CS542200 Parallel Programming
Homework 3: Fully Distributed Shortest Path Algorithms

Revision 10

Due: Sun December 2, 23:59PM, 2018

1 Goal

This assignment helps you learning the differences between **synchronous** and **asynchronous** computing by implementing **fully distributed parallel algorithms for shortest path problem.** In this assignment, you need to implement the two fully distributed shortest path algorithms and a graph partition algorithm:

1. **Moore's asynchronous single source shortest path algorithm** using MPI+Pthread.
2. **Floyd-Warshall's synchronous all pair shortest path algorithms** using MPI+OpenMP.
3. **Graph partition algorithm for performance optimization**

2 Problem Description

2.1 Moore's Single Source Shortest Path Algorithm

- In this version, you are asked to implement the Moore's single source shortest path algorithm explained in the lecture slides of **Chap5 p36**.
- Your implementation must use **MPI** and **Pthread** to parallelize the algorithm across multiple compute nodes. **Only one MPI process is allowed to be launched on a compute node**, so you must use threads to utilize all the CPU cores on a node.
- **Your implementation must be fully distributed** which means the **graph topology must be partitioned and stored independently** on nodes, **no master process or single task queue is allowed** for centralized task dispatching.

- Each thread is allowed to handle the computations of a **group of vertices**. In other words, it is not necessary to create one thread per vertices. However, **only Moore's algorithm is allowed to be used** throughout the whole computations even for the vertices within the same group.
- **Collective MPI call is not allowed except for file input/output.**
- **You may implement any asynchronous termination detection algorithm for ensuring correct result.**
- ~~**Single ring** termination detection is recommended because dual ring algorithm may cause too much performance overhead, so it is not required in this assignment.~~

2.2 Floyd-Warshall's All Pair Shortest Path Algorithm

- In this version, you are asked to implement the **Floyd-Warshall's all pair shortest path algorithm** explained in the lecture slides of **Chap7 p38**.
- Your implementation must use **MPI** and **OpenMP** to parallelize the algorithm across multiple compute nodes. **Only one MPI process is allowed to be launched on a compute node. All computations should be done by the threads created by OpenMP.**
- **MPI processes are only for communications** among compute nodes. There is **no restriction to the communication methods** among MPI processes.

2.3 Graph Partitions

- Implement a **graph partition algorithm** for improving your performance.
- Hint: a better graph partition strategy can **minimize communication overhead and balance load** (i.e., the number of vertices per process can be different).
- **The graph partition time will not be counted into the execution time.**

3 Input/Output

3.1 Input Parameters

Your programs for 2.1 and 2.2 should accept 2 or 3 parameters.
Command line arguments with 2 parameters:

```
$ ./executable ${in} ${out}
```

Command line arguments with 3 parameters:
(You only need to support this format if you implement 2.3)

```
$ ./executable ${in} ${out} ${par}
```

- ${in}: the input graph file name [string]
- ${out}: the output distance file name [string]
- ${par}: the partition file name [string]

For 2.3, you need to implement an additional partition program to pre-partition your graph:

```
$ ./partition ${in} ${par} ${#processes}
```

- ${in}: the input file name [string]
- ${par}: the output partition file name [string]
- ${#processes}: the number of processes [integer]

3.2 INPUT FILE FORMAT

➢ The input is a binary file containing a **connected directed graph with non-negative edge weight**.

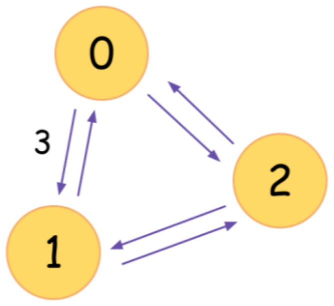
Here is an example:

[offset]	[type]	[decimal value]	[description]
0000	32 bit integer	3	# vertices
0004	32 bit integer	6	# edges
0008	32 bit integer	0	Src id for edge 0
0012	32 bit integer	1	Dst id for edge 0
0016	32 bit integer	3	Weight on edge 0
0020	32 bit integer		Src id for edge 1
…	…	…	…
0076	32 bit integer		Weight on edge 5

➢ The first two integers mean {number of vertices} {number of edges}

➢ After vertices and edges is a list of {source vertex id} {destination vertex id} {edge weight}

➢ The edge weights are non-negative integers. **The values of vertex id start from 0**.



Notice that:

✧ **$1 \leq V \leq 2000000000$**

✧ **$V \leq E \leq 2000000000$**

✧ **$0 \leq i, j \leq V$**

✧ **$0 \leq W \leq 10000$**

✧ **All vertices are connected**

✧ **No need to consider the condition of signed integer overflow**

3.3 PARTITION FILE FORMAT

- Given a graph with N vertices, your partition program should output a partition file that contains N lines, each line containing an integer which represented the id of each vertex's group
- The number of line is equal to vertex id
- Here is an example

Vertex 0

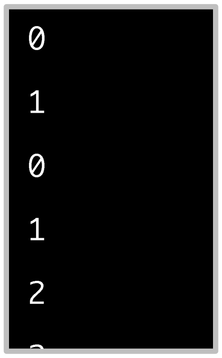
Vertex 1

Vertex 2

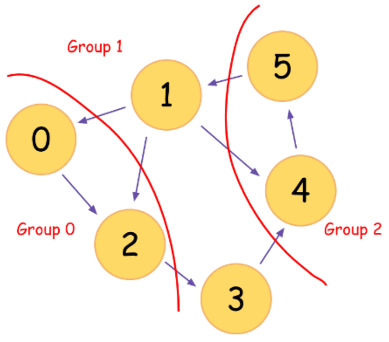
.

