CS342 Operating Systems – Spring 2020
Project 1: Processes, IPC, and Threads

Assigned: Feb 21, 2020 Document Version: 1.1
Due date: Mar 7, 2020. Time: 23:55.

You will do this project individually. You have to program in C and Linux. We will do our tests in the following distribution of Linux: **Ubuntu 18 LTS – 64 bit**.

Part A. Processes (Mutrix-Vector Multiplication)

In this project, you will develop a multi-process application that will perform matrix-vector multiplication for large matrices and vectors. More precisely, the application will multiply an *n*x*n* matrix **M** with a vector **v** of size *n*. For matrix **M**, the element in row *i* and column *j* is denoted as m_{ij}. For vector **v**, *j*th element is denoted as v_j. Then the matrix-vector product is the vector **x** of length *n*, whose *i*th element x_i is given by:

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

The matrix **M** information is stored in an input text file (ascii). The vector **v** information is also stored in an input text file (ascii). We assume that the row-column coordinates of each matrix element is also is stored in the file. Hence for each non-zero matrix value we store a triple (i, j, m_{ij}) in a line of the *matrixfile*. The line format is:

<rownumber> <columnnumber> <value>

For exampe, for row 3, column 13, and the corresponding value 2, the line will be:

3 13 2

We assume the position of element v_j in the vector **v** is also stored together with the value as a 2-tuple (j, v_j) in a line of the *vectorfile*. The line format will be:

<rownumber> <value>

For example, if the 5th element of vector v has value 10, we will store:

5 10

For vector **v**, information about zero-valued entries will be stored in the input file as well. But for matrix, we do not store information about zero-valued entries in the input file.

You will develop a multi-process program called **mv** that will do the matrix-vector multiplication in the following way (for educational purposes). It will take the following parameters.

mv *matrixfile vectorfile resultfile K*

The main process will read the *matrixfile* (which can be quite large) and will partition it into *K* splits (each split will be a file). The partitioning will be very simple. Assume there are *L* values (lines) in *matrixfile*. Then the first split will contain the first s = ceiling(L/K) values from the *matrixfile*, the next split will contain the next s values, and so on. The last split may contain less than s, which is fine. The number of values *L* in the file can be obtained by first reading the file from beginning to end in the main process before generating the splits (yes this is not efficient, but for this project it is fine).

After generating the split files, the main process will create *K* child processes to process (map) the split files. These child processes will be called as mapper processes (mappers). Each mapper process will process another split file. Mapper processes will run and process their splits concurrently.

The processing of a split file in a mapper process will be as follows. The mapper process will first read the whole *vectorfile* and put the vector values into an array in main memory. If the vector is of size *n*, the array size will be *n*. Then the mapper will start reading its split file one value at a time (one line at a time). The value will be multiplied by the corresponding element in vector **v** and the result will be added into the corresponding entry in a (partial) result array. If, for example, the triple read from a line is (i, j, m_{ij}), then m_{ij} will be multiplied by v_j and the multiplication result will be added to the value at entry *i* of the partial result array. Each line of the split file will be read and processed as described. In this way, the mapper will create a partial result for the multiplication. After mapper has finished with processing its split file, it will write out the partial result array into an intermediate output file, one result per line in the following format:

<rownumber> <value>

When all mappers finish, a reducer process (another child) will be started by the main process. The reducer process will open and read and process the intermediate files produced by the mappers. It will read the intermediate files and will sum up the respective values corresponding to the same vector index. At the end, the result vector will be obtained (after processing all input files). The result vector will be printed out to the *resultfile* in sorted order with respect to row numbers. The line format for the *resultfile* will be:

<rownumber> <value>

Let us consider an example. Assume we have a 6x6 matix and a vector of size 6, as follows.

