Homework 2

Assignments: All parts are to be solved individually (turned in electronically, no ZIP files or tarballs, written parts in ASCII text, NO Word, postscript, PDF etc. permitted unless explicitly stated). All filenames must be exactly what is requested, capitalization included.

Please use the ARC cluster for this assignment. All programs have to be written in C, translated with mpicc/gcc and turned in with a corresponding Makefile.

1. (40 points) Write your own custom version of MPI using sockets for communication between nodes.

   In this assignment, you will

   - Write your own version of **prun** called **my_prun**.
     - This should be a bash shell script.
     - After reserving multiple nodes using **srun**, this script will connect to your reserved nodes using ssh and run the same code on each.
     - Use **ssh $node $binary** to connect to your reserved nodes (i.e., c26, c104, etc.) and run your code (your binary should be named **my_rtt**).
     - The name of the nodes you have reserved can be found using the environment variable **$SLURM_NODELIST**.

   - Create a custom "message passing" library for MPI function calls using C Socket programming called **my_mpi.c**/**my_mpi.h**.
     - You only need to implement the subset of MPI functions used in HW1 problem 2 (i.e., MPI_Init, MPI_Send, MPI_Recv, etc.).
     - Your MPI functions should be named the same as the existing MPI functions and also take the same parameters (even if some parameters are unused in your function).

   - Test your library using HW1 problem 2 to measure the RTT.
     - The objective is for your custom MPI library to work with the least amount of HW1 problem 2 code changes possible. You should not modify any of the parameters of the MPI function calls.
     - You may **#define** built in MPI datatypes such as **MPI_COMM_WORLD**, **MPI_DOUBLE**, etc. Most of the time when these datatypes are used as parameters, you will not have to use the parameter in your function anyway.
     - You must implement **MPI_Barrier()** (you'll need it in **MPI_Finalize()** anyway).

   If you are having trouble with scripting, take a look at this simple implementation of bash scripting, and ssh to launch programs on several nodes: simple_mpi.tar.

   Hints:

   - Each node you reserve should only have one process (-n equals -N).
   - Each process must communicate their hostname/port number to the other processes to initiate communication over sockets.
   - Before the sockets are opened, the only method of communication between nodes is the ARC filesystem. You should use files to store the hostnames and port numbers so that each node can read from the files and know what port the node it needs to connect to is on.
   - When spawning other nodes, you may add additional parameters to the call of the binary (my_rtt in this case). They could indicate things like the node's rank, the number of nodes being used, the nodelist file, etc.
   - You may pass these additional parameters (argc and argv) from main() to MPI_Init().

   Discussion information:

   - OpenMPI TCP FAQ (lots of gritty details on how OpenMPI does network communication)
   - MPI Performance Topics (general performance characteristics of MPI)
   - Comparing Ethernet and Myrinet for MPI Communication" (an analysis of MPI over different network architectures)

   Run the same setup as HW1 #2. In the README, compare your results with your previous HW1 results.

   ```
   my_prun -n 8 ./my_rtt
   ```

   Turn in **p1.Makefile**, **p1.README**, **my_prun** (a script), **my_mpi.c**/**my_mpi.h** (module containing the subset of MPI functionality required) and **my_rtt.c** (same as in HW1 but referencing my_mpi.h). The binary that **p1.Makefile** generates should be named **my_rtt**.

2. (10 points) Learn how to compile and execute a CUDA program.

   The number 'pi' is a mathematical constant, the ratio of the circumference of a circle to it's diameter. Several methods exist to calculate the value of pi. In this problem we will use the Monte Carlo method to approximate the value of pi to a certain accuracy. For more information about the Monte Carlo method go here

   Write the CUDA code to approximate the value of 'pi' by parralelizing the serial code.

   - Your code does not need to have MPI
   - CUDA only works on compute nodes, not the login node, so be sure to get a compute node first with **srun**.

