# Project #1 Part 2: MPI PageRank (Due Sep. 24th, 11:59 pm)

1. **Deliverables**

You are required to turn in following items in a zip file (**groupID\_Project1P2.zip**) in this assignment

1) The source code of parallel PageRank you implemented.

2) The executable class file, the README file that describe its usage.

3) Technical report (**groupID\_Project1P2\_report.docx**) that contains:

a. The description about the main steps and data flow in your program.

b. The output file (**groupID\_pagerank\_output.txt**) which contains the top 10 ranking url numbers.

Note that, this document only provides MPI C/C++ instruction. If you prefer or know MPJExpress <http://mpj-express.org/>, please free feel to implement with it

Points will be reduced (maximum 0.5 point) if the filename or directory structure are different from instructed above.

# Evaluation

The total points of project #1 Part 1 is 5, where the distribution is as following

* 1. Completeness of your code and output (3 point)
  2. Correctness of written report (1.5 point)
  3. Readability and clarity of README.txt (0.5 point)

# Introduction

As the second part of Project #1, you will implement a parallel version of PageRank by using MPI programming interface. We provide a pseudo code for MPI PageRank and you need to complete the implementation based on the pseudo code to make a working version of it. We provide functions for file IO so that you can focus on main implementation only. For more details of PageRank algorithm, you can refer to the previous descriptions for part 1 of Project #1.

## Parallel PageRank

Developing parallel PageRank is an active research area for both in industry and academia and numerous algorithms have been proposed. The key idea in developing parallel PageRank is to partition PageRank problem into N sub problems so that N processes solve each sub-problem concurrently. One of simple approaches in partitioning is a vertex-centric approach. The graph of PageRank can be divided into groups of vertices and each group will be processed by a process. We take this approach for our MPI PageRank implementation.

# Pseudo code for MPI PageRank

You need to complete two files: mpi\_main.c and mpi\_pagerank.c. Pseudo codes are as follows.

**mpi\_main.c**

#include <mpi.h>

#include "pagerank.h"

**int main(int argc, char \*\*argv)**

{

/\* Definition of data structure and variables for MPI PageRank \*/

int numUrls, totalNumUrls;

char \*filename;

int \*\*am\_index; /\* int[numUrls][2] \*/

/\* am\_index[i][0]refers to second index for am, am\_index[i][1] refers to length of target urls list \*/

int \*adjacency\_matrix; /\* [numTotalWebPages] \*/

double \*rank\_values\_table; /\* [numUrls] \*/

double threshold;

int rank, nproc, mpi\_namelen;

char mpi\_name[MPI\_MAX\_PROCESSOR\_NAME];

/\* MPI Initialization \*/

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);

MPI\_Get\_processor\_name(mpi\_name, &mpi\_namelen);

/\* Parse command line arguments \*/

/\* Read local adjacency matrix from file for each process \*/

mpi\_read(filename, &numUrls, &am\_index, &adjacency\_matrix, MPI\_COMM\_WORLD);

/\* Set totalNumUrls \*/

MPI\_Allreduce(&numUrls, &totalNumUrls, 1, MPI\_INT, MPI\_SUM,MPI\_COMM\_WORLD);

/\* Global page rank value table \*/

rank\_values\_table = (double \*) malloc(totalNumUrls \* sizeof(double));

assert(rank\_values\_table != NULL);

/\* Root(rank 0) computes the initial rank value for each web page \*/

/\* Broadcast the initial rank values to all other compute nodes \*/

MPI\_Bcast(rank\_values\_table, totalNumUrls, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

/\* Start the core computation of MPI PageRank \*/

mpi\_pagerank(adjacency\_matrix, am\_index, numUrls, totalNumUrls,

num\_iterations, threshold, rank\_values\_table,

MPI\_COMM\_WORLD);

/\* Save results to a file \*/

/\* Release resources e.g. free(adjacency\_matrix); \*/

MPI\_Finalize();

return (0);

}

**mpi\_pagerank.c**

#include <mpi.h>

#include "pagerank.h"

/\* Define mpi\_pagerank function with following signature \*/

**int mpi\_pagerank(**

int **adjacency\_matrix**, /\* adjacency matrix for pagerank \*/

int **\*\*am\_index**, /\* index array for adjacency matrix \*/

int **numUrls**, /\* num of urls assigned to local machine \*/

int **total\_num\_urls**, /\* num of total urls \*/

int **num\_iterations**, /\* num of maximum interations \*/

double **threshold**, /\* control the number of iterations \*/

double **\*rank\_values\_table**, /\* double[total\_num\_urls] \*/

**MPI\_Comm comm)** /\* MPI communicator \*/

{

/\* Definitions of variables \*/

/\* Get MPI rank \*/

MPI\_Comm\_rank(comm, &rank);

/\* Allocate memory and initialize values for local\_rank\_values\_table \*/

/\* Start computation loop \*/

do

{

/\* Compute pagerank and dangling values \*/

/\* Distribute pagerank values \*/

MPI\_Allreduce(local\_rank\_values\_table, rank\_values\_table,

total\_num\_urls, MPI\_DOUBLE, MPI\_SUM, comm);

