# CMPE478 – Parallel Processing
# Assignment 2 Report
# Parallel Versions of Google Ranking Process Using MPI & Thrust

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

## Parsing Erdös Web Graph File (graph.txt) and Constructing P Matrix (in CSR format)

First of all, I calculated the number of nodes (vertices) and the number of edges in the graph using the bash commands below and hard-coded the results (**1,850,065** and **16,741,171**) as *#define* statements:

Vertex Number: `cat graph.txt | tr '\t' '\n' | sort | uniq | wc -l`
Edge Number: `cat graph.txt | 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 vector 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 ith element of **columnIndices**>*.

## Google Ranking Algorithm

Ranking algorithm is as follows:

$$\textit{REPEAT} \quad r^{(t+1)} = \alpha * P * r^{(t)} + (1-\alpha) * c \quad \textit{UNTIL} \quad \Sigma|r_i^{(t+1)} - r_i^{(t)}| \le \varepsilon$$

$$\textit{WHERE} \quad r^{(0)} = [1, 1, \ldots]^T, c = [1, 1, \ldots]^T, a = 0.2, \varepsilon = 10\text{-}6$$

**CMPE478 – Parallel Processing**
**Assignment 2 Report**
**Parallel Versions of Google Ranking Process Using MPI & Thrust**

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

## Part 1: Google Ranking Algorithm using MPI (hw2_mpi.cpp)

In this part, METIS (from Karypis Lab) is utilized in order to partition the Erdös Web Graph between different processors/processes. However, there is a little problem. Erdös Web Graph is a directed graph whereas METIS requires the graph to be undirected. I tried to convert the web graph to an undirected one in the C++ program, but it takes a bit longer (~5 minutes). So, I decided to implement a Python3 script (named **graph_to_metis.py**). This script takes the web graph file (**graph.txt**) as an argument and produces a text file named **metis_graph.txt** (~250MB) which stores the array arguments (xadj and adjncy) for \`METIS PartGraphRecursive\` C++ API. Note that this script also needs the hard-coded number of vertices (NODE_NUM) and edges (EDGE_NUM).

The logic of the MPI (C++) program is as follows:

1) **Process-0** begins by reading the graph file (**graph.txt**) and creating the related **P** matrix in CSR format. Then, it reads **metis_graph.txt** file, which stores the undirected version of the graph, and partitions that graph via METIS PartGraphRecursive() method. Then, **process-0** creates a map for itself (rankToNodes) which maps the processor ranks to a vector of node IDs assigned to that processor. Meanwhile, other processors wait on an MPI_Barrier.

2) **Process-0** distributes chunks of the **P** matrix to other processors (it takes a chunk too). **P** matrix is divided into chunks as follows: If **processor-i** is assigned the nodes i1, i2, … , iN; then $i1^{th}$, $i2^{th}$, …, $iN^{th}$ rows of the **P** matrix, say **partialP**, are sent to **processor-i** (again, in CSR format). **Processor-0** creates an array of doubles for **processor-i** dynamically which stores the following entities in the written order:
<number of nodes assigned to processor-i>|<node IDs>|<elements of rowBegin array of partialP>|<size of values array of partialP>|<elements of values array of partialP>|<elements of colIndices array of partialP>

**Process-0** creates **partialP** for the other processors and sends it via MPI_Send() method in a for-loop. Other processors first learns about the length of the **partialP** by probing via MPI_Probe() method, and then receives the array via MPI_Recv() method.

3) After an MPI_Barrier to ensure that each of the processors gets its chunk, processors continue with executing the ranking algorithm. $i^{th}$ element of $r^{(t+1)}$ vector is the product of the $i^{th}$ row of **P** matrix and the vector $r^{(t)}$ . Hence, all the processors need the vector $r^{(t)}$ after each iteration.
In a while-loop, **processor-0** broadcasts the full (updated) $r^{(t)}$ vector to every processor via MPI_Bcast() method. Every processor calculates a partial $\alpha * r^{(t+1)} * (1-\alpha)$ vector and calculates a partial sum of difference $|r_i^{(t+1)} – r_i^{(t)}|$, say partialSumOfDifference. Then, partialSumOfDifference values from every processor is reduced into sumOfDifference in every processors by summation via MPI_AllReduce() method. If partialSumOfDifference is less than some threshold, the while loop is broken and the program is terminated. Else, the loop continues with the next iteration of the ranking algorithm. Note that since **process-0** has the knowledge of which processor is assigned which nodes (via rankToNodes map), it can correctly update $r^{(t)}$ from partial $r^{(t+1)}$ vectors taken from other processors.