- In your **p2.Makefile** you may use **-Wno-deprecated-gpu-targets** to suppress a warning about deprecated architectures. Do not worry about this warning, everything is fine.

```
nvcc p2.cu -o p2 -O3 -lm -Wno-deprecated-gpu-targets
```

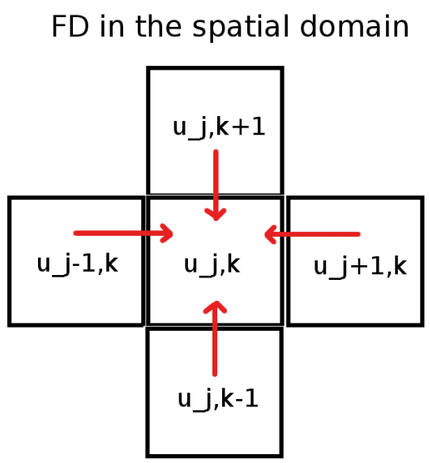
Turn in **p2.cu** (CUDA file), **p2.Makefile** and **p2.README** (Explaining the implementation).

3. (50 points) Group problem (3 per group)

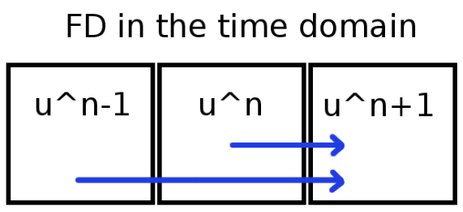
We will extend the methods of the last HW into two dimensions.

Download, extract, compile the code lake.tar

This program models the surface of a lake, where some pebbles have been thrown onto the surface. The program works as follows. In the spatial domain, a centralized finite difference is used to inform a zone of how to update itself using the information from its neighbors



The time domain does something similarly, but here using information from the previous two times



The program runs two versions of the algorithm, a CPU version, and a skeleton GPU version. Your task is to fill in the GPU algorithm to solve the same problem. Instructions:

V0:

- Reserve one node with one processor. (Each node in the ARC cluster has its own nVidia GPU.)

```
srun -N1 -n1 --pty /bin/bash
```

- Run the lake program

```
./lake {npoints} {npebbles} {end_time} {nthreads}
```

npoints defines the grid size (npoints x npoints), **npebbles** is the number of pebbles that are generated in the program, **end_time** is the final time of the simulation, and **nthreads** will be used withe the GPU implementation.

The following runs on a grid of **(128 x 128)**, with **5** pebbles, for **1.0** seconds, using **8** GPU threads (implemented later):

```
./lake 128 5 1.0 8
Running ./lake with (128 x 128) grid, until 1.000000, with 8 threads
CPU took 0.284713 seconds
GPU computation: 0.003168 msec
GPU took 0.327409 seconds
```

- View the output in a heatmap with gnuplot:

