::pair<std::string const, int>, std::_Select1st<std::pair<std::string const, int> >, std::less<std::string>, std::allocator<std::pair<std::string const, int> > >::_M_erase(std::_Rb_tree_node<std::pair<std::string const, int> >*) [5]

```
                0.63    0.00       1/1          VertexDegree::run(char**) [1]
[3]     9.7     0.63    0.00       1          VertexDegree::mapper(char**) [3]
-----------------------------------------------
                                1134891                    frame_dummy [4]
                0.07    0.00 1134890/1134890     VertexDegree::reducer(char**) [2]
[4]     1.1     0.07    0.00 1134890+1134891   frame_dummy [4]
                                1134891                    frame_dummy [4]
-----------------------------------------------
```

3) Conversion of profiling output to a dot graph : Gprof2Dot

Usage : gprof path/to/your/executable | gprof2dot.py | dot -Tpng -o output.png



Conclusion and Inference : With the observations seen above the reducer is consuming more time (more than 80%) as compare to other functions call. In the above computation, the I/O call is done inside the mapper and reducer function. Since in the reducer function, the file is read line by line and corresponding key,value are maintained in maps. Hence it is taking more time than mapper function. Clearly **I/O access times** is dominated as compare to computation time since every unit operation in mapper-reducer call is performing file read-write operation while the computation is mostly wrt to hashmaps which are cheaply accessed in memory.

References:

- **https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html**
- **https://sourceware.org/binutils/docs/gprof**
- **http://stackoverflow.com/questions/874134/find-if-string-endswith-another-string-in-c?lq=1**
- https://code.google.com/p/jrfonseca/wiki/Gprof2Dot