The duration of the ranking algorithm is measured with the help of MPI_Wtime() method. The algorithm converges after **14** iterations and takes about **4** seconds when executed with **4** processes on my machine (information about my machine can be found at the end of the report) as can be seen below:

```
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0
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:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ mpiexec --version
mpiexec (OpenRTE) 2.1.1

Report bugs to http://www.open-mpi.org/community/help/
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ ls
graph_to_metis.py  graph.txt  hw2_mpi.cpp
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ time python3 graph_to_metis.py graph.txt
        > graph.txt read.
        > Indices assigned to nodes.
        > METIS input arrays created.
        > ./metis_graph.txt created.

real    1m1,615s
user    0m58,046s
sys     0m2,680s
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ mpic++ --std=c++14 hw2_mpi.cpp -lmetis
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ ls
a.out  graph_to_metis.py  graph.txt  hw2_mpi.cpp  metis_graph.txt
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$ time mpirun -np 4 ./a.out graph.txt
Reading graph.txt                                                        ... DONE
Constructing CSR-format related arrays                                   ... DONE
Reading ./metis_graph.txt                                                ... DONE

Difference after 01 iterations: 371074.928382
Difference after 02 iterations: 12582.951876
Difference after 03 iterations: 1104.578951
Difference after 04 iterations: 113.339487
Difference after 05 iterations: 15.190087
Difference after 06 iterations: 2.114532
Difference after 07 iterations: 0.310768
Difference after 08 iterations: 0.045917
Difference after 09 iterations: 0.006935
Difference after 10 iterations: 0.001054
Difference after 11 iterations: 0.000162
Difference after 12 iterations: 0.000025
Difference after 13 iterations: 0.000004
Difference after 14 iterations: 0.000001

Time Elapsed for Ranking Algorithm: 3.928191 sec

real    1m12,705s
user    3m29,540s
sys     0m55,029s
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/mpi$
```

## Part 2: Google Ranking Algorithm using Thrust (hw2_thrust.cpp)

In this part, **values** array of CSR format is copied to the device through a **thrust::device_vector** named **values_D**. This is because matrix-vector multiplication will be done on the device. The variables with **_D** suffix are **thrust::device_vector** objects.

**CMPE478 – Parallel Processing**
**Assignment 2 Report**
**Parallel Versions of Google Ranking Process Using MPI & Thrust**

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

To implement the ranking algorithm, 3 **thrust::device_vector** objects are created on the device: **rt1_D** ($r^{(t)}$ vector), **rt2_D** ($r^{(t+1)}$ vector), and **difference_D** which stores the absolute value of **|** **rt2_D** - **rt1_D** **|** after each iteration.

Calculation of **rt2_D** from **rt1_D** is implemented as a for-loop on the host. This loop calculates one element of **rt2_D** at a time. At each iteration of the loop, dot product of the $i^{th}$ row of **P** matrix and **rt1_D** is calculated on the device with the help of **thrust::inner_product** function and **thrust::permutation_iterator** object. At the $i^{th}$ iteration of the loop, $i^{th}$ row of **P** matrix is extracted from **values_D** array as *values_D[rowBegin[i]] , … , values_D[rowBegin[i+1]]*. Also, a **thrust::permutation_iterator** object is created which iterates through *rt1_D[col_indices[j]]* for *rowBegin[i]* ≤ j ≤ *rowBegin[i+1]*. Lastly, result of the dot product is multiplied with α and added with (1-α).

After **rt2_D** is constructed, result of **|** **rt2_D** - **rt1_D** **|** is written to **difference_D** via **thrust::transform** function with a functor which returns the absolute value of the difference of its two parameters. As the final step, sum of the elements of **difference_D** is calculated on the device via **thrust::reduce** function. If the result is greater than ε, the host repeats the process; else, it terminates.

The duration of the ranking algorithm is measured with the help of clock() method of **C time library**. The algorithm converges after **14** iterations and takes about **2** seconds when executed with **OpenMP** backend (**4** threads) on my machine (information about my machine can be found at the end of the report) as can be seen below:

# CMPE478 – Parallel Processing
# Assignment 2 Report
# Parallel Versions of Google Ranking Process Using MPI & Thrust

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

```
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0
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:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2017 NVIDIA Corporation
Built on Fri_Nov__3_21:07:56_CDT_2017
Cuda compilation tools, release 9.1, V9.1.85
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ ~/Desktop/thrust/thrust_version
Thrust v1.9
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ ls
graph.txt  hw2_thrust.cu
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ nvcc -std=c++14 -O2 hw2_thrust.cu -Xcompiler -fo
penmp -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP -lgomp -I /usr/include/thrust/
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ ls
a.out  graph.txt  hw2_thrust.cu
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ export OMP_NUM_THREADS=4
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$ time ./a.out graph.txt
Reading graph.txt                                                         ... DONE
Constructing CSR-format related arrays                                    ... DONE

Difference after 01 iterations: 371074.928382
Difference after 02 iterations: 12582.951876
Difference after 03 iterations: 1104.578951
Difference after 04 iterations: 113.339487
Difference after 05 iterations: 15.190087
Difference after 06 iterations: 2.114532
Difference after 07 iterations: 0.310768
Difference after 08 iterations: 0.045917
Difference after 09 iterations: 0.006935
Difference after 10 iterations: 0.001054
Difference after 11 iterations: 0.000162
Difference after 12 iterations: 0.000025
Difference after 13 iterations: 0.000004
Difference after 14 iterations: 0.000001

Time Elapsed for Ranking Algorithm: 2.150102 sec

real    0m15,381s
user    0m15,269s
sys     0m0,841s
cba@cba-desktop:/DOSYA/BOUN/18-19-2/CMPE478/HOMEWORK/HW2/Code/thrust$
```

## Details of the Machine/CPU I Used

I tested my programs on a desktop computer on Linux Mint 19.1 Cinnamon (Linux Kernel: 4.15.0-50-generic).

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 |

## CMPE478 – Parallel Processing
## Assignment 2 Report
## Parallel Versions of Google Ranking Process Using MPI & Thrust

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

| | |
|---|---|
| CPU family: | 6 |
| Model: | 94 |
| Model name: | Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz |
| Stepping: | 3 |
| CPU MHz: | 800.026 |
| 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 md_clear flush_l1d