# CMPE 478 Assignment 2
# Page Ranking with MPI

Batuhan Celik(2018400051)
Ilgaz Er(2019400018)

Fall 2022

## Contents

## 1 Introduction

Aim is to calculate rankings for the websites in the Erdos Web Graph using page ranking algorithm proposed at The Anatomy of a Large-Scale Hypertextual Web Search Engine[1] by Brin and Page using 2 different parallelism paradigms: MPI and Thrusts.
MPI implementation consists of preprocessing the web graph to convert string ids to integer ids to reduce the average message size. Then, the graph is partitioned between processes using METIS, an open source graph partitioning tool, finally, ranks for partitioned graphs are computed as proposed at 1. Between each iteration, page rankings are synchronized between threads using prefix method.

## 2 MPI implementation

Our MPI implementation performs the following steps to find the highest 5 ranks in the graph:

- Graph Partitioning and preprocessing
- File digestion and node distribution
- Page rank synchronization between processes
- Page rank calculation
- $\sigma$ calculation and convergence checking
- Finding top 5 ranks

Following 5 sections will discuss these steps in depth, then, speedup results and testing methods will be investigated.

## 2.1 Graph Partitioning and Preprocessing

Erdos Web graph consists of 16 million edges indicated by sender/receiver pairs. However, this much information is not sufficient for our MPI implementation. One problem is that distributing nodes between processes. To achieve an efficient partitioning, we used METIS, which distributes nodes evenly between processes while minimizing the intra-process edges. Thus, converting Erdos Web graph representation to Metis representation was required.
These 2 representations differ in terms of:

- METIS requires integer ids whereas Erdos uses string ids.
- METIS is implemented for undirected graphs while web graph contains a directed graph.
- METIS does not allow nodes self pointing nodes.

Our preprocessing script reads the directed web graph edge by edge and converts it to a undirected graph with weighted edges following this equation:
$w(v,e) = f(v,e) + f(e,w) | e \neq w$
where $w(v,e)$ corresponds to the edge weight in METIS undirected graph and $f(e,w)$ corresponds to the number of edges with tail $e$ and head $v$ in the web graph. This function is selected because METIS aims to minimize weights of cutted edges when partitioning a edge weighted graph.
In addition, while iterating over edges 2 more operations are performed. Firstly vertices are enumareted as they occur in edges. Thus, string ids are converted to integer ids. Secondly, edges where tail and head are the same node are discarded while creating the METIS file since they do not contribute to the partitioning.
Finally, partitioning files are created for various number of threads, from 2 to up to 12, using METIS and the web graph is reconstructed in a file named "id_edges" with all the same edges while using integer ids and a mapping file matching integer ids with their string id in the web graph file. Our implementation will use the graph file containing integer ids since they result in a reduced MPI message size and finding their thread in METIS output file is easier.

## 2.2 File Digestion and Data Structures

File reading only performed at process 0 in our application as our files require considerable amount of memory. At the beginning of the program, process 0 reads the id_edges and METIS partitioning files line by line and converts the tokens it has seen to integers. Then not nodes, but edges are distributed between processes. An edge is sent to a process if either tail or head node is assigned to that process. To do so, process 0 creates 2 mappings contained in 4 vectors for other processes:

- `sender` - `receiver`
  Both vectors contain edge ids where intex `i` indicates an edge starting from `sender[i]` and ending at `receiver[i]`

- `external_node_ids` - `process_id`
  Given index `i`, `external_node_ids[i]` is a node id appearing in edges assigned to the process but `external_node_ids[i]` is an external node, meaning its rank will not be computed in this process, `process_id[i]` indicates the process the external node is assigned to.

Then, process 0 sends their corresponding vectors to other processes. Thus, each process gets the data contained in edges and partitioning files regarding their own nodes.
Upon completion of this transmission, each process initializes `Node` objects and updates their fields for its own nodes as the process iterates over its edges, class structure for these objects are given bellow:

```
class Node
{
public:
  long id;
  int num_outgoing_edges;
  std::vector<long> incoming_edges;
};
```

Where `id` is the integer assigned to the node in preprocessing phase. Number ob outgoing edges are stored in `num_outgoing_edges` and a set of node ids point to this node is stored at `incoming_edges`. These fields are populated as the process iterates over edges came from process 0.
Finally 3 different mappings are also initialized for each process for its nodes:

- `curr_ranks`
  Is an unordered map, mapping from node id to its ranks. Only stores the processes nodes.

- `curr_ranks_over_edges`
  Is a mapping from node $A$ to $\frac{P(A)}{C(A)}$ where $P(A)$ is page rank and $C(A)$ is the number of edges originating from $A$

- `next_ranks_over_edges`
  Values that will be assigned to `curr_ranks_over_edges` at the end of iteration.

## 2.3 Rank Synchronization Between Processes

During iterating over edges, processes populate a vector containing nodes they need to send to other processes. Then, these edges are broadcasted to all processes using the prefix calculation where the binary operation is vector insertion using the following algorithm executed by all processes:

```
$curr\_ranks\_over\_edges \leftarrow$ Node and rank pairs to be broadcasted
$receiving\_vector \leftarrow$ Node and rank pairs received from outside.
$step \leftarrow$ Step of the prefix operation.
$numproc \leftarrow$ Number of processes
$mypid \leftarrow$ Id of the processor
while $step < numproc$ do
    $target \leftarrow mypid - step$
    if $target < 0$ then
        $target \leftarrow target + numproc$
    end if
    $sender \leftarrow (mypid + step)\%numprocs$
    $MPI\_Isend(curr\_ranks\_over\_edges, target)$ rank vector send to target
    $MPI\_Ireceive(receiving\_vector, sender)$ receive vector received from sender
    $curr\_ranks\_over\_edges \leftarrow curr\_ranks\_over\_edges + receiving\_vector$
    $step \leftarrow step * 2$
end while
```

Upon completion of this broadcasting, `curr_ranks_over_edges` is updated using received vectors. Thus, all processes posses $\frac{P(A)}{C(A)}$ value regarding all nodes that are vertices of edges cut during partitioning.

