Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2021
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: A. Dolfini, L. Gaedke-Merzhaeuser
T. Holt, R. Janalik, I. Labarca, D. Pasadakis,
G. Pollak, M. Sudwoj

Project 4
Parallel Programming using the Message Passing Interface MPI
Due date: 26 April 2021, 12pm (midnight)

This assignment will introduce you to parallel programming using MPI. You will implement a simple MPI message exchange, compute a process topology, parallelize the computation of the Mandelbrot set, and finally either a parallel matrix-vector multiplication within the Power iteration (Option A) or a parallel PageRank Algorithm using the Power iteration (Option B). You may do this project in groups of two or three students. In fact, we prefer that you do so.

1. Ring addition using MPI [10 Points]

This example demonstrates basic MPI send/receive functionality and identification of the neighbors in a 1-dimensional process grid layout. The processes are organized in a circular chain, where each process has 2 neighbors and first and last processes are neighbors as well. The algorithm proceeds in a following way: every process initially sends its rank number to a neighbor; then every process sends what it receives from that neighbor. This is done n times, where n is the number of processes. As a result, all ranks will obtain the sum of all ranks. The first two iterations of the algorithm are illustrated in Figure 1.

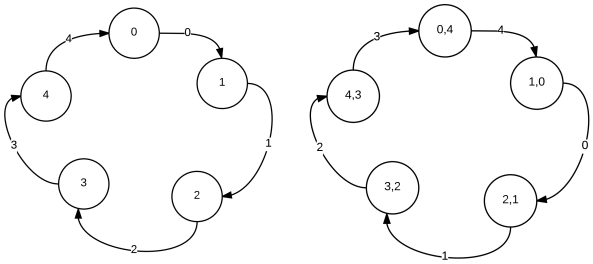

Figure 1: First two iterations of the ring sum algorithm.

Solve the following tasks:

1. Determine the left/right neighbors of each process.
2. Implement a ring addition code.

Compilation and execution with 4 ranks on the Euler cluster :

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi/3.0.0
[user@eu-login]$ cd ring/
[user@eu-login]$ make
[user@eu-login]$ bsub -n 4 -W 00:10 mpirun ./sum_ring
```

and the result of the parallel computation should be the sum of all four MPI ranks (0,1,2,3):

```
Process 0:      Sum = 6
Process 1:      Sum = 6
Process 2:      Sum = 6
Process 3:      Sum = 6
```

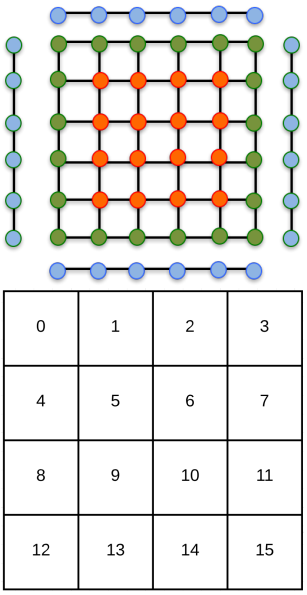

Figure 2: Local domain (left) and processor grid (right).

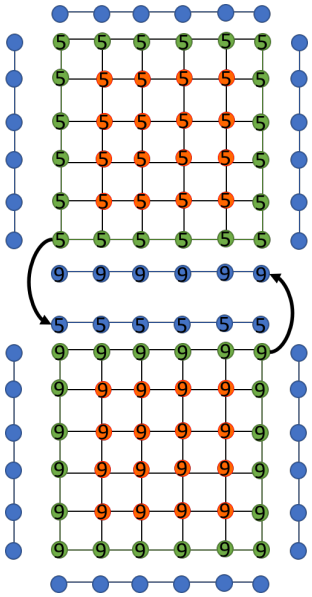

Figure 3: Exchange of the ghost cells.

2. Ghost cells exchange between neighboring processes [15 Points]

The objective of this problem is to write a parallel program, using MPI, to exchange ghost cells between neighboring processes. The term "ghost cell" refers to a copy of remote process' data in the memory space of the current process. The ghost data is often needed for local computations, therefore, before proceeding with the computation, the processes need to exchange their ghost cells first. In this simple example, each process has a 6×6 local domain which is extended by 1 row/column from each side in order to accommodate the copy of its neighbors' borders (also known as "ghost cells"). The local domain is illustrated in red and green in Figure 2. The green cells are the local data that will be communicated to other processes. On the other hand, the ghost cells, containing the copy of the remote processes' data, are indicated by the color blue. The local domain extended by ghost cells therefore has size $(6 + 2) \times (6 + 2)$ (we will ignore the corners). In order to easily inspect the result of the communication, we initialize the local domain of each process by its `mpi_rank`. Furthermore, we will assume we have an $n \times n$ grid of processes organized in a Cartesian topology, as shown in the bottom image of Figure 2. We also assume cyclic borders, which means that, for example, process 0 is also a neighbor of process 3 and 12. Each process has 4 neighbors, sometimes referred to as the neighbor north, south, east and west. For testing purposes, always assume we are launching 16 processes. The exchange of the borders between process 5 and 9 is illustrated in Figure 3. The compilation and execution on the Euler cluster are shown below:

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi/3.0.0
[user@eu-login]$ cd ghost/
[user@eu-login]$ make
[user@eu-login]$ bsub -n 16 -W 00:10 mpirun ./ghost
```

