CMSC5702 Advanced Topics in Parallel/Distributed Systems

Assignment Two: The All-pairs Shortest Path Problem

Objectives

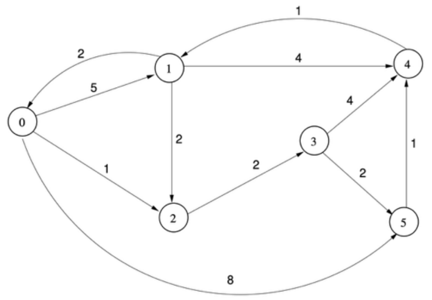
The objectives of this assignment are as follows:

(1) To provide more exposure to advanced topics in MPI programming, including point-to-point or collective communications in a 2D grid topology.
(2) To reinforce concepts of performance analysis, including scalability and isoefficiency, through hands-on experience with benchmarking.
(3) To implement a parallel algorithm using CUDA on a GPGPU.

Introduction

The *all-pairs shortest path (APSP) problem* is to find the length of the shortest path for all pairs of vertices in a weighted graph. APSP belongs to the most fundamental problems in graph theory and it has many practical applications in a broad range of engineering fields like VLSI circuit routing, transportation systems and communications networks.

The *Floyd-Warshall* algorithm, or simply *Floyd's algorithm*, is a classic method to solve an APSP problem. This algorithm evolves the adjacency (weight) matrix of the graph using a triple loop so that every element (*i*, *j*) in the matrix, representing the length of path from vertex *i* to vertex *j*, is minimum (shortest), having all reachable paths from *i* to *j* compared. For a $n \times n$ matrix, the comparison operation is performed n^3 times, so the time complexity of the algorithm is $\Theta(n^3)$. See the following graph as an example. The infinity symbol represents a nonexistent edge. It is usually assigned with a very large value such as 999999 in the matrix. After running the Floyd's algorithm on the adjacency matrix, the matrix will contain the shortest-path lengths between every pair of vertex *i* and vertex *j*.



(a) A weighted directed graph

	0	1	2	3	4	5
0	0	5	1	∞	∞	8
1	2	0	2	∞	4	∞
2	∞	∞	0	2	∞	∞
3	∞	∞	∞	0	4	2
4	∞	1	∞	∞	0	∞
5	∞	∞	∞	∞	1	0

(b) Initial adjacency matrix

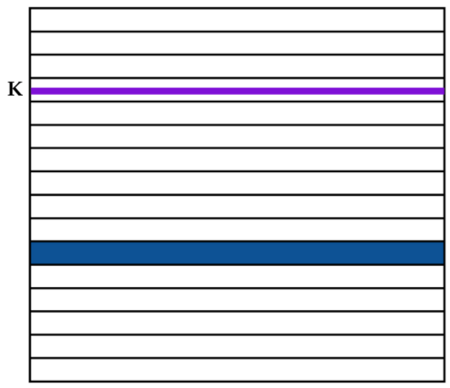
	0	1	2	3	4	5
0	0	5	1	3	6	5
1	2	0	2	4	4	6
2	8	6	0	2	5	4
3	6	4	6	0	3	2
4	3	1	3	5	0	7
5	4	2	4	6	1	0

(c) Shortest-paths matrix

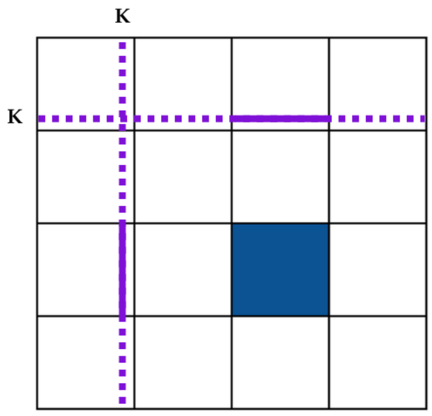
```
Algorithm 1: Floyd's Algorithm
  Input  : $n$ — number of vertices
  Input  : $a[0..n-1][0..n-1]$ — adjacency matrix
  Output: Transformed $a$ that contains the shortest path lengths
  for $k \leftarrow 0$ to $n-1$ do
      for $i \leftarrow 0$ to $n-1$ do
          for $j \leftarrow 0$ to $n-1$ do
              $a[i][j] \leftarrow \min(a[i][j],\ a[i][k] + a[k][j])$;
          end
      end
  end
```

To parallelize the Floyd’s algorithm, there are basically two schemes partitioning the matrix:

(1) Striped Version (row wise)



(2) Checkerboard Version



The row-wise striped version, version (1), is easier to implement while the checkerboard version, version (2), is of better agglomeration and theoretically better performance. Detailed description of both parallel algorithms is documented in Section 3.9 of Foster’s book [2]. Chapter 6 of Quinn’s book [1] has provided implementation of version (1). Since the book is available in hardcopy only, a scanned PDF of the book chapter is provided along with this assignment to help you get started.

Tasks

Part 1 – MPI Programming

In this assignment, you are to write an MPI program implementing the *checkerboard* version of the Floyd’s algorithm, assuming that the number of processors executing the program is a square number.