.

.



```
0
1
0
1
2
```



[Output file from partition program]

3.4 OUTPUT FORMAT

- The output file contains the shortest path distance from your algorithms.
- The output file is also binary format, and the details are listed below.
- For the first problem (Moore's algorithm), you should **output N integers which are the shortest path distances from vertex 0**.

[offset]	[type]	[decimal value]	[description]
0000	32 bit integer	0	*min* Dist(0, 0)
0004	32 bit integer	**Do it yourself**	*min* Dist(0, 1)
0008	32 bit integer	**Do it yourself**	*min* Dist(0, 2)
…	…	…	…
xxxx	32 bit integer	**Do it yourself**	*min* Dist(0, N-1)

- For the second problem (Floyd-Warshall's algorithm), you should **output N^2 integers which are the shortest path distances between each pair of vertices.**

[offset]	[type]	[decimal value]	[description]
0000	32 bit integer	0	*min* Dist(0, 0)
0004	32 bit integer	**Do it yourself**	*min* Dist(0, 1)
0008	32 bit integer	**Do it yourself**	*min* Dist(0, 2)
...	...	...	...
xxxx-4	32 bit integer	**Do it yourself**	*min* Dist(N-1, N-2)
xxxx	32 bit integer	0	*min* Dist(N-1, N-1)

- **The output records must be sorted by vertex id**
- **Distance (i, j) = 0, where $i = j$;**

4 GRADING

This homework will be graded by the correctness, report, demonstration, and performance as described below:

4.1 CORRECTNESS (40%)

TAs will use different graph to test your program, and check whether your **output** can pass our judge script.

The detail score distribution of each part is:

- ✓ Moore's single source shortest path algorithm (20%)
- ✓ Floyd-Warshall's all pair shortest path algorithm (20%)

4.2 REPORT (25%)

A. Title, name, student ID

B. Implementation

Explain the **details** of your implementations in diagrams, figures, sentences. You also need to answer the questions below:

- ✓ Why graph partition can have significant performance impact to your implementation? (You still need to discuss this question, even if you don't implement any optimized graph partition strategy.)
- ✓ What are the pros and cons of synchronous and asynchronous versions?
- ✓ Other efforts you've made in your program

C. Experiment & Analysis

Explain how and why you do these experiments? Explain how you collect those measurements? Show the results of your experiments in plots, and explain your observations.

i. Methodology:

- ✓ **System Spec (**If you run your experiments on your own machine**)**
  Please specify the system spec by providing the CPU, RAM, disk and network (Ethernet / Infiniband) information of the system.

- ✓ Performance Metrics: How do you measure computing time of your program? How do you compute the values in the plots?

ii. Plots:

(a). **Strong scalability** chart **for both of your implementations**: Conduct strong scalability experiments, and plot the speedup. The plot must contain at least 4 settings (e.g., scales), and include both single node and multi-node environment.

(b). **Load balancing** chart for your **Moore's algorithm implementation**: Evaluate the workload distribution by plotting the number of computing tasks (i.e., vertex update operations) per process. This only needs to be reported from one chosen scale setting.

Both scalability and load balancing experiments **must be conducted on the 3 evaluation test datasets released by the TA**.

- ➢ rand_graph: a graph whose edges are randomly generated between vertices.
- ➢ dense_graph: a graph with higher #edge/#vertex ratio.
- ➢ skew_graph: a graph whose number of edges per vertex is generated by a powerlaw instead of a uniform probability distribution.

Based on your experimental results, discuss the **performance comparison between asynchronous algorithm (Moore's algo) and synchronous algorithm (Floyd-Warshall)**.

(c). Conduct experiments to explain why your graph partition algorithm can improve performance.

(d). Any other experiments you think meaningful to compare and analyze the differences between implementation versions.

D. Experience and conclusion

- ✓ What have you learned and observed from this assignment?

- ✓ What difficulty did you encounter when implementing this assignment?
- ✓ If you have any feedback, please write it here.

4.3 DEMO (15%)

- ✓ Test the correctness of your codes.
- ✓ Go through your codes. Make sure your implementation follows the requirements.
- ✓ We will ask some questions about your codes and report.

4.4 PERFORMANCE (20%)

- ✓ Floyd-Warshall's all pair shortest path algorithm with graph partition

5 Submission

Upload these files to `apollo31`, and place them under `~/homework/hw3/`:

- `sssp.c`
- `apsp.c`
- `partition.c`
- `Makefile`

Upload this file to iLMS and do not compress these into a compressed file:

- `report.pdf`
- `sssp.c`
- `apsp.c`
- `partition.c`

Note:

1. You can also write your code in C++ by submitting `*.cc` files.
2. Your Makefile must be able to build the corresponding targets of the implementations: `sssp`, `apsp`, `partition`. If you're unsure how to write a Makefile, you can use the provided example Makefile as-is. (or ask for assistance)

3. Your submission time for your source code will be based on the time on `apollo31` and your submission time for your report will be based on iLMS.
4. Late submission policy:

score *= 1		Before deadline
score *= 0.95	After deadline	Before deadline+3days
score *= 0.9	After deadline+3days	Before deadline+7days
score *= 0	After deadline+7days	

6 Reminder

1. Refer to iLMS for the location of the Makefile and test cases.
2. Since we have limited resources, please start your SSSP and APSP ASAP. Do not leave it until the last day!
3. 0 will be given to cheaters (even copying code from the Internet), but discussion on code is encouraged.
4. Asking questions through iLMS is welcomed!