Note: If you are running more processes than the available number of CPUs, you will need to oversubscribe your CPU, see here:

```
[user@eu-login]$ mpirun -np 16 --oversubscribe ./ghost
```

The result of the boundary exchange on rank 9 should be

```
9.0 5.0 5.0 5.0 5.0 5.0 5.0 9.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
```

```
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
9.0 13.0 13.0 13.0 13.0 13.0 13.0 9.0
```

Note that `mpirun` automatically uses all cores allocated to the job by LSF . It is therefore not necessary to indicate this number again following the `mpirun` command.

Solve the following tasks:

1. Create a Cartesian 2-dimensional communicator (4×4) with periodic boundaries and use it to find your neighboring ranks in all dimensions in a cyclic manner.
2. Create a derived data type for sending a column border (east and west neighbors).
3. Exchange ghost cells with the neighboring cells in all directions and verify that correct values are in the ghost cells after the communication phase.

3. Parallelizing the Mandelbrot set using MPI [20 Points]

In this subproject, we will parallelize the Mandelbrot set computation from Project 2 using MPI. The computation of the Mandelbrot set will be partitioned into a set of parallel MPI processes, where each process will compute only its local portion of the Mandelbrot set. Examples of a possible partitioning are illustrated in Figure 4. After each process completes its own computation, the local domain is sent to the master process that will handle the I/O and create the output image containing the whole Mandelbrot set.

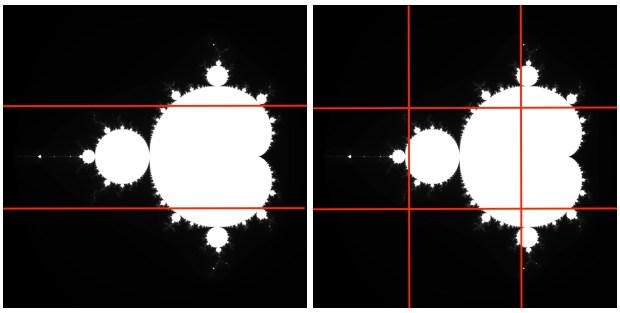

Figure 4: Possible partitionings of the Mandelbrot set.

We introduce two structures, that represent the information about the partitioning. These structures are defined in `consts.h`. The structure `Partition` represents the layout of the grid of processes and contains information such as the number of processes in x and y directions and the coordinates of the current MPI process.

```
typedef struct
{
    int x; // Coordinates of the current MPI process within the processor grid
    int y;
    int nx; // Dimensions of the processor grid
    int ny;
    MPI_Comm comm;
} Partition;
```

The second structure `Domain` represents the information about the local domain of the current MPI process. It holds information such as the size of the local domain (number of pixels in each dimension) and its global indices (index of the first and the last pixel in the full image of the Mandelbrot set that will be computed by the current process).

```
typedef struct
{
    long nx; // Size of the local domain
    long ny;
    long startx; // Begining of the local domain (global index)
    long starty;
    long endx; // End of the local domain (global index)
    long endy;
} Domain;
```

The code you will find on Moodle or on the Git repository is initialized in a way that each process computes the whole Mandelbrot set. Your task will be to partition the domain (so that each process only computes an appropriate part), compute the local part of the image, and send the local data to the master process that will create the final complete image. The compilation and execution on the Euler cluster are shown below:

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi/3.0.0
[user@eu-login]$ cd mandel/
[user@eu-login]$ make
[user@eu-login]$ bsub -n 16 -W 01:00 mpirun ./mandel_mpi
```

Running the benchmark and creating the performance plot (please compile and run on the compute node):

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi/3.0.0
[user@eu-login]$ cd mandel/
[user@eu-login]$ make
[user@eu-login]$ bsub -n 16 -W 01:00 < run_perf.sh
[user@eu-login]$ ./plot_perf.sh
```

Solve the following tasks:

1. Create the partitioning of the image by implementing a body of functions `Partition createPartition(int mpi_rank, int mpi_size)` and `Partition updatePartition(Partition p, int mpi_rank)`. You can find a dummy implementation of these functions in `consts.h`.

2. Determine the dimensions and the start/end of the local domain based on the computed partitioning by implementing a function `Domain createDomain(Partition p)`. The function is defined in the `consts.h`.

3. Send the local domain to the master process if `mpi_rank > 0` and receive it at the master process where `mpi_rank == 0`. Compare the output of the parallelized program to that of the sequential program in a graphic and verify that it is correct.

4. Analyze the performance in the graph `perf.ps`. It shows the computational time of each process for multiple runs with a varying number of processes. Comment on the benefits of the MPI parallelization and load balancing you have observed. How does the performance change when running on 1 compute node vs. multiple nodes?

