ECE4122/6122 **Spring Semester, 2017**

Project 1 – The Distributed Two–Dimensional Discrete Fourier Transform

Assigned: Wednesday January 18, 2017 *Due: Wednesday February 1, 2017, 11:59pm*

Given a one–dimensional array of complex or real input values of length N, the Discrete Fourier Transform consists af an array of size N computed as follows:

$$H[n] = \sum_{k=0}^{N-1} W^{nk} h[k] \qquad \text{where } W = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \qquad \text{where } j = \sqrt{-1} \quad (1)$$

For all equations in this document, we use the following notational conventions. h is the discrete–time sampled signal array. H is the Fourier transform array of h. N is the length of the sample array, and is always assumed to be an even power of 2. n is an index into the h and H arrays, and is always in the range $0 \ldots (N-1)$. k is also an index into h and H, and is the summation variable when needed. j is the square root of negative one.

The above equation clearly requires N^2 computations, and as N gets large the computation time can become excessive. There are a number of well–known approaches that reduce the comutation time considerably, but for the purpose of this assignment you can just use the simple double summation shown above.

Given a two–dimensional matrix of complex input values, the two–dimensional Fourier Transform can be computed with two simple steps. First, the one–dimensional transform is computed for each row in the matrix individually. Then a *second* one–dimensional transform is done on each *column* of the matrix individually. Note that the transformed values from the first step are the inputs to the second step.

If we have several CPU's to use to compute the 2D DFT, it is easy to see how some of these steps can be done simulataneously. For example, if we are computing a 2D DFT of a 16 by 16 matrix, and if we had 16 CPUs available, we could assign each of the 16 CPU's to compute the DFT of a given row. In this simple example, CPU 0 would compute the one–dimensional DFT for row 0, CPU 1 would compute for row 1, and so on. If we did this, the first step (computing DFT's of the rows) should run 16 times faster than when we only used one CPU.

However, when computing the second step, we run into difficulties. When CPU 0 completes the one–dimensional DFT for row 0, it would presumably be ready compute the 1D DFT for *column* 0. Unfortunately, the computed results for all other columns are not available to CPU 0 easily. We can solve this problem by using *message passing*. After each CPU completes the first step (computing 1D DFT's for each row), it must send the required values to the other processes using *MPI*. In this example, CPU 0 would send to CPU 1 the computed transform value for row 0, column 1, and send to CPU 2 the computed transform value for row 0, column 2, and so on. When each CPU has received k messages with column values (k is the total number of columns in the input set), it is then ready to compute the column DFT.

Finally, each CPU must report the final result (again using message passing) to a designated CPU responsible for collecting and printing the final transformed value. Normally, CPU 0 would be chosen for this task, but in fact any CPU could be assigned to do this.

We are going to use 16 CPUs in the *deepthought* cluster to perform the 2d DFT using distributed computing.

Copying the Project Skeletons

1. Log into `deepthought19.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE6122 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE6122/FourierTransform2D .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `FourierTransform2D`

   ```
   cd FourierTransform2D
   ```

4. Copy the provided `fft2d-skeleton.cc` to `fft2d.cc` as follows:

   ```
   cp fft2d-skeleton.cc fft2d.cc
   ```

5. Then edit `fft2d.cc` to implement the transform

6. Compile your code using `make` as follows:

   ```
   make
   ```

7. Run your program using the provided runLocal as follows:

   ```
   ./runLocal
   ```

1. Implement a simple one–dimensional DFT using the double summation approach in the equations above. Use 16 CPU's, and allocate the 1D transforms to each CPU depending on the rank and the number of rows in the image (the height).

2. Use MPI send and receive to send partial information from the 16 processes to the CPU at rank zero. After CPU zero has collected the transformed rows from all other ranks, use the `SaveImageData` function provided by the `InputImage` object to save the 1D results to file `MyAfter1D.txt`. **NOTE. It is important to use exactly the right file name as specified above. If the name is wrong the grading script will report mismatches and count off.**

3. Once you have gotten the fft2d program to properly compute the 1D transforms, continue to edit your program to calculate the 2D transform, calculating the 1D transforms on the *Columns* of the image.