To facilitate your work, we have provided a starter code file called `floyd_chk_starter.c`. Rename it to `floyd_chk.c`, fill in your name and student ID, and add your code for the missing TODO parts to complete the implementation of the MPI program.

We also provided the sequential version and row-striped parallel version of the Floyd’s algorithm (`floyd_seq.c` and `floyd_row.c`). You can better understand the sequential and parallel algorithms by reading their code. The row-striped MPI program will be useful for your benchmarking and performance comparison in Part 2.

Note: the MPI programs have made use of the helper source files (MyMPI.h, MyMPI.c), which have supplied handy functions, e.g., `read_checkerboard_matrix()` for reading different blocks of a matrix from a file into the memory of the processors in a distributed checkerboard arrangement. The main task required from you is to create a communicator that organizes the processes into a Cartesian or 2D grid virtual topology for broadcasting row/column vectors to the processors needing them. More details about how to use a 2D grid virtual topology in MPI can be found in the last part of Lecture 5 notes (MPI) and Chapter 8 of the Quinn’s book [1]. Please find them on Blackboard.

Part 2 – MPI Benchmarking

Benchmark both the provided striped version and your implemented checkerboard version of the MPI program with combinations below on the HPC cluster:

- Number of processors (*p*): 1, 4, 9, 16, 25, 36
- Problem size (*n*): 60, 480, 1500, 3000, 4020, 6000

Record the execution time of each combination of (program version, *p*, *n*). Then plot the following four charts (optionally using our provided Excel template):

- **Striped version**
  - ***Relative speedup*** curve for each problem size on one chart.
  - ***Efficiency*** curve for each problem size on one chart.
- **Checkerboard version**
  - ***Relative speedup*** curve for each problem size on one chart.
  - ***Efficiency*** curve for each problem size on one chart.

Analyze the results and discuss the scalability of the two parallel programs considering the factor of problem size. You may also illustrate your predicted speedup and efficiency according to some cost models. (Hint: Perform *isoefficiency* analysis on the programs. More details can be found in the paper [3] provided along with this assignment.

Notes:

1. Execution time is taken ***excluding*** the file I/O time (better the standard I/O time as well).
2. Input data files (in binary format) containing weight matrices of different problem size are provided on Blackboard. Download the tarball and extract it.
3. It is normal that the longest execution (sequential run for *n* = 6000), compiled without code optimization, requires about 10-12 minutes on our HPC benchmarking platform.
4. To ease your benchmarking, a shell script `floyd.sh` is provided for creating a job script on-the-fly for a specified combination of program version, *p* and *n* and submitting the job to the required Slurm's HPC job queue.

Part 3 – CUDA Programming and Benchmarking

Based on the provided sequential program (`floyd_seq.c`), implement the Floyd's Algorithm into a CUDA program named `floyd_gpu.cu`. It should spawn CUDA kernels to parallelize the computations of the shortest paths between all pairs of vertices in the adjacent matrix.

To simplify your work, we have provided a starter code file `floyd_gpu_starter.c`. Rename it to `floyd_gpu.c`, fill in your name and student ID, and add your code for the missing TODO parts to complete the implementation of the CUA program.

Similar to the sequential program, it will read a binary input file to load the adjacent matrix, the first two integers in the file represent the number of rows (`m`) and columns (`n`) of the adjacency matrix, and we have assumed that `m` is always equal to n, or else we will abort the execution.

The starter code has helped you allocate a 1-D array named d_a to store the matrix values on the GPU device memory. After the file gets loaded, you need to copy the loaded array from the host side to the device.

Your main task is to complete the CUDA kernel `compute_shortest_paths(dtype *d_a, int n, int k)`. After the kernel calls complete, copy the computed matrix values from the device back to the host, i.e., array a.

You need not perform detailed performance analysis on the CUDA program. However, your written report should include a table to show the CUDA program execution time (one timing is enough) when running on a single GPGPU device (Nvidia Titan V) in our GPU cluster for each of the problem size:

- Problem size (*n*): 60, 480, 1500, 3000, 4020, 6000

Submission

Submit the following files via Blackboard:

(1) `floyd_chk.c`: The MPI C program source implementing the checkerboard version of Floyd's algorithm.
   - Please include sufficient comments to make your program code more readable.
   - Since this time, your program will link with `MyMPI.c`, you are required to write a C program instead of C++.

(2) `floyd_gpu.cu`: The CUDA program source implementing the Floyd's algorithm.

(3) Written report including the performance graphs and analysis.
   - An Excel template is provided to facilitate your graph plotting. It is up to your choice whether you would use it. If you prefer other charting options like using Matplotlib in Python to plot the curves, feel free to go ahead. You need not submit this Excel file to us. The charts produced in whatever ways should be incorporated into your report document.

Note: In both the programs to submit, you may add additional functions to assist your implementation of the required TODO tasks, but please try NOT to modify the signature of the `compute_shortest_paths()` function because we may perform unit testing on it.

References

[1] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw Hill, 2004.

[2] Ian Foster, *Designing and Building Parallel Programs*, Addison Wesley, 1995.

[3] V. Kumar and V. Singh, Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem, *Journal of Parallel and Distributed Computing* (Special Issue on Massively Parallel Computation), Vol 13, #2, 1991, 124-138.