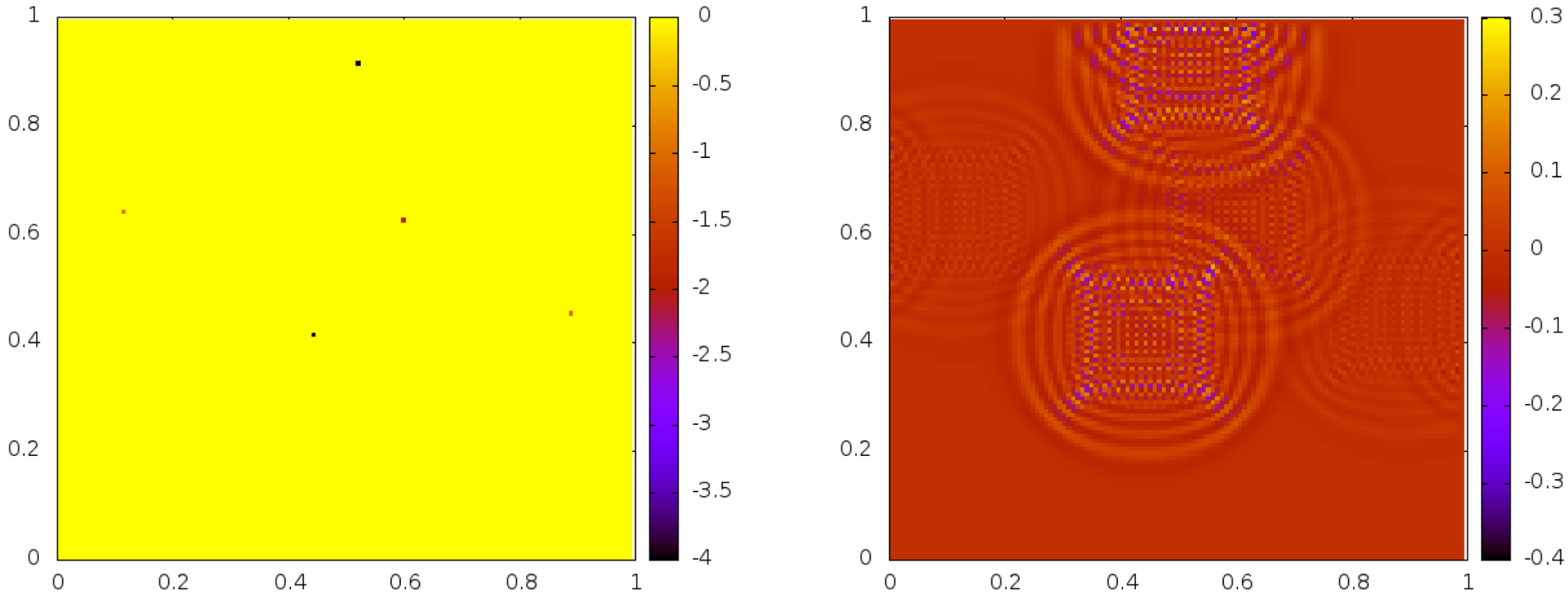
The output files

```
lake_i.dat
lake_f.dat
```

can be converted into a .png image using the gnuplot script **heatmap.gnu**. Run

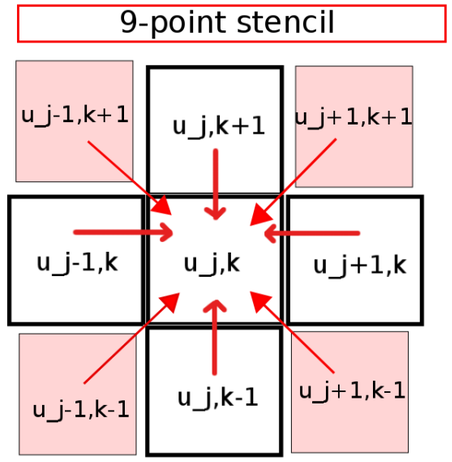
```
gnuplot heatmap.gnu
```

This will create the files **lake_i.png**(the initial configuration), **lake_f.png** (the final configuration) in the directory.



V1:

- So far you used 5-point stencil for discretizing the wave equation (look at the **evolve** method in lake.cu). In this section, you extend that and use 9-point stencil. Under this configuration, 8 more neighbors are involved in the communication:



- The pseudocode for 5-point is as following:

```
un[idx] = 2*uc[idx] - uo[idx] + VSQR *(dt * dt) *(( WEST + EAST + NORTH + SOUTH -
    4 * uc[idx])/(h * h) + f(pebbles[idx],t));
```

- The pseudocode for 9-point is as following:

```
un[idx] = 2*uc[idx] - uo[idx] + VSQR *(dt * dt) *(( WEST + EAST +
                NORTH + SOUTH + 0.25*(NORTHWEST + NORTHEAST + SOUTHWEST + SOUTHEAST)-
                5 * uc[idx])/(h * h) + f(pebbles[idx],t));
```