4. When you are confident of the resulting 1D transformed image, use the `SaveImageData` method from the `InputImage` object. The 2D image MUST be saved in a file named `MyAfter2D.txt`.

5. **GRADUATE STUDENTS ONLY.** Implment a *reverse DFT Algorithm* and use it to restore the transformed image to the original real values for each pixel. Use the `SaveImageDataReal` method to save the image to file named `MyAfterInverse.txt`.

Resources

1. `fft2d-skeleton.cc` is a starting point for your program.

2. `runLocal` is the script to run your program using MPI.

3. `Complex.cc` and `Complex.h` provide a completed C++ object containing a complex (real and imaginary parts) value.

4. `Makefile` is a file used by the `make` command to build fft2d.

5. `Tower.txt` is the input dataset, a 256 by 256 image of the Tech tower in black and white.

6. `after1d.txt` is the expected value of the DFT after the initial one–dimensional transform on each row, but before the column transforms have been done.

7. `after2d.txt` is the expected output dataset, a 256 by 256 matrix of the transformed values.

8. `InputImage.cc` and `InputImage.h` that will ease the reading of the input data. This object has several useful functions to help with managing the input data.

9. DO NOT ASSUME that the MPI size will always be 16. Instead implement the program with a variable (perhaps `nCPUs`) that contains the MPI size.

   (a) The `InputImage` constructor, which has a `char*` argument specifying the file name of the input image.

(b) The `GetWidth()` function that returns the width of the image.

(c) The `GetHeight()` function that returns the height of the image.

(d) The `GetImageData()` function returns a one-dimensional array of type `Complex` representing the original time-domain input values.

(e) The `SaveImageData` function writes a file containing the transformed data.

Turning in your Project. Details on how to turn in your project will be provided by email prior to the due date.

Some thoughts on implementing the 2D DFT

1. After reading in the original `Tower.txt`, create a second array of size (width * height) which represents the H array (the transformed data). We need this because the simple DFT algorithm given above cannot transform *in-place*. In other words, the H array and the h array are different areas of memory.

2. Clearly a working one–dimensional DFT is needed before a correct two–dimensional DFT can be implemented. Consider using a single CPU (no MPI) to read the `Tower.txt` file (using `InputImage`), and doing a one–dimensional DFT on all rows. Then save the resulting file (again using the `InputImage.SaveImageData`) and comparing to `after1d.txt`.

3. Once the one–dimensional DFT is working, use MPI as described above, and assign each CPU the correct number of rows, and what row they should start on. Assuming `nCpus` is the variable containing the number of CPU's, `nRows` is the number of rows in the image, and `myRank` is the rank number of this CPU, then the number of rows per CPU is `nRows / nCpus`, and the starting row number for each CPU is `nRows / nCpus * myRank`.

4. After computing the one–dimensional DFT on each assigned row use MPI to send transformed information to CPU zero for further processing

5. Consider using non-blocking send (`MPI_Isend`) to queue up all needed send operations, then use blocking receive to receive the data from your peers. You can use the `MPI_SOURCE` returned in the `MPI_STATUS` variable filled in by `MPI_Recv` to determine where each message came from and where it belongs in the H array. This is preferred but not a requirement.

6. After CPUP zero receives information from all other ranks, use the 16 CPU's again to compute the DFT on each column. This should be similar to what was done on the original 1D DFT.

7. After the second set of transforms, use MPI to send all computed information to the master (CPU rank 0). You can use non-blocking send here, as we are sure CPU 0 will eventually call `recv`.

8. Finally, write out the final 2d transform using `SaveImageData` using file name `MyAfterInverse.txt`.

Information on how to turn in your program and how to run the automatic grading script will be provided later.