CMPE478 – Parallel Processing
Assignment 1 Report
A Parallel Version of Google Ranking Process Using OpenMP

CEMAL BURAK AYGÜN (2014400072) **SPRING 2019**

Implementation Details

First of all, I calculated the number of nodes (vertices) in the graph using the below *bash* command and hard-coded the result (**1,850,065**) as a *#define* statement:

```
cat graph.txt | tr '\t' '\n' | sort | uniq | wc -l
```

When I inspected Erdös Web Graph file (**graph.txt**), I observed that the *<source, destination>* node pairs (lines) are grouped together with respect to the *destination* nodes. In **P** matrix, each row (and column) corresponds to a node/host and the columns of non-zero entries of a row (say, *node1*) specify the nodes that have an edge going into *node1*. If we assign the rows (indices) of **P** matrix to the destination nodes in **graph.txt** file in the line order, we see that the file stores information in row-wise order. This structure of the file makes it easy to construct *values* and *col_indices* arrays of CSR format because those arrays store entries of **P** matrix in row-wise order, too.

I needed to traverse the file twice: Once for assigning (row) indices to the nodes and once for constructing the related CSR format arrays. Since traversing the file is time consuming, I store the node pairs (lines) of the file in a vector of *<string, string>* pairs.

While traversing the graph (file) the first time, a **pair** object is created from a line and inserted into a **vector** named **graphVector**. Also, the next available (row) index is assigned to the destination node of that pair. Node-Index assignments are stored in a **map** object named **nodeIndexMap**.

All non-zero entries in a column of **P** matrix is equal to *1/<out degree of the node corresponding to that column>*. Out degree values of each node are stored in an array named **outDegrees** (initialized as 0). The program traverses the graph again; this time, through **graphVector**. For each *<source, destination>* node pair, the next available (row) index is assigned to the *source* node (if it has not been encountered before) and out degree of the *source* node is incremented by 1. Also, for each node pair (line) *groups*, current size of **columnIndices** vector (which stores the column indices of the corresponding entries in **values** vector) is appended to **rowBegin** vector and the indices of source nodes of that group are appended to **columnIndices**. After the iterations are done, **values** vector is constructed such that i^{th} element of **values** is equal to *1/<out degree of the node with the index given by i^{th} element of **columnIndices**>*.

After **P** matrix is constructed in CSR format, the program continues to perform the tests in which rankings of each host (node) are calculated under different parallelization parameters. **rt1** ($r^{(t)}$ vector) array is initialized as [1, 1, …] and the elements of **rt2** ($r^{(t+1)}$ vector) array are calculated by $r^{(t+1)} = \alpha * P * r^{(t)} + (1-\alpha) * c$ (α=0.2 and c = [1, 1, …]) until the *difference* between the 2 vectors converges; $\Sigma|r_i^{(t+1)} - r_i^{(t)}| \le \varepsilon$ ($\varepsilon = 10^{-6}$). This calculation is implemented using a **for-loop** inside another for-loop. Basically, the *outer* loop traverses the rows of **P** matrix (stored in CSR format) and the *inner* loop traverses the columns of each row. Each iteration of the *outer* loop calculates one element of **rt2** and a partial difference. The calculation part (only the outer loop) of the program is parallelized via **OpenMP**. An integer named **difference** is used as the *reduction* variable with *summation*. The serial part of the program maintains a **while-loop**. Inside that loop, the parallel region calculates **rt2** and **difference**. Then, the serial part checks **difference** and breaks the while-loop if it is smaller than some threshold (ε).

Finally, the program finds and prints the 5 highest ranked hosts. The search occurs like this: **rt2** (which contains the final ranking values of the hosts) is traversed to find the *index* of the highest element. Then, this element is replaced with **-1** in **rt2** to eliminate this host from the next search. Then, **nodeIndexMap** (which maps host IDs to indices) is traversed to find the ID of the host with the found *index*. This process is repeated 5 times. Ranking results are as follows:

1: 4mekp13kca78a3hfsrb0k813n9
2: 0491md82hej8u15vi98isrmuih
3: 3165mii1s1g0invqs94q303v0v
4: 46o3c5beh6kiojkvr1tvsk4ptt
5: 2494c7mt12frm3c3go86abe13h

Details of the Machine/CPU I Used

I tested my program on a desktop computer on *Linux Mint 19.1 Cinnamon (Linux Kernel: 4.15.0-46-generic)*.

