ECE6122/4122 **Fall Semester, 2016**

Project 1 – The Distributed Two–Dimensional Discrete Fourier Transform

Assigned:Thursday September 1, 2016 *Due: Friday September 16, 2016, 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.

`http://support.cc.gatech.edu/facilities/instructional-labs/deepthought-cluster/how-to-run-jobs-on-the-deepthought-cluster`

For now ignore the discussion about using `qsub` *to submit job requests. We will discuss this in class.*

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.
   (a) Implement a simple one–dimensional DFT using the double summation approach in the equations above.
   (b) Use MPI send and receive to send partial information between the 16 processes.
   (c) Use CPU at rank zero to collect the final transformed values from all other CPU's, and write these results to a file called `MyAfter2D.txt` using the `SaveImageData` method in class `InputImage`.

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

   ```
   make
   ```

Resources

1. `fft2d-skeleton.cc` is a starting point for your program.
2. `Complex.cc` and `Complex.h` provide a completed C++ object containing a complex (real and imaginary parts) value.
3. `Makefile` is a file used by the `make` command to build fft2d.
4. `Tower.txt` is the input dataset, a 256 by 256 image of the Tech tower in black and white.
5. `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. This is for debugging only as the assignment does not need to write the one–dimensional transformed results.
6. `after2D.txt` is the expected output dataset after the two dimensional transformation, a 256 by 256 matrix of the transformed values.
7. `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.
8. 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.
   (f) The `SaveImageDataReal` function writes a file with only the real part of the image transformed data.

Output File Naming Convention In order to ease the grading procedure, you must write the results of the 2D transform on a file named `MyAfter2D.txt`. For debugging, if you want the results of the 1D transforms, write the results to a file named `MyAfter1D.txt`. For graduate students also computing the reverse transform, write those results to a file named `MyAfterInverse.txt`.

Graduate Students only . After the 2D transform has been completed, use MPI again to calculate the *Inverse* transform, and write the results to file `MyAfterInverse.txt`. Use the `SaveImageDataReal` function to write results. This function writes the real part only (as the imaginary parts should be zero or near zero after the inverse.

Turning in your Project. Details about turning in your program will be provided.

Some thoughts on implementing the 2D DFT

1. Consider using Rank 0 as the traffic cop to handle the data collection and dissemination, while at the same time Rank 0 would process its share of the FFT computations. Sharing the computation in this way allows us to use exactly 16 logical processes, since Rank 0 is performing "double duty" computing 1D FFT's and managing all of the data exchanges between the ranks.

2. 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.

3. 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`.

4. 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`.

5. After computing the one–dimensional DFT on each assigned row use MPI to send information to all other CPU. Each CPU will need to send `nCpus - 1` messages, one message to all other CPUs. Also you will likely need a separate array of type Complex, as the data to be send is not necessarily in sequential memory locations.

6. You shoudl consider a mix of blocking and non-blocking MPI calls to move the data around the various ranks as needed.

7. Also consider using the Tag field in the send and receive calls to get the received data in the proper location as you move the rows and columns around.

8. After receiving information from all of your peers (and storing that information in the H array in the right place), perform the column-wise one–dimensional DFT on each of your columns. You might need yet another array of size (width * height) for this as well, although you can actually use a smaller array (each CPU only uses operates on a subset of all the columns).

9. 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`.

10. Finally, write out the final 2d transform using `SaveImageData`.