/\* Distribute dangling values \*/

MPI\_Allreduce(&dangling, &sum\_dangling, 1, MPI\_DOUBLE, MPI\_SUM, comm);

/\* Recalculate the page rank values with damping factor 0.85 \*/

/\* Root(process 0) computes delta to determine to stop or continue \*/

/\* Root broadcasts delta \*/

MPI\_Bcast(&delta, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

while (delta > threshold && loop++ < num\_iterations);

return 1;

}

# FILE IO functions

Besides pseudo codes, we provide two MPI functions (mpi\_read and mpi\_write) for file reading and writing, so that you can simply call them without re-implementation.

**mpi\_read**

This function is to read an adjacency matrix file and assign a local matrix to each process. After this call each process will have its own adjacency matrix. Thus, each process should call this function altogether. Following is a signature of mpi\_read function.

/\*

\* Read an adjacency matrix file and assign a local matrix to each process

\* Note

\* -. After this call, each process will have its own adjacency matrix

\* -. Each process will have roughly eqaul amount of number of urls

\*/

int mpi\_read(char \*filename, /\* (IN) file name \*/

int \*numUrls, /\* (OUT) number of urls assigned to local machine \*/

int \*\*\*am\_index, /\* (OUT) index array for adjacency matrix, assigned to local machine \*/

int \*\*adjacency\_matrix, /\* (OUT) adjacency matrix assigned to local machine \*/

MPI\_Comm comm); /\* (IN) MPI communicator \*/

**mpi\_write**

This function is for saving rank values to a file. We assume rank values are synchronized through all processes. Thus, it doesn’t matter which process to write. In this function, simply we have root (rank 0) process to write rank values to a file.

/\*

\* Save rank\_values\_table[] to a file

\* Note

\* -. Only root (rank 0) will save rank values to a file

\* -. Assume rank\_values\_table[] are the same across all proc

\*/

int mpi\_write(char \*filename, /\* (IN) file name \*/

int totalNumUrls, /\* (IN) number of total urls \*/

double \*rank\_values\_table, /\* (IN) array of rank values. double[total\_num\_urls] \*/

MPI\_Comm comm); /\* (IN) MPI communicator \*/

# Compile guide

Instructions for compiling and recommended execution guide are as follows (We assume that you will compile and run on machines of CS department, such as burrow cluster).

**Step one: Compile**

To compile MPI code, you need to set up correct MPI development environments. You can do that by using “mpi-selector” command:

First, check which MPI versions are available:

$ mpi-selector –list

In Silo, you can see the following list:

lam-i386

lam-x86\_64

openmpi-1.4-gcc-i386

openmpi-1.4-gcc-x86\_64

Second, you choose one of MPI versions in the list by executing the following command:

$ mpi-selector --set openmpi-1.4-gcc-i386

You may see a warning for overwriting your previous setting. You can answer “yes”

Defaults already exist; overwrite them? (y/N) y

Now you are ready to compile the sample simply by using “make” command (We provide Makefile)

$ make clean

$ make

You will see an executable “mpi\_main” in the directory.

**Step two: Execution**

The executable you need to make should take a few parameters as an input. Most importantly we should be able to specify input file by using “-i” option and the number of iterations by using “-n” option. The full list of options you need to implement is as follows:

$ mpi\_main -h

Usage: mpi\_main -i filename -n num\_iterations

-i filename : adjacency matrix input file

-n num\_iterations: number of iterations

-t threshold : threshold value (default 0.0010)

-o : output timing results (default yes)

-d : enable debug mode

-h : print help information

To run “mpi\_main”, you need to use a MPI command called “mpirun”. For an example, to process “pagerank.input” file (included as a sample input) by using 2 processes concurrently, you can type the following commands:

$ mpirun -np 2 mpi\_main -i pagerank.input -n 10 -t 0.001

The following is an example of execution of MPI PageRank.

max\_iterations=10, threshold=0.001000000

->cur\_iteration=0 delta=0.137787065

->cur\_iteration=1 delta=0.101166196

->cur\_iteration=2 delta=0.033934278

->cur\_iteration=3 delta=0.026608634

->cur\_iteration=4 delta=0.018108631

->cur\_iteration=5 delta=0.014472174

->cur\_iteration=6 delta=0.010200567

->cur\_iteration=7 delta=0.007725284

->cur\_iteration=8 delta=0.005532020

->cur\_iteration=9 delta=0.004065336

->cur\_iteration=10 delta=0.002924901

Proc:0 is writing rank values of 11 urls to file pagerank.output

\*\*\*\* MPI PageRank \*\*\*\*

Num of processes = 2

Input file = pagerank.input

totalNumUrls = 11

num\_itertaions = 10

threshold = 0.001000000

I/O time = 0.4031 sec

Computation timing = 0.0002 sec