CSCI B534 2011

Part 2 of Project 1: MPI PageRank

1. 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.

2. 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;
}
```

3. 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 */
```

4. Compile guide

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

CSCI B534 2011

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
```

5. Deliverables (Due Feb. 7)

You are required to turn in following items 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 that contains:
   a. The description about the main steps or program flow in your program.
   b. The output file which contains the top 10 ranking url numbers.