4. Option A: Parallel matrix-vector multiplication and the power method [40 Points]

This subproject[1] is about to write a parallel program to multiply a matrix A by a vector x, and to use this routine in an implementation of the power method to find the absolute value of the largest eigenvalue of the matrix. Your code will call routines that we supply to generate matrices, record timings, and validate the answer.

[1] Only one of the two subprojects *Option A* or *Option B* is needed to obtain all points. You can either select *Option A* or *Option B*. If you decide to submit a solution for both subprojects then we will only count the best of these two submissions towards your entire grade for this project.

4.1. Mathematical background

A square matrix A is an n-by-n array of numbers. The entry in row i, column j of A is written either a_{ij} or $A(i,j)$. The rows and columns are numbered from 1 to n. A vector is a one-dimensional array x whose i'th entry is x_i or $x(i)$. Recall the definition of matrix-vector multiplication: The product $y = Ax$ is a vector y whose elements are

$$y_i = \sum_{j=1}^{n} a_{ij} x_j. \tag{1}$$

In words, each element of y is obtained from one row of A and all of x, by computing an inner product (that is, by adding up the pointwise products). Every element of x contributes to every element of y; each element of A is used exactly once. The *power method* uses matrix-vector multiplication to estimate the size of the largest absolute eigenvalue of the matrix A, which is also called the spectral radius of A. It works as follows. Start with an arbitrary vector x. Then repeat the following two steps: divide each element of x by the norm of x; second, replace x by the matrix-vector product Ax. The length of the vector eventually converges to the spectral radius of A. In your code, you will repeat the matrix-vector product 1000 times. There is a sequential Matlab code `powermethod` for the Power method on the Moodle course web page [2] and on the git repository[3].

```
function lambda = powermethod(A)
% POWERMETHOD : Power method to estimate norm of dominant eigenvalue
%
% lambda = powermethod(A);
%
% This routine generates a random vector, then repeatedly
% multiplies matrix A by the vector and normalizes the result
% to have length 1.  The norm converges to the magnitude of
% the dominant eigenvalue, which is returned as lambda.
%
% This is a sequential Matlab template -- your task is
% to implement this in parallel.
%
[n,n] = size(A);       % number of rows and columns in A
x = rand(n,1);         % creates a starting n-vector
for i = 1:1000
    x = x / norm(x);   % norm(x) is sqrt(x(1)^2 + x(2)^2 + ... + x(n)^2)
    x = A * x;         % use your parallel matrix * vector routine here
end;
lambda = norm(x);
```

Your task is to write a parallel C/MPI code that does the same computation.

4.2. Test harnesses

We will supply some 3 routines that you can call from your code for testing. We will also use these routines in grading your program. The routines include a matrix generator `hpc_generateMatrix`, a validator that checks correctness of the results `hpc_verify` (for the particular matrix we generate), and a timer `hpc_timer`.

4.3. What to implement

You need to write the following MPI parallel C routines:

[2] https://moodle-app2.let.ethz.ch/course/view.php?id=14316
[3] https://gitlab.math.ethz.ch/karoger/hpclab_fs2021

- `generateMatrix`: Although we will supply a matrix generator for grading, you should also generate various matrices for testing your code. This routine should generate a matrix of specified size, with the data distributed across the processors as specified in the next section.
- `powerMethod`: Implements the power method on a given matrix (which is already distributed across the processors). This routine calls `norm` and `matVec`.
  - `norm`: Computes the norm of a given vector.
  - `matVec`: Multiplies a given matrix (which is already distributed across the processors) by a given vector.
- `main`: The main routine that calls `generateMatrix` and e.g. $k = 1000$ times the `powerMethod` routine.

4.4. Where's the data?

To simplify the parallel code you may assume that n, the number of rows and columns of the matrix, is divisible by p, the number of processors. Distribute the matrix across the processors by rows, with the same number of rows on each processor; thus, processor 0 gets rows 1 through n/p of A, processor 1 gets rows $n/p + 1$ through $2n/p$, and so forth. Your `generateMatrix` routine should not do any communication except for the value of n; each processor should generate its own rows of the matrix, independently of the others, in parallel. Put the vector on processor 0. For our purposes, the "arbitrary vector" you start with can be a vector of random doubles. When you write your `matVec` routine, you should do the communication with `MPI_Bcast` and `MPI_Gather`; you will find the code to be much simpler this way than if you do it all with `MPI_Send` and `MPI_Recv`.