## 2.4 Rank Calculation

Rank of a page is defined as:

$$P(A) = (1 - \alpha) + \alpha\Sigma_V \frac{P(V)}{C(V)}$$

where V indicates set of nodes annotating $A$ and $\alpha = 0.2$. Notice that $\frac{P(A)}{C(A)}$ values are broadcasted between processes in the previous step. Thus, in this step, all processes iterate over nodes and ranks in sequential manner and calculate the new ranks of web pages. Then, $\sigma = \Sigma_V |R_{i+1}(V) - R_i(v)|$ is calculated where V is set of processes internal nodes. Those $\sigma$ values are sent to master process and summed, thus, master process signals if the next iteration of ranks will be calculated to other processes.

## 2.5 Finding top 5 websites

To elicit the results, each process finds the top 5 highest ranks in its `curr_ranks` table and sends them to master process. Then, master process sorts the top 5 ranks from all processes and prints out the result.

## 2.6 Speedup Observations

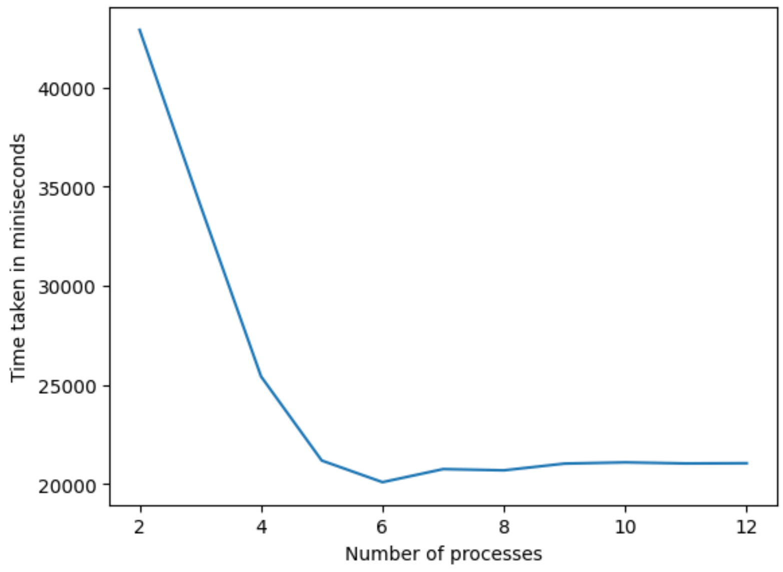

Table above displays execution time changing with the number of processors. Measurements were taken on a 6 core Intel i7 CPU. If we accept rank calculation as basic operation, $O(N)$, where N is the number of processes, speedup would be expected as one rank calculation. Which is inline with the observation up to 4 processes. However, after this point, MPI messages become huge and sending messages become the new basic operation. Prefix operation provides $O(log(N)$ speedup which is observed in 4 , 5 , 6 process experiments.
The processor we ran the experiment has 6 physical cores, thus, when number of processes exceed 6, all cores are populated and further speedup can not be observed. Moreover, context switches result in a decrease in performance.

## 3 Algorithmic Statements

To find the top websites, given P is the edge weight matrix with weight of edges incoming to a node summing up to 1 and are equal. following algorithm used.

```
$P \leftarrow$ weighted edge matrix
$r_i \leftarrow$ ranking vector at iteration i
$c \leftarrow$ ones vector
$\alpha \leftarrow 0.2$
$error \leftarrow 1$
**while** $error > 1e - 6$ **do**
    $r_{i+1} \leftarrow \alpha P r_i + (1 - \alpha)c$
    $error \leftarrow ||r_{i+1} - r_i||$
**end while**
```

In the end, last $r$ vector will be the converged rankings.
Given $P$ is a matrix, $r$ and $c$ are vectors, and $\alpha$ is a constant, we need to calculate $\alpha Pr + (1 - \alpha)$ to be able to perform ranking operations. P is stored in CSR format, thus, algorithms to perform linear operations on CSR matrices should be developed.
To perform this calculation on CSR matrices, algorithm given bellow is constructed under the assumption that $c$ vector will consists of 1s. We designed this algorithm around for loops to be able to parallelise it using OMP for loops.

```
$row\_begin \leftarrow$ CSR row begin array
$values \leftarrow$ CSR values array
$col\_indices \leftarrow$ CSR column indices array
$r \leftarrow$ Multiplied vector array
$r\_next \leftarrow$ Resulting vector array
$\alpha \leftarrow 0.2$
**for** $row$ from 1 to $row\_begin.length - 1$ **do**
    $sum \leftarrow 0$
    **for** $col\_indice$ from $row\_begin[row]$ to $row\_begin[row + 1]$ **do**
        $col \leftarrow col\_indices[col_indice]$
        $sum+ = r[col] * values[col_indice]$
    **end for**
    $r\_next[row] = sum * \alpha + (1 - \alpha)$
**end for**
```