EEE5501-01 **Spring 2021**

Assignment 4: Message Passing Interface

Due: Sunday, May 30, 2021, 11:59PM

1 Introduction

- The objective of this assignment is to implement a message passing interface (MPI) program in C++.
- In this assignment, you have to write a two-dimensional discrete Fourier transform (DFT) algorithm using MPI based on the lecture notes of `mpi.pdf`.
- In Ubuntu or Mac terminal, download a tar copy of skeleton code.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/mpi.tar
```

- Decompress the `tar` file, and go to the `mpi/` directory. Type `ls` to find the following list of files.

```
$ tar xf mpi.tar
$ cd mpi/
$ ls
abort.h  dft.h  main.cc   output-1d  stopwatch.h
data.h   input  Makefile  output-2d  tar.sh
```

- From the listed files, `dft.h` is the only file you will have to work on.
- You are allowed to change other files for your own testing and validation, but they will revert to the original state when your assignment is graded.
  - `dft.h` defines a function named `dft2d()` that is supposed to perform 2-D DFT and called from `main()`.
  - `main.cc` has the `main()` function.
  - `data.h` provides a couple of file-handling functions that read/write data points from/to a file.
  - `abort.h` has an inline function that aborts the program on a failure of MPI function call.
  - `input` is an input file. It has 1024×1024 data points to perform 2-D DFT.
  - `output-1d` has absolute values of 1024×1024 complex numbers after row-wise 1D DFT.
  - `output-2d` has absolute values of complex numbers after 2-D DFT. These output files can be used for validating your DFT implementation.
- Type `make` in the terminal to compile the code.

```
$ make
mpic++ -Wall -Werror -g -std=c++11 -o main.o -c main.cc
mpic++ -o mpi main.o
```

- A run command needs to specify an input to perform 2-D DFT. It must be the `input` file in the `mpi/` directory.

```
$ ./mpi
Usage: ./mpi <input_file>

$ mpiexec -n 4 ./mpi input
Elapsed time = 0 usec
```

- The `input` file is stored as a simple text file and contains 1024×1024 data points. The first line of text file specifies the data dimension (i.e., width and height) simply as `1024 1024`.

2 Implementation

- The next code box shows the `main()` function. Unlike the previous assignment, `main()` does not restrict the number of processes engaged in a program execution.
- It loads data points from an input file specified by `argv[1]` through `read()` defined in `data.h`.
- The type of `data` array is defined as `std::complex<float>`. `std::complex` is a template class of complex number, and its real and imaginary numbers are stored as `float`.
- When the `read()` function returns, `data` points to an array of 1 million data points (= 1024×1024), and `width` and `height` variables contain the width and height of 2-D matrix.
- The data points are stored in an 1-D array of row-major arrangement, but the array is supposed to be considered as a 2-D matrix.
- After the initialization of MPI environment, `main()` calls `dft2d()` to perform 2-D DFT. This function is surrounded by `stopwatch` operations to measure the execution time of `dft2d()` function.
- After the 2-D DFT is done, rank 0 displays the elapsed time and stores the 2-D DFT outcome to a file named `result` in the same format as `input`.
- Some data points in the `result` file may have different fraction digits compared to those in the `output-2d` file depending on how `float` numbers are calculated and rounded. You can disregard the rounding errors.

```
/* main.cc */

#include <complex>
#include <iostream>
#include <mpi.h>
#include "abort.h"
#include "data.h"
#include "dft.h"
#include "stopwatch.h"

int main(int argc, char **argv) {
    if(argc != 2) {                      // Run command message
        std::cerr << "Usage: " << argv[0] << " <input_file>" << std::endl;
        exit(1);
    }

    const char *data_file = argv[1];     // Input file
    std::complex<float> *data = 0;       // Data array
    unsigned width = 0, height = 0;      // Data dimension
    int num_ranks = 0;                   // Communicator size
    int rank_id = -1;                    // Rank ID