4.5. What experiments to do

First, debug your code on very small matrices, using one MPI process, then two processes, then several MPI processes. Two matrices you can use for debugging are the n-by-n matrix of all ones (whose spectral radius is n) and the identity matrix (with ones on the main diagonal and zeros elsewhere), whose spectral radius is 1. For debugging you should probably only do a few iterations of the power method instead of 1000. Your code should only time the call to `powerMethod`, not the matrix generation. You should call our harness routines to start and stop the "official" timer; you can also use `MPI_Wtime` for your own timings. Here are some experiments to do:

- **Strong scaling analysis.** Choose a value for n for which your code runs on one process in a reasonable amount of time, say 30 seconds to a minute, with 1,000 iterations of the main loop. (On my laptop, $n = 10,000$ takes about 50 seconds.) Run your code for $p = 1, 4, 8, 12, 16, 32$, and (if possible) 64. Also run your code using various number of Euler cluster nodes. For each run, report the running time and the parallel efficiency. Make plots of the running time versus p, and the parallel efficiency versus p.
- **Weak scaling analysis.** Change your program to do only 100 iterations of the main loop, to make the experiments in this part run faster. Now choose a starting value of n to use for $p = 1$, possibly the same n as above. Run your code for $p = 1, 4, 8, 12, 16, 32$, and 64, but this time use a different n for each p, chosen so that n is (nearly) proportional to $\sqrt{p}$. Since both total memory and total work scale as n^2 , this implies that the memory required per processor and the work done per processor will remain constant as you increase p. This is called weak scaling. Again, report the running time and the parallel efficiency for each run, and make the same plots you did for the previous experiment.

4.6. Extra credit

For extra credit, you may experiment with

- **Different data layouts**. If the matrix is distributed by columns instead of by rows, your `matVec` routine need to use `MPI_Scatter` and `MPI_Reduce`. You can also try distributing the matrix by blocks. In that case, you may want to use separate MPI communicators for the rows and columns of processors. One way to do this is with `MPI_Comm_split`.
- You may also try **ScaLAPACK - Scalable Linear Algebra PACKage** which is a library of high-performance linear algebra routines for parallel distributed memory machines.

5. Option B: Parallel PageRank Algorithm and the Power method [40 Points]

The purpose of this subproject[4] is to learn the importance of parallel numerical linear algebra algorithms to solve fundamental linear algebra problems that occur in search engines.

5.1. The initial Page-Rank Algorithm – Parallel Eigenvalue Graph Computation

One of the first reasons why Google is such an effective search engine is the Page-Rank algorithm developed by Google's founders, Larry Page and Sergey Brin, when they were graduate students at Stanford University. PageRank is determined entirely by the link structure of the World Wide Web. In 2002, it was recomputed about once a month and did not involve the actual content of any Web pages or individual queries. Then, for any particular query, Google finds the pages on the Web that match that query and lists those pages in the order of their PageRank. Imagine surfing the Web, going from page to page by randomly choosing an outgoing link from one page to get to the next. This can lead to dead ends at pages with no outgoing links, or cycles around cliques of interconnected pages. So, a certain fraction of the time, simply choose a random page from the Web. This theoretical random walk is known as a *Markov chain* or *Markov process*. The limiting probability that an infinitely dedicated random surfer visits any particular page is its PageRank.
A page has high rank if other pages with high rank link to it. Let W be the set of Web pages that can be reached by following a chain of hyperlinks starting at some root page and let n be the number of pages in W. Let G be the n-by-n connectivity matrix of a portion of the Web, that is $g_{ij} = 1$ if there is a hyperlink to page i from page j and zero otherwise. The matrix G can be huge, but it is very sparse. Its j-th column shows the links on the j-th page. The number of nonzeros in G is the total number of hyperlinks in W. Let r_i and c_j be the row and column sums of G:

$$r_i = \sum_j g_{ij}, \quad c_j = \sum_i g_{ij}, \tag{2}$$

The quantities r_j are the *in-degree* and c_j are the *out-degree* of the j-th page. Let p be the probability that the random walk follows a link. A typical value is $p = 0.85$. Then $1 - p$ is the probability that some arbitrary page is chosen and $\delta = (1 - p)/n$ is the probability that a particular random page is chosen.
Let A be the n-by-n matrix whose elements are

$$a_{ij} = \begin{cases} p\ g_{ij}/c_j + \delta & c_j \neq 0 \\ 1/n & c_j = 0. \end{cases} \tag{3}$$

Notice that A comes from scaling the connectivity matrix by its column sums. The j-th column is the probability of jumping from the j-th page to the other pages on the Web. If the j-th page is a dead end, that is, it has no out-links, then we assign a uniform probability of $1/n$ to all the elements in its column. Most of the elements of A are equal to δ, the probability of jumping from one page to another without following a link. If $n = 4 \cdot 10^9$ (current estimation of

[4]This description is a revised version of a SIAM book chapter from *Numerical Computing with Matlab* from Clever B. Moler that has been published in 2002.

webpages) and $p = 0.85$, then $\delta = 3.75 \cdot 10^{-11}$. The matrix A is the transition probability matrix of the Markov chain. Its elements are all strictly between zero and one and its column sums are all equal to one. An important result in matrix theory known as the *Perron-Frobenius theorem* applies to such matrices. It concludes that a nonzero solution of the equation

$$x = Ax \tag{4}$$

exists and is unique to within a scaling factor. If this scaling factor is chosen so that

$$\sum_i x_i = 1, \tag{5}$$

then x is the *state vector* of the Markov chain and is *Google's PageRank*. The elements of x are all positive and less than one. For modest n, an easy way to compute x using Matlab notations is to start with some approximate solution, such as the PageRanks from the previous month, or

```
x = ones(n,1)/n
```