- You need to calculate the indices for neighbors (NORTHWEST, NORTHEAST, SOUTHWEST, SOUTHEAST, NORTHNORTH, EASTEAST, SOUTHSOUTH, WESTWEST). Note that a lake surface is a grid (2-dimension), and it is represented as a vector (1-dimension) in the code. Make sure the calculated indices do not go outside the bounds of the vector!
- Implement **evolve9pt** method that performs 9-point stencil under CPU.
- Perform experiments and compare and contrast the two versions (5-point vs 9-point). What are the tradeoffs? Note that the 9-point version evolves faster and you need to run it with shorter **end_time** when comparing with 5-point version.
- Turn in plots and a description of your observation.
- Use 9-point stencil for the rest of homework.

V2:

- Fill in the function **run_gpu** in the file **lakegpu.cu** to run the same algorithm as the cpu version, but using CUDA kernels. The grid will be decomposed on the GPU into 2D blocks.

  The program takes as an argument **nthreads**. This will be the number of threads per block used on the GPU. So, for instance, with **nthreads=8**, and a domain of grid points **(npoints=128 x 128)**, you will create **(npoints/nthreads)x(npoints/nthreads) = (16 x 16)** blocks, with **(8 x 8)** threads on each block.

- You will time your CUDA implementation using cudaEventXXX() API. Be sure to start timing **before** the first memcpy to the GPU and stop **after** the last memcpy off of the GPU.
- Compare the CPU/GPU runs for varying grid sizes (16, 32, 64, 128, ..., 1024, etc.)

V4:

- Create an MPI version of your program the further decomposes your grid based on processor rank.
- Use **4** nodes (i.e. 4 GPU's) in your implementation.

  ```
  srun -N4 -n4 --pty /bin/bash
  ```

- Each node should communicate boundary information to the appropriate neighbor, then run the CUDA kernel during a time-step (one iteration of **evolve**).
- Have each node output it's own data file, labeled as

  ```
  lake_f_0.dat //node 0
  lake_f_1.dat //node 1
  ```

```
//etc.
```

Include in **p3.README** a discussion of your results. Your discussion should include answering the following questions:

- How well does your algorithm scale on the GPU? Do you find cases (grid size, thread number, etc.) where the GPU implementation does not scale well? Why?
- In the serial code, compare your CPU and GPU runtimes for different grid sizes. When is the GPU better, and when is it worse?
- Integrating CUDA and MPI involves more sophisticated code. What problems did you encounter? How did you fix them?

Hints:

- Here is a sample **p3.Makefile** for both V2 and V4.
- If you are interested in the particular math behind this algorithm, here is a good introduction. In particular, we are solving the 2D wave equation with sources using finite differencing.
- The code stores the initial configuratons of **u^0** and **u^1** in the variables

  ```
  double *u_i0
  ```

  ```
  double *u_i1
  ```

  These are passed to both the **run_cpu** and **run_gpu** routines, both the routines should produce the same results.

Turn in **p3.README**, **lake.cu/lakegpu.cu** (version V1, V2), **lake_mpi.cu/lakegpu_mpi.cu** (version V4), **p3.Makefile**, **lake5pt.png** and **lake9pt.png**

4. Peer evaluation: Each group member must submit a **peer.txt** evaluation form.

What to turn in for programming assignments:

- commented program(s) as source code, comments count 15% of the points (see class policy on guidelines on comments)
- Makefiles (if required)
- test programs as source (and input files, if required)
- README (documentation to outline solution and list commands to install/execute)
- in each file, include the following information as a comment at the top of the file, where "username" is your unity login name and the single author is the person who wrote this file:

  ```
  Single Author info:
  username FirstName MiddleInitial LastName
  Group info:
  username FirstName MiddleInitial LastName
  username FirstName MiddleInitial LastName
  username FirstName MiddleInitial LastName
  ```