    // Read data file.
    read(data_file, data, width, height);
    // Initialize MPI.
    abort(MPI_Init(&argc, &argv));
    // Get the communicator size and rank ID.
    abort(MPI_Comm_size(MPI_COMM_WORLD, &num_ranks));
    abort(MPI_Comm_rank(MPI_COMM_WORLD, &rank_id));

    stopwatch_t stopwatch;
    stopwatch.start();
    // Two-dimensional discrete Fourier transform
    dft2d(data, width, height, num_ranks, rank_id);
    stopwatch.stop();
```

```
    // Rank 0 displays the runtime and stores the final result to a file.
    if(!rank_id) { stopwatch.display(); write("result", data, width, height); }

    // Finalize MPI.
    abort(MPI_Finalize());
    // Close data.
    fin(data);

    return 0;
}
```

- The next shows the skeleton code of `dft.h` that you will have to implement a 2-D DFT algorithm.
- There are several different ways to implement the 2-D DFT, but you are asked to follow the steps described in the comment of `dft2d()` function.
- First, perform row-wise 1-D DFT. Each 1-D DFT works on 1024 data points of a row, and it is repeated for 1024 rows. Since row-wise DFTs can execute independently, this is where parallel MPI processes divide the data.
- Be aware that 1024 rows of `data` may not be evenly divisible by `num_ranks`. You must consider load balancing between MPI processes.
- After row-wise 1-D DFTs, `MPI_Allgather()`-like operations are needed to synchronize the results across processes. If 1024 rows of `data` are not evenly divisible, every process must work on different number of rows. This condition hinders a naive use of `MPI_Allgather()`.
- Instead, you will have to implement `MPI_Allgather()`-like operations using a pair of `MPI_Irecv()` and `MPI_Send()` (or the opposite pair).
- `MPI_Irecv()` needs to identify how many rows of data have to be collected from other processes based on their rank IDs and the load distribution method. `MPI_Send()` is repeatedly called to broadcast 1-D DFT result of the caller process to others.
- The 2-D matrix needs to be processed once again for column-wise DFT. This can be done by transposing the matrix and repeating the previous steps of row-wise DFT.
- After the second round of DFT is done and all-gathered, the matrix is transposed back to the original orientation.

```
/* dft.h */

#ifndef __DFT_H__
#define __DFT_H__

#include <complex>
#include <cstdlib>
#include <cstring>
#include <mpi.h>
#include "abort.h"
#include "data.h"

// Perform 2-D discrete Fourier transform (DFT).
template <typename T>
void dft2d(T *data, const unsigned width, const unsigned height,
           const int num_ranks, const int rank_id) {
    /* Assignment:
       a. Perform row-wise one-dimensional DFT.
       b. Transpose the data matrix for column-wise DFT.
       c. Perform row-wise one-dimensional DFT on the transposed matrix.
       d. Transpose the data matrix back to the original orientation.
```

```
    */
}

#endif
```

- Once the 2-D DFT is implemented, executing the MPI code will produce the following output, for example.
- Gradually increasing the number of MPI processes up to total CPU core count decreases the execution time.