Then simply repeat the assignment statement

```
x = x / norm(x);   % norm(x) is sqrt(x(1)^2 + x(2)^2 + ... + x(n)^2)
x = A*x;
```

until successive vectors agree to within a specified tolerance. This is known as the Power method and is about the only possible approach for very large n. In practice, the matrices G and A are never actually formed. One step of the power method would be done by one pass over a database of Web pages, updating weighted reference counts generated by the hyperlinks between pages. The best way to compute PageRank in Matlab is to take advantage of the particular structure of the Markov matrix. Here is an approach that preserves the sparsity of G. The transition matrix can be written

$$A = pGD + ez^T, \tag{6}$$

where D is the diagonal matrix formed from the reciprocals of the out-degrees,

$$d_{jj} = \begin{cases} 1/c_j & c_j \neq 0 \\ 0 & c_j = 0. \end{cases} \tag{7}$$

e is the n-vector of all ones, and z is the vector with components

$$z_j = \begin{cases} \delta = (1-p)/n & c_j \neq 0 \\ 1/n & c_j = 0. \end{cases} \tag{8}$$

The rank-one matrix ez^T accounts for the random choices of Web pages that do not follow links. The **Power method** can be implemented in a way that does not actually form the Markov matrix and so preserves sparsity. Compute

```
G = p*G*D;
z = ((1-p)*(c~=0) + (c==0))/n; // this implements in Matlab the equation (8)
```

Start with

```
x = e/n
```

and then repeat the statement

```
x = G*x + e*(z*x)
```

until x settles down to several decimal places. The full Page Rank method in Matlab is shown below.

```
function x = pagerank(G,U)
% PAGERANK  Google's PageRank
% x = pagerank(U, G,p) uses the power method to compute the page
% rank for a connectivity matrix G with a damping factory p,
% (default is p = 0.85).
p = .85;
% Eliminate any self-referential links
G = G - diag(diag(G));
% c = out-degree
[n,n] = size(G);
c = sum(G,1);
% Form the components of the Markov transition matrix.
k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);
G = p*G*D;
e = ones(n,1);
z = ((1-p)*(c~=0) + (c==0))/n;
% Power method to find Markov vector, A*x = x.
x = e/n;
it = 0;
xs = zeros(n,1);
while norm(x-xs,inf)> 1.e-6*norm(x,inf)
   xs = x;
   x = G*x + e*(z*x);
   it = it + 1;
   disp(['Iteration #', int2str(it), '; Error: ', num2str(norm(x-xs,inf))]);
end
```

5.2. Example – A tiny webgraph

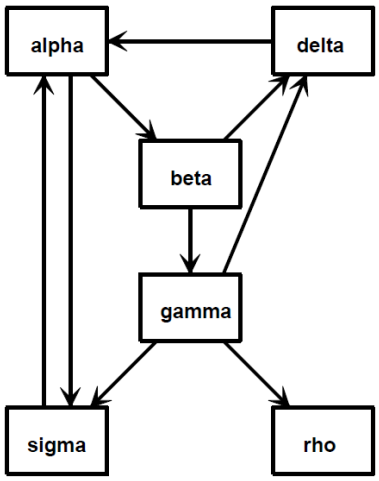

Figure 5: A tiny web graph.

Figure 5 is the graph from a tiny web example, with $n = 6$. This small example involves 6 vertices (webpages):

```
U = {'www.alpha.com'
     'www.beta.com'
     'www.gamma.com'
     'www.delta.com'
     'www.rho.com'
     'www.sigma.com'}
```

We can generate the connectivity graph matrix by specifying the pairs of indices (i, j) of the nonzero elements. Because there is a link to `www.beta.com` ($i = 2$) from `www.alpha.com` ($j = 1$), the $(2, 1)$ element of G is nonzero. The nine connections (i, j) of G are described by

```
i = [ 2 6 3 4 4 5 6 1 1]
j = [ 1 1 2 2 3 3 3 4 6]
```

A sparse matrix is stored in a data structure that requires memory only for the nonzero elements and their indices. This is hardly necessary for a 6-by-6 matrix with only 27 zero entries, but it becomes crucially important for larger problems. The Matlab statements

```
n = 6
G = sparse(i,j,1,n,n);
full(G)
```

generate the sparse representation of an n-by-n matrix with ones in the positions specified by the vectors i and j and display its full representation of G

```
     0     0     0     1     0     1
     1     0     0     0     0     0
     0     1     0     0     0     0
     0     1     1     0     0     0
     0     0     1     0     0     0
     1     0     1     0     0     0
```

The statement

```
c = full(sum(G))
```

computes the column sums (or out-degree of these webpages)

```
 2     2     3     1     0     1
```