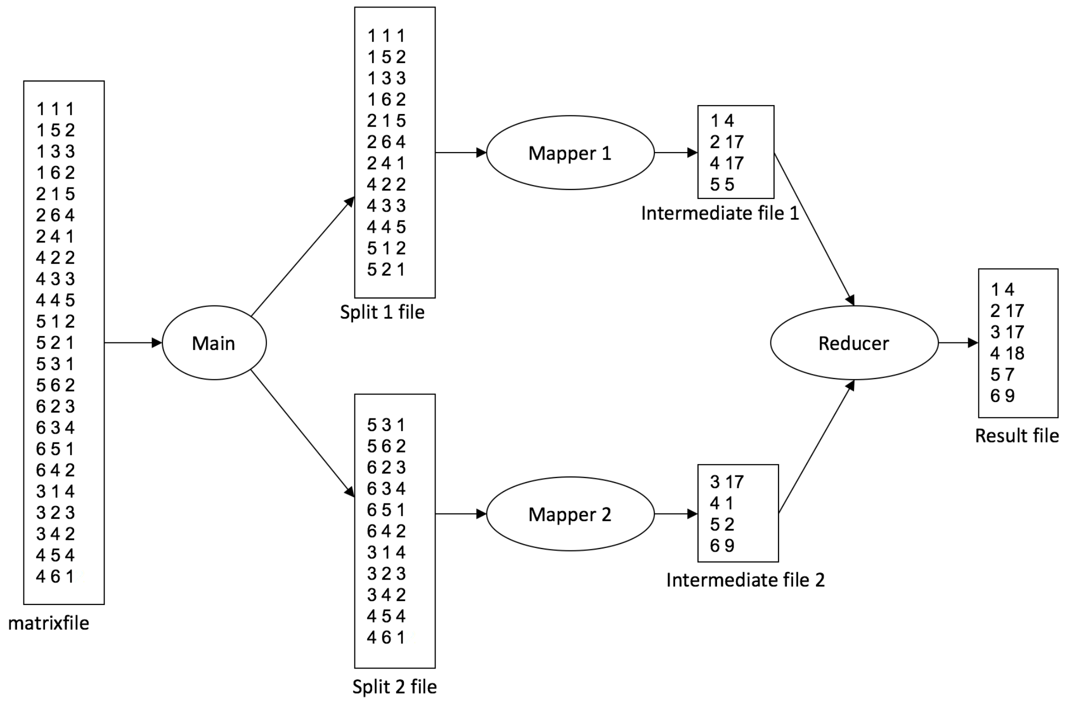
Matrix **M**

	1	2	3	4	5	6
1	1	0	3	0	2	2
2	5	0	0	1	0	4
3	4	3	0	2	0	0
4	0	2	3	5	4	1
5	2	1	1	0	0	2
6	0	3	4	2	1	0

Vector **v**

1	2
2	1
3	0
4	3
5	0
6	1

Assume there are 2 mappers. The first mapper will read the first 12 lines and the second mapper will read the remaining 11 lines. The figure below shows the results (intermediate files) produced by mapper 1 and mapper 2. Reducer will read the intermediate files and will aggregate the values corresponding to the same row numbers, and finally will generate the *resultfile* as shown below. In example below, (4, 17) from intermediate file 1 and (4, 1) from intermediate file 2 are aggregated to be (4, 18). The figure below is also an example for the content format of the files.



Part B. IPC (with pipes)

Implement the same program this time using pipes instread of use of intermediate files between mapper processes and reducer process. The name of the program will be **mvp**. If there are *K* mappers, then there will *K* pipes created. Each pipe will be used for sending data from a mapper process to the reducer process. Each mapper will put data in a different pipe. Reducer process will get data from all pipes.

Part C. Threads

Implement the same program using threads this time. The name of the program will be **mvt**. There will be *K* mapper threads and 1 reducer thread created. Global variables (structures like arrays or linked lists) will be used to pass information between mapper threads and the reducer thread.

Part D: Experiments

Do some timing experiments to measure the running time (turnaround time) of the 3 programs for various values of inputs. Try to draw some conclusion.

Submission

Put all your files into a project directory named with your ID (one of the IDs of team members), tar the directory (using **tar xvf**), zip it (using **gzip**) and upload it to Moodle. For example, a student with ID 20140013 will create a directory named 20140013, will put the files there, tar and gzip the directory and upload the file. The uploaded file will be 20140013.tar.gz. Include a **README.txt** file as part of your upload. It will have the names and IDs of group members, at least. Include also a

Makefile to compile your programs. We want to type just **make** and obtain the executables. Do not forget to put your report (PDF form) into your project directory.

Additional Information and Clarifications

- Maximum value of *K* is 10.
- Maximum value of *n* is 10000.
- The matrix information in the matrix file does not have to be order. For example, the information for row 2 can be earlier then for row 1, or the information for an entry (x, 3) can be earlier than an entry (x, 2).
- The project is to teach processes, IPC, and threads. We would normally not write this program the way descibed for a single compute node. For cluster computing, however, multiple processes running in different nodes (computers of the cluster) can be used to do fast matrix multiplication for very large sparse matrices (*n* in the order of 10^9 or 10^12).
- Startup skelaton is provided in github. You can download and use it as the starting point. https://github.com/korpeoglu/cs342spring2020-p1