```
$ for i in `seq 1 $(nproc)`; do echo -e "NP = $i: \c"; mpiexec -n $i ./mpi; done
NP = 1: Elapsed time = 867.292 msec
NP = 2: Elapsed time = 433.143 msec
NP = 3: Elapsed time = 298.946 msec
NP = 4: Elapsed time = 231.657 msec
NP = 5: Elapsed time = 192.586 msec
NP = 6: Elapsed time = 165.399 msec
NP = 7: Elapsed time = 146.195 msec
NP = 8: Elapsed time = 140.539 msec
NP = 9: Elapsed time = 135.668 msec
NP = 10: Elapsed time = 128.036 msec
NP = 11: Elapsed time = 120.773 msec
NP = 12: Elapsed time = 116.154 msec
NP = 13: Elapsed time = 108.427 msec
NP = 14: Elapsed time = 105.638 msec
NP = 15: Elapsed time = 102.589 msec
NP = 16: Elapsed time = 99.293 msec
```

- Since the execution time of a code strongly depends on the performance of executing hardware and number of cores, the absolute time in seconds is not a part of grading.
- However, your code must show improving performance for increasing the number of processes.
- You should observe the maximum performance when the number of processes is the same as the number of (physical or logical) cores in your CPU. Oversubscribed MPI processes are likely to degrade the performance.

3 Discrete Fourier Transform

3.1 Danielson-Lanczos Lemma

- This section explains a DFT implementation that achieves O(Nlog$N$) as opposed to O(N^2) that a naive DFT algorithm gives.
- Given an one-dimensional array of length N, DFT is calculated as follows.

$$X[n] = \sum_{k=0}^{N-1} W^{nk} x[k] \ , \ \text{where} \ \ W = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \tag{1}$$

- In Eq. (1), x is a discrete-time sample, and X is the Fourier-transformed array of x samples. The array length, N, is assumed to be power of two. n and k are indices into the X and x arrays, ranging from 0, 1, 2 $\cdots$ to $N-1$. j is the square root of -1.
- The naive calculation of Eq. (1) requires O(N^2) operations. However, it can be much efficiently calculated by following the Danielson-Lanczos Lemma for O(Nlog$N$) operations.
- The insight of Danielson-Lanczos Lemma is that the Fourier transform of an array of length N can be decomposed into the sum of two smaller Fourier transforms of length $N/2$, where the first half of transformation contains even-index data points, and the second half has odd-index data.

- Eq. (2) shows the decomposition of $X[n]$ into two parts, one comprised of even indices of x array and another of odd indices of x.

$$\begin{aligned} X[n] &= \sum_{k=0}^{N-1} e^{-j2\pi kn/N} x[k] \\ &= \sum_{k=0}^{N/2-1} e^{-j2\pi 2kn/N} x[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi(2k+1)n/N} x[2k+1] \\ &= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} x[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} e^{-j2\pi n/N} x[2k+1] \\ &= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} x[2k] + W^n \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} x[2k+1] \\ X[n] &= X^e_{n,N/2} + W^n X^o_{n,N/2} \end{aligned} \tag{2}$$

- In this equation, X^e_n refers to the nth element of Fourier-transformed array of length $N/2$ that is comprised of even-index elements from the input array of length N.
- Similarly, X^o_n refers to the nth element of Fourier-transformed array of length $N/2$ that is formed by odd-index elements from the input array of length N.
- The Fourier transform of the array of length N is calculated by adding two transforms of half-length arrays.