Notice that $c(5) = 0$ because the 5th page, labeled rho, has no out-links. For this tiny example with $p = .85$, the smallest element of the Markov transition matrix A is $\delta = .15/6 = .0250$.

```
A =
  0.0250 0.0250  0.0250  0.8750  0.1667  0.8750
  0.4500 0.0250  0.0250  0.0250  0.1667  0.0250
  0.0250 0.4500  0.0250  0.0250  0.1667  0.0250
  0.0250 0.4500  0.3083  0.0250  0.1667  0.0250
  0.0250 0.0250  0.3083  0.0250  0.1667  0.0250
  0.4500 0.0250  0.3083  0.0250  0.1667  0.0250
```

Notice that the column sums of A are all equal to one. The Power method applied to

```
x = G*x + e*(z*x)
```

and implemented in `pagerank.m` (Matlab) using

```
i = [2 6 3 4 4 5 6 1 1];
j = [1 1 2 2 3 3 3 4 6];
n = 6;
G = sparse(i, j, 1, n, n);
U = {'http://www.alpha.com';
     'http://www.beta.com' ;
     'http://www.gamma.com';
     'http://www.delta.com';
     'http://www.rho.com'  ;
     'http://www.sigma.com'};
x = pagerank(G,U)
```

or in `pagerank.c`

```
./pagerank tinygraph.mat
```

solves for the PageRank vector x until it settles down to several decimal places to produce

```
x =
0.3210
0.1705
0.1066
0.1368
0.0643
0.2007
```

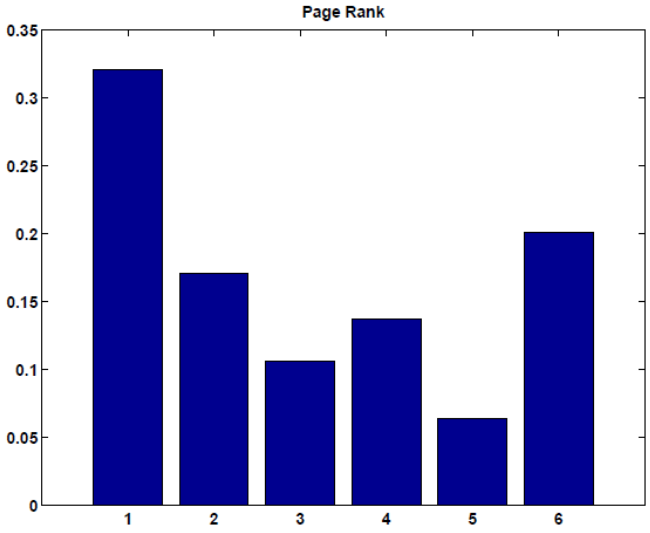

Figure 6: Page Rank for the tiny web graph.

The bar graph of x is shown in Figure 6. If the URLs are sorted in PageRank order and listed along with their in- and out-degrees, and

```
x = pagerank(G,U)
```

results in

```
   page-rank   in      out     url
1  0.3210       2       2     www.alpha.com
6  0.2007       2       1     www.sigma.com
2  0.1705       1       2     www.beta.com
4  0.1368       2       1     www.delta.com
3  0.1066       1       3     www.gamma.com
5  0.0643       1       0     www.rho.com
```

We see that `alpha` has a higher PageRank than `delta` or `sigma`, even though they all have the same number of in-links. A random surfer will visit `alpha` over 32% of the time and `rho` only about 6% of the time.

5.3. PageRank for graphs from the Stanford Network Dataset Collection (SNAP)

Since, a vertex will be connected to only few other vertices, most of the entries in the weighted adjaceny matrix will be zero. The resulting matrix will be very sparse. Also, a sizeable webgraph from the Stanford Large Network Dataset Collection (SNAP)) will have tens of thousands of nodes. Hence we will need a technique which can efficiently store these large sparse matrices without running out of memory. A common way for storing large sparse matrices is either

(i) the compressed column storage (CCS) or (ii) the compressed row storage scheme (CRS). We will provide both versions of the project to you.
The CCS storage technique can be used in our HPC implementation and it consists of creation of three arrays:

- G – Holds all non zero entries of the connectivity gragh G.
- indices - Contains the corresponding row indices of the elements in G.
- colptr - Contains the indices of indices array of those elements which start a new column in G.

The CRS storage technique can be used as well in our HPC implementation and it consists of creation of three arrays:

- G – Holds all non zero entries of the connectivity gragh G.
- indices - Contains the corresponding column indices of the elements in G.
- rowptr - Contains the indices of indices array of those elements which start a new row in G.

