Dijkstra’s Algorithm (MPI & OpenMP/OpenACC)

Introduction

The purpose of this assignment is to improve your familiarity with MPI and OpenMP/ OpenACC by implementing Dijkstra’s algorithm.

Dijkstra’s algorithm

You need to write two programs, ***dijkstra_mpi*** and ***dijkstra_omp/dijkstra_acc***, that will take as input a single file (an adjacency matrix) and a number of random input source vertex IDs to test and output to a file with the weights of a shortest path from the source vertex 0 to every other vertex. As the names suggest, the first program will use MPI and the second OpenMP or OpenACC for parallelizing the algorithm. The input file will consist of n+1 lines, with the first line containing only the number n, and each other line containing n values (i.e., a dense matrix), with each value representing the weight of the edge. You should assume that the values in the input file are represented as quad-precision floats. An edge that does not exist in the graph is represented by the value `inf`. Your output file should also follow this convention.

For example, an adjacency matrix for a directed graph with five vertices and nine edges would be stored in a file as,

```
5
inf   0.2   inf   3.7   0.4
2.1   inf   3.9   1     inf
2     0.01  inf   0.7   6
9     1.2   2.5   inf   inf
3     3     7     4     inf
```

Parallelization strategy

Your code should **parallelize the search algorithm**, not simply execute different searches using each processor/thread. Feel free to try any parallelization strategy you wish, including strategies we discussed in class. Moreover, the input graph should be split and distributed to processes in the MPI version. The code should work for very large graphs that do not fit in the memory of a single node as long as they fit in the aggregate memory of the set of nodes tasked to execute the program (i.e., you cannot assume that the root node can read the whole graph in memory; it should read one section of it at a time and send it to its destination). While the input data is stored as a dense matrix, consider reading it into a sparse matrix data structure and modifying the algorithm to work with sparse matrices. Extra credit will be awarded for a sparse matrix solution (graph communication should also be done as a sparse structure).

Output

The output file consists of the shortest path weight vector written to a text file, one value per line. In other words, the kth line in the file has the weight of the shortest path from the source to the kth node in the graph. The weight from the source to itself should be marked as `inf`.

Baseline code

Serial baseline code in C/C++ is provided. A Makefile is provided with the code. Execute 'make dijkstra' to compile the code, and 'make clean' to remove the executable and artifacts. Note that the Makefile already has code necessary to compile your dijkstra_mpi and dijkstra_omp files, assuming your source files are called dijkstra_mpi.cpp and dijkstra_omp.cpp. If not, edit the appropriate variables in the Makefile with the name of your sources file(s).

Testing

Test graphs can be generated by executing the Python script make_data.py. The only Python library that the script requires is Numpy and the script should run in most Python 3.x environments. The largest file, *20000.graph*, contains 400 million numbers, representing a graph with 20,000 nodes, and takes up 1.5G of space. For each input graph, you can run the provided serial program to get a reference solution. You can then use the 'diff' Linux program to see if your solution is equivalent with the given solution for a given problem size.

The test graphs have already been generated on the HPCs and can be found at `/WAVE/projects/COEN-145-Fa22/data/pr3` on the WAVE HPC. **DO NOT COPY** the graph files to your own directories. You can either call your programs with a full path to the graph files, e.g.,

```
<your_project_dir>/dijkstra /WAVE/projects/COEN-145-Fa22/data/pr3/10000.graph 100
```

or you can create a symbolic link to the data within your project file, like this:

```
cd <your project directory>
for f in 1000 5000 10000 20000; do
   ln -s /WAVE/projects/COEN-145-Fa22/data/pr3/${f}.graph;
done
```

Note that the evaluation may be done using a different set of input files than the ones in the test directory. As such, do not over-specify your code to work perfectly on those files and less so on others not in that set.

What you need to turn in

1. The source code of your program.
2. A short report that includes the following:
   a. A short description of how you went about parallelizing dijkstra's algorithm. One thing that you should be sure to include is how you decided what work each process would be responsible for doing.

b. Timing results for your parallel execution on 1 node using 1, 2, 4, 8, 16, and 28 processes (for MPI), and 1, 2, 4, 8, 16, and 28 threads in the case of the OpenMP program, for the search step (i.e., not including time for I/O). For each experiment, timing should be obtained using 100 randomly-selected sources. These results should be reported in a table (in a text file, not in the report) as well as a speedup graph. The OpenACC version should use only 1 GPU. The following is an explicit list of the problems to time for each of these process counts:

```
1000.graph
5000.graph
10000.graph
20000.graph
```

c. A brief analysis of your results. Some things to consider might be:
   i. How does the number of processes/threads affect the runtime? Why do you think that is?
   ii. How does the size of the problem affect the runtime? Why do you think that is?
   iii. How well does your program scale, i.e., if you keep increasing the number of processes, do you think the performance will keep increasing at the same rate?

Do NOT include the test input files in your submission. They take up almost 2GB of space. **You will lose points for including any of the test files.**

Submission specifications

- A makefile must be provided to compile and generate the executable file.
- The executable files should be named 'dijkstra_mpi' and 'dijkstra_omp'.
- Your programs should take as arguments the input file name, source ID (0-based index), and the output file name. An optional number of threads parameter may also be specified. For example the program would be invoked as follows,

```
mpirun -n 28 ./dijkstra_mpi 10000.graph 100 test.out
[or]
./dijkstra_omp 10000.graph 100 test.out -t 28
```

  where 10000.graph is the input graph, 100 is the number of random source nodes to test, test.out is the optional output file, and 28 is the number of threads.

- Your program MUST print to standard out the timing information in the same format as that provided in the baseline code function print_time in dijkstra.cpp.
- All files (code + report) MUST be in a single directory and the directory's name MUST be your university student ID. Your submission directory MUST include at least the following files (other auxiliary files may also be included):

```
<Student ID>/dijkstra_mpi.cpp
<Student ID>/dijkstra_omp.cpp
<Student ID>/Makefile
```

```
<Student ID>/times_omp.txt
<Student ID>/times_mpi.txt
<Student ID>/report.pdf
```

- Submission MUST be a tar.gz archive.
- The following sequence of commands should work on your submission file:

```
tar xzvf <Student ID>.tar.gz
cd <Student ID>
make
ls -ld dijkstra_mpi
ls -ld dijkstra_omp
```

This ensures that your submission is packaged correctly, your directory is named correctly, your makefile works correctly, and your output executable files are named correctly. If any of these does not work, modify it so that you do not lose points. I can answer questions about correctly formatting your submission BEFORE the assignment is due. Do not expect questions to be answered the night it is due.

Evaluation criteria

The goal for this assignment is for you to become more familiar with the OpenMP and MPI APIs and develop efficient parallel programs. As such, the following things will be evaluated:

1. follows the assignment directions,
2. solve the problem correctly,
3. do so in parallel,
4. achieve speedup:
   - 5 / 10 points will be reserved for this criterion.
   - points will be awarded by comparing the relative performance of your solution to the solution of the benchmark program provided.
   - The speedups obtained will probably depend on the size of the input file. It is not expected that you get good speedups for small files, but you should be able to get good speedups for large files.