- Each of the $N/2$ transforms can be calculated as the sum of two $N/4$ transforms, which again can be calculated as the sum of two $N/8$ transforms, and so on. This process repeats until the array length becomes 1.
- Consequently, it exploits a divide-and-conquer strategy similar to the previous assignment of merge sort.
- The special case of length 1 simply gives $X[0] = x[0]$, which means that the Fourier transform of a single data point is just the original sample itself.
- Using the Danielson-Lanczos Lemma, the DFT of length $N = 2^m$ can be rapidly calculated. This strategy is known as Cooley-Tuckey algorithm, and it achieves O(Nlog$N$) for one-dimensional transformation.

3.2 Reordering Input Array x

- Suppose an array x has eight elements (i.e., N = 8); $x[0]$, $x[1]$, $x[2]$, $\cdots$, $x[7]$.
- Categorizing the eight elements into even and odd indices gets the following sets.

$$\begin{aligned} X^e &: x[0], x[2], x[4], x[6] \\ X^o &: x[1], x[3], x[5], x[7] \end{aligned} \tag{3}$$

- Recursively categorizing the sets into even and odd elements gives the following.

$$\begin{aligned} X^{ee} &: x[0], x[4] \\ X^{eo} &: x[2], x[6] \\ X^{oe} &: x[1], x[5] \\ X^{oo} &: x[3], x[7] \end{aligned} \tag{4}$$

- Finally, dividing each of the above in Eq. (4) into even and odd elements produces the following.

$$\begin{aligned} X^{eee} &: x[0] \\ X^{eeo} &: x[4] \\ X^{eoe} &: x[2] \\ X^{eoo} &: x[6] \\ X^{oee} &: x[1] \\ X^{oeo} &: x[5] \\ X^{ooe} &: x[3] \\ X^{ooo} &: x[7] \end{aligned} \tag{5}$$

- Each line of Eq. (5) is simply a Fourier transform of length 1, and it is trivial to calculate it; nothing to do.
- An actual challenge is in finding the sequence of $x[0]$, $x[4]$, $x[2]$, $\cdots$. It turns out that it is possible to find the sequence by mapping 0 to e and 1 to o for the superscripts of X and interpreting the binary representations in the reverse order.
- For instance, a superscript of X^{eeo} is represented as 001. Reading the binary digits from the right gives 100 that is 4 in decimal.
- The first step of Cooley-Tuckey algorithm is to shuffle the input array x from the natural ordering (i.e., $x[0]$, $x[1]$, $x[2]$, $\cdots$, $x[N-1]$) to bit-reversed order.
- For the array of length 8, bit-reversed order of array elements is shown in Eq. (5). Arrays of different lengths result in different bit-reversed orderings, but they are not difficult to find.
- After shuffling the input array x, it becomes easier to calculate the partial sums of X. For the array of length 8, elements of the first set of length 2 (i.e., $x[0]$ and $x[4]$) are located at the index 0 and 1. Likewise, elements of the second set of length 2 (i.e., $x[2]$ and $x[6]$) are at the index 2 and 3.

3.3 Pre-calculating W^n Weights

- A complex weight, W, shown in Eq. (1) is defined as follows.

$$\begin{aligned} W &= e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \\ W^n &= e^{-j2\pi n/N} = \cos(2\pi n/N) - j\sin(2\pi n/N) \end{aligned} \tag{6}$$

- Since n is an index into an array of length N, there are total N weight values such as $W^0, W^1, W^2, \cdots, W^{N-1}$.
- The trigonometric functions are relatively expensive to compute in every iteration of Fourier transform, and thus they better be pre-calculated and stored in advance.