The Moodle and git repository contains the C codes `pagerank.c` (matrix is stored in CCS format) or other version of `pagerank_csr.c` (matrix is stored in CCS format), which both is a serial implementations of pagerank computation for datasets from SNAP. Here is an example on how to run it using the SNAP matrix soc-LiveJournal1 that is stored in MatrixMarket format and comes from an online social network application:

```
[user@eu-login]$ module load new gcc/6.3.0
[user@eu-login]$ cd pagerank
[user@eu-login]$ wget https://sparse.tamu.edu/MM/SNAP/soc-LiveJournal1.tar.gz
[user@eu-login]$ gunzip soc-LiveJournal1.tar.gz
[user@eu-login]$ tar -xvf soc-LiveJournal1.tar
[user@eu-login]$ make
[user@eu-login]$ bsub -n 1 -W 00:10 ./pagerank soc-LiveJournal1/soc-LiveJournal1.mtx
```

This results in

```
[HPC for CSE] Number of nodes 4847571 and edges 68993773 in webgraph soc-LiveJournal1.mtx
[HPC for CSE] 21 PageRank iterations with norm of 9.6e-07 computed in 8.6 sec.
```

and the pagerank vector is available in the file `PageRank.dat`.

5.4. What experiments to do

- To familiarize yourself with the Power method you may read the chapter 8 from the book ”A First Course on Numerical Methods” by C. Greif and U. Ascher.
- Study the Power method, the Matlab template `pagerank.m`

  ```
  G = p*G*D
  z = ((1-p)*(c~=0) + (c==0))/n;
  while termination_test
  x = G*x + e*(z*x)
  end
  ```

  and the C code `pagerank.c` or the `pagerank_csr.c` which implement the Power method in C using an appropriate test for terminating.
- We have provided this serial implementation. It will be your task to make the necessary MPI changes to obtain a parallel implementation to achieve favorable performance across a range of Euler cluster nodes. We need to be able to build and execute your implementations for you to receive credit. Spell out in your report what Makefile targets we are to build for the different parts of your report. Here are some items you need to add to your report for the parallel experiments on three SNAP graphs:

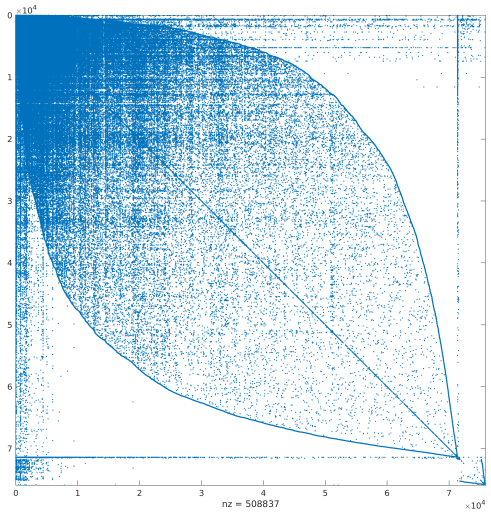

Figure 7: The soc-Epinions1 graph.

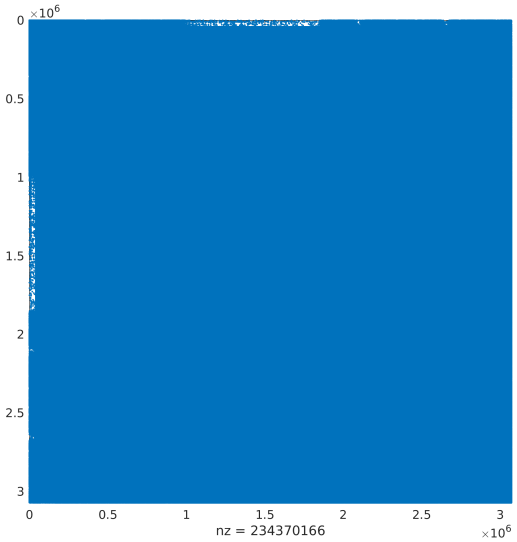

Figure 8: The com-Orkut graph.

  - A description of the communication you used in the distributed memory implementation.
  - A description of the design choices that you tried and how did they affect the performance.
  - Speedup plots that show how closely your MPI code approaches the idealized p-times speedup and a discussion on whether it is possible to do better.
  - Where does the time go? Consider breaking down the runtime into computation time, synchronization time and/or communication time. How do they scale with p? A discussion on using MPI will be useful as well.

- Benchmark your code on the three SNAP example below. Please make sure that you download the SNAP examples either directly to your `$SCRATCH` directory (and not to your own local `$HOME` directory). As an alternative, all these three examples are also available in `/cluster/scratch/oschenk/`.
  - `soc-LiveJournal1.mtx` will result using one compute core in

```
[user@eu-login]$ bsub -n 1 -W 00:10 -R "rusage[mem=4GB]" ./pagerank \
                 /cluster/scratch/oschenk/soc-LiveJournal1.mtx
[HPC for CSE] Number of nodes 4847571 and edges 68993773 in soc-LiveJournal1.mtx
[HPC for CSE] 21 PageRank iterations with norm of 9.6e-07 computed in 6.4 sec.
```

