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 FFT, it is easy to see how some of these steps can be done simultaneously. For example, if we are computing a 2D FFT 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 FFT of a given row. In this simple example, CPU 0 would compute the 1D FFT for row 0, CPU 1 would compute for row 1, and so on. If we did this, the first step (computing FFT’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 1D FFT for row 0, it would presumably be ready compute the 1D FFT 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 FFT’s for each row), it will send the required values to the other processes in a message. 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 FFT.

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.

You should use your solution to the original FFT lab to perform the individual 1D FFT values, either using the efficient Danielson-Lanczos approach or the less efficient approach. I will provide an inefficient transform for anyone who was unable to get the FFT working properly in the earlier assignment. Clearly, the efficient approach is more desirable, as your program will run quite a bit faster with the efficient algorithm.

Details of the Assignment We are going to use 16 CPUs in the Hogwarts cluster for this assignment. The cluster has 6 individual computers, each with 8 CPU’s. We will use 4 CPUs on each of 4 systems. The job scheduling on hogwarts is done by the Torque Resource Manager. This is described on the hogwarts web page at:

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

A correct job scheduling script is provided for you in file runFFT.sh. Instructions for submitting jobs are given below.

We will use a simplified version of the Message Passing Interface (MPI) to exchange information between the processors. This is described in detail below.

The skeleton FFT2D object will write the transformed values to the Tower-complex.txt file using the DumpTransformedValues function. This should be called by exactly one of the CPUs (the CPU that you designated to collect the final transformed values). CPU 0 seems a reasonable choice for this.

You can assume that the input matrix is square. The width and height will be the same.

For this assignment, we won’t be using valgrind to check for memory leaks, as memory management is not the focus of this assignment.

Copying the Project Skeletons

1. Log into dumbledore.cc using ssh and your prism log-in name.

2. Copy the files from the ECE3090 user account using the following command:

/usr/bin/rsync -avu /nethome/ECE3090/FourierTransform2D .
Be sure to notice the period at the end of the above command.

3. Change your working directory to FourierTransform2D
   
   cd FourierTransform

4. Start with your previous implementation of the Complex class, and your previous FFT solution. For this assignment, you won’t need the string constructor and StringParse, so remove those if you have them.
   
   cp ../FourierTransform/complex.cc .
   cp ../FourierTransform/complex.h .

5. Edit the runFFT.sh script to change YourLastNameHere to give your last name. This is how your jobs are identified to the job scheduler.

6. Then edit fft2d.cc and fft2d.h to implement the transform.

7. Once you have implemented the transforms, you can test your solution with the provided inputs. Testing is done by submitting your job to the scheduler using qsub.
   
   qsub runFFT.sh

8. You can check the status of your submitted jobs by using qstat
   
   qsub runFFT.sh

9. When your submitted job completes, you will have two files created in your working directory with the outputs from the job (presumably the 2D Transformed values) and any errors produced by your job (hopefully empty).

Resources

1. runFFT.sh is the job submission script to use. Be sure to change the -N parameter to your name.

2. hostfile is the list of hosts that you want your job to run on. You should not change this.

3. Tower.txt is the input dataset, a 1024 by 1024 image of the Tech tower in black and white.

4. Tower-Complex-Correct.txt is the expected output dataset, a 1024 by 1024 matrix of the transformed values.

5. 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.
   
   (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 GetRows() function has two arguments, the starting row number desired and the number of rows desired. The return type is Complex**. This means it is a pointer to an array of Complex*. Each of the array elements represents one row in the input data. This function is useful since for the distributed 2D FFT, each processor is only interested in a subset of all of the rows.

6. MPI.h and MPI.cc that implement the simple MPI interface described below.

7. The DisplayImage program that will produce a graphical window with the specified image. The input image contains real values only.

8. The DisplayImageComplex program that will display an image of complex values. The magnitude of each complex value in the input dataset determines the pixel brightness. You can use this to display the Tower-complex.txt file that is the final output.
The MPI Interface  The Message Passing Interface (MPI) is widely used in large-scale scientific applications on supercomputer platforms. The specification is quite large and complex, with a moderately steep learning curve. Since the purpose of this assignment is to understand concepts of distributed computing and message passing, rather than the actual MPI subroutines, we will use a simplified wrapper around MPI. The wrapper consists of four basic actions:

1. Initialize the message passing interface. This is done by simply creating an object of class MPI in your main program (this is already included in the skeleton provided). The constructor for the MPI object does the necessary actions to get started.

2. Terminate the message passing interface. This is done by having the MPI object go out of scope (which calls the destructor). The destructor provided for the MPI object makes the necessary calls to shut down gracefully.

3. Send a message. This is done with three sub-steps.
   (a) Call the StartSendMessage member function of the MPI object. The single argument is the destination CPU number. This sets up the necessary buffers for creating a new message.
   (b) Call the AddToMessageInt or AddToMessageDouble member function to add message contents. For our simple wrapper around MPI, we can only send 32-bit integers or 64-bit doubles. However, we can send as many of each as we want in a single message, as long as the complete message size does not exceed 64k bytes.
   (c) Call EndSendMessage to finish a single message and send it to the destination. Recall the destination was specified in the first step above.

4. Receive a message. To receive a message, there are three sub-steps given below.
   (a) To start a message receive, call the StartReceiveMessage member function of the MPI object. The return value is either −1 to indicate no messages are available, or the CPU identifier of the sender if a message is ready to be received.
   (b) If a message is available, call GetFromMessageInt or GetFromMessageDouble to extract the message contents.
   (c) When all of the message values have been received, call EndReceiveMessage to terminate this message and clean up the allocated buffers.

In addition, the wrapper provides a way to find out how many CPU’s are assigned to this job (by calling CPUCount) and to find out which CPU index this process is (by calling ThisCPU()). These values are needed in this assignment for each process to determine which rows and columns it is responsible for.

A simple example exercising these calls is provided in testmpi.cc.

Turning in your Project.  Your assignment will be turned in automatically on the due date. Be sure to put your code in subdirectory FourierTransform2D in your home directory on dumbledore.cc