I compiled the program via *g++* (version 7.3.0) and *C++11*:
g++ -std=c++11 main.cpp -fopenmp

Output of **lscpu** command is as follows:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	2
Core(s) per socket:	2
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	94
Model name:	Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz

Stepping:	3
CPU MHz:	800.129
CPU max MHz:	3700,0000
CPU min MHz:	800,0000
BogoMIPS:	7392.00
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	3072K
NUMA node0 CPU(s):	0-3

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d

```
cba@cba-desktop:~/Desktop/Code$ g++ --version
g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

cba@cba-desktop:~/Desktop/Code$ ls
graph.txt  main.cpp
cba@cba-desktop:~/Desktop/Code$ g++ -std=c++11 main.cpp -fopenmp
cba@cba-desktop:~/Desktop/Code$ ./a.out graph.txt
Parsing graph.txt        ... DONE
Constructing CSR-format related arrays  ... DONE

Starting Tests:
        [TEST 1/9] Scheduling Method: STATIC    Chunk Size: 1000        ... DONE
        [TEST 2/9] Scheduling Method: STATIC    Chunk Size: 3000        ... DONE
        [TEST 3/9] Scheduling Method: STATIC    Chunk Size: 5000        ... DONE
        [TEST 4/9] Scheduling Method: DYNAMIC   Chunk Size: 1000        ... DONE
        [TEST 5/9] Scheduling Method: DYNAMIC   Chunk Size: 3000        ... DONE
        [TEST 6/9] Scheduling Method: DYNAMIC   Chunk Size: 5000        ... DONE
        [TEST 7/9] Scheduling Method: GUIDED    Chunk Size: 1000        ... DONE
        [TEST 8/9] Scheduling Method: GUIDED    Chunk Size: 3000        ... DONE
        [TEST 9/9] Scheduling Method: GUIDED    Chunk Size: 5000        ... DONE


Test results were saved in results.csv.

5 Most Highest Ranked Hosts (IDs):
1: 4mekp13kca78a3hfsrb0k813n9
2: 0491md82hej8u15vi98isrmuih
3: 3165mii1s1g0invqs94q303v0v
4: 46o3c5beh6kiojkvr1tvsk4ptt
5: 2494c7mt12frm3c3go86abe13h
cba@cba-desktop:~/Desktop/Code$ ls
a.out  graph.txt  main.cpp  results.csv
```

CMPE478 – Parallel Processing
Assignment 1 Report
A Parallel Version of Google Ranking Process Using OpenMP

CEMAL BURAK AYGÜN (2014400072) **SPRING 2019**

Timings Table (Generated CSV File)

The timing values are the duration (in seconds) of ranking calculation only. It does not contain the duration of parsing graph file and finding 5 highest ranked hosts.

Test No.	Scheduling Method	Chunk Size	No. of Iterations	1 Thread(s)	2 Thread(s)	3 Thread(s)	4 Thread(s)	5 Thread(s)	6 Thread(s)	7 Thread(s)	8 Thread(s)
1	STATIC	1000	14	6.91657	3.45623	3.29699	2.81902	3.06657	3.11822	3.07514	2.92303
2	STATIC	3000	14	6.87924	3.59565	3.42763	2.8074	3.00122	2.8937	2.84887	2.86944
3	STATIC	5000	14	6.87854	3.49436	3.63862	3.11349	3.08564	2.85515	3.04547	2.92646
4	DYNAMIC	1000	14	6.8884	3.74073	3.29896	2.96845	2.94139	3.01887	3.02411	2.89057
5	DYNAMIC	3000	14	6.74564	3.43397	3.10135	2.76424	2.76524	2.78874	2.77026	2.81294
6	DYNAMIC	5000	14	7.43208	3.71588	3.09492	2.78703	2.94658	2.89341	2.8083	2.76923
7	GUIDED	1000	14	6.75017	3.91398	3.42369	2.80251	3.05022	3.03848	3.04954	2.92987
8	GUIDED	3000	14	6.9321	3.64461	3.55004	2.99871	2.91162	3.08007	3.07145	2.90331
9	GUIDED	5000	14	6.70258	3.48548	3.41129	2.8538	2.85741	3.0804	3.20852	3.16207

Discussion of the Results

As it can be seen in the timings table above, chunk size does not have a significant impact on the results when the scheduling method is STATIC or GUIDED. However, when the scheduling method is DYNAMIC, chunk size of 5000 has a apparent negative impact versus chunk sizes of 1000 and 3000.

Scheduling method doesn't seem like having an effect on the results.

There is a huge difference (about 50%) between the timing values of 1 thread versus 2 threads. This suggests that the parallelism is definitely helping. From 2 threads to 4 threads, the results get slightly better. Generally, the best results are obtained with 4 threads of parallelism and the results of 5, 6, 7 and 8 threads are worse but very close to the results of 4 threads. This is expected since the CPU (Intel(R) Core(TM) i3-6100) on my machine has 2 cores supporting a maximum of 4 threads at a time.