  - `com-Friendster.mtx` will result using one compute core in

```
[user@eu-login]$ bsub -n 1 -W 00:20 -R "rusage[mem=40GB]" \
                 ./pagerank /cluster/scratch/oschenk/com-Friendster.mtx
[HPC for CSE] Number of nodes 65608366 and edges 1806067135 in com-Friendster.mtx
[HPC for CSE] 21 PageRank iterations with norm of 7.8-07 computed in 704.5 sec.
```

    and

  - `com-Orkut.mtx` will result using one compute core in

```
[user@eu-login]$ bsub -n 1 -W 00:10 -R "rusage[mem=4GB]" \
                 ./pagerank /cluster/scratch/oschenk/com-Orkut.mtx
[HPC for CSE] Number of nodes 3072441 and edges 117185083 in webgraph com-Orkut.mtx
[HPC for CSE] 16 PageRank iterations with norm of 6.6e-07 computed in 18.6 sec.
```

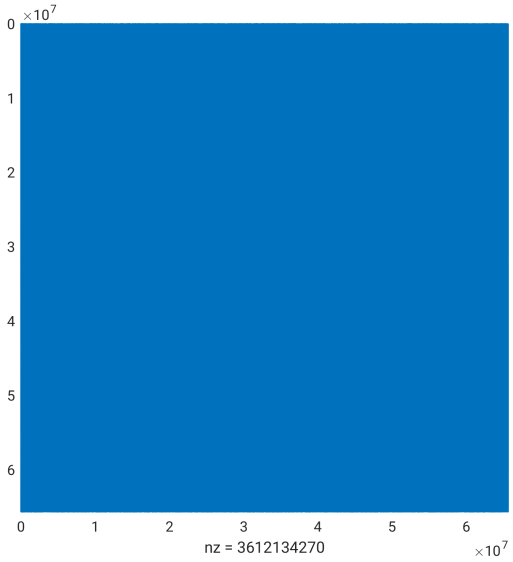

Figure 9: The com-Friendster graph.

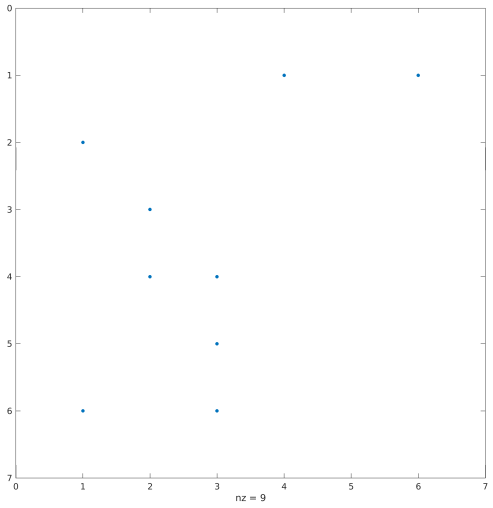

Figure 10: The tiny webgraph matrix.

and report the node number, pagerank, in-degree and out-degree of the five node with the highest pagerank value similar to the table below from the tiny graph from our example above.

```
node  page-rank   in      out
1     0.3210       2       2
6     0.2007       2       1
2     0.1705       1       2
4     0.1368       2       1
3     0.1066       1       3
5     0.0643       1       0
```

Since some of the simulations require a substantial amount of memory, be careful to request enough memory for you jobs to run. This is done above with the option

- `-R "rusage[mem=XXGB]"` ,

which requests `XX` GB *per process* . Keep the latter fact in mind when running in parallel. For example when the total amount of memory needed is 40 GB and you run on 8 processes, then the memory request per process should be 5 GB:

```
[user@eu-login]$ bsub -n 8 -W 00:20 -R "rusage[mem=5GB]" \
                 mpirun ./pagerank /cluster/scratch/oschenk/com-Friendster.mtx
```

Please also note that the actual wall-clock run time of the jobs may be slithly longer due to the sizable I/O.

5.5. MATLAB Visualization of the SNAP Graph Networks

Graphs model the connections in a network and are widely applicable to a variety of physical, biological, and information systems. You can use graphs to model the neurons in a brain, the flight patterns of an airline, and much more. The structure of a graph is comprised of ”nodes” and ”edges”. Each node represents an entity, and each edge represents a connection between two nodes. In our case above we studied the SNAP graphs. You can visualize the connectivity of these SNAP networks using the following commands in MATLAB and the file mmread.m:

```
wget https://math.nist.gov/MatrixMarket/mmio/matlab/mmread.m
matlab -nodesktop
```

```
>> [A,rows,cols,entries,rep,field,symm] = mmread('soc-LiveJournal1.mtx');
>> spy(A);
>> [A,rows,cols,entries,rep,field,symm] = mmread('com-Orkut.mtx');
>> spy(A);
>> [A,rows,cols,entries,rep,field,symm] = mmread('com-Friendster.mtx);
>> spy(A);
```

The visulalization is shown in the Figures 7 to 10. You might take into consideration the irregular structure of these SNAP graphs when parallelizing the pagerank computation using MPI.

6. Task: Quality of the Report [15 Points]

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

Additional notes and submission details

We only accept submissions using our Latex template and C/C++ code. Please submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and observations for all exercises by writing an extended Latex report. Use the Latex template provided on the webpage and upload the Latex summary as a PDF to Moodle .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your MPI solutions;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .zip/.tgz through Moodle .