- In fact, it is necessary to calculate only a half of the complex weights because $W^{n+N/2} = -W^n$. Using this identity, W^n needs to be calculated only for n = 0, 1, 2, $\cdots$, $N/2-1$.

3.4 Logarithmic Transformation

- Once the input array x is shuffled in the bit-reversed order and complex weights of W are pre-calculated, a transformation starts with a pair of two neighboring elements in the shuffled array.
- For instance, the array of length 8 starts with four sets of length 2, which are $(X[0], X[1])$, $(X[2], X[3])$, $(X[4], X[5])$, and $(X[6], X[7])$. Note that each element of X at this stage contains an element of shuffled x.
- In the first set, each component of the even-odd pair is calculated as Eq. (7).

$$\begin{aligned} X[0] &= X^e_{0,2/2} + W_2^0 X^o_{0,2/2} = X[0] + W_2^0 X[1] \\ X[1] &= X^e_{1,2/2} + W_2^1 X^o_{1,2/2} = X[0] + W_2^1 X[1] \end{aligned} \tag{7}$$

- The notation of W_N^k indicates that there are N samples in this transformation, and k in an index out of N.
- Fortunately, it is not necessary to re-calculate W_N^k weights for transformations of doubling N (e.g., N = 1, 2, 4, $\cdots$), since the following equality holds for the pre-calculated weights of length N.

$$W_x^k = W_N^{kN/x} \tag{8}$$

- In the previous example of two-element transformation, the weights can be obtained from the pre-calculated array as follows.

$$\begin{aligned} W_2^0 &= W_8^{0(8/2)} = W_8^0 \\ W_2^1 &= W_8^{1(8/2)} = W_8^4 = -W_8^0 \end{aligned} \tag{9}$$

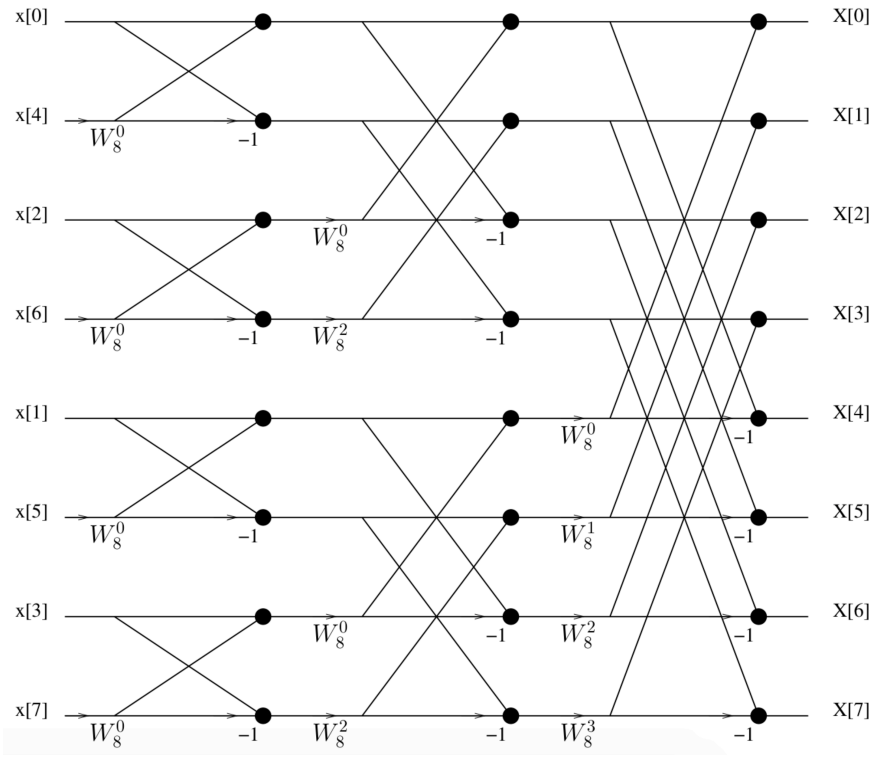
- As a result, the two-element transformation becomes the following.

$$\begin{aligned} X[0] &= X_{0,2/2}^e + W_2^0 X_{0,2/2}^o = X[0] + W[0]X[1] \\ X[1] &= X_{1,2/2}^e + W_2^1 X_{1,2/2}^o = X[0] - W[0]X[1] \end{aligned} \tag{10}$$

- Similar to the two-element transformation, a four-element transformation is expressed as follows.

$$\begin{aligned} X[0] &= X_{0,4/2}^e + W_4^0 X_{0,4/2}^o = X[0] + W[0]X[2] \\ X[2] &= X_{2,4/2}^e + W_4^2 X_{2,4/2}^o = X[0] - W[0]X[2] \\ X[1] &= X_{1,4/2}^e + W_4^1 X_{1,4/2}^o = X[1] + W[2]X[3] \\ X[3] &= X_{3,4/2}^e + W_4^3 X_{3,4/2}^o = X[1] - W[2]X[3] \end{aligned} \tag{11}$$

- Note that the transformations occur *in place* without allocating a separate memory space. The following picture illustrates the paths of 8-element array transformation.



4 Submission

- Once the assignment is done, execute the `tar.sh` script in the `mpi/` directory.
- It will compress the `mpi/` directory but not including the data files (i.e., `input`, `result`, and `output-*`). A resulting `tar` file is named after your student ID such as `2021310000.tar`.

```
$ ./tar.sh

$ ls
2021310000.tar  data.h  input    Makefile    output-2d    tar.sh
abort.h         dft.h   main.cc  oputput-1d  stopwatch.h
```

- Upload the tar file (e.g., `2021310000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

5 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: A submitted `tar` file is renamed and includes redundant tags such as `hw4`, student name, etc.

  -5 points: A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  -15 points: The code produces the correct result of 2-D DFT, but increasing the number of processes does not decrease the execution time.

  -25 points: The code does not compile or fails to produce the correct result, but it shows substantial efforts to complete the assignment.

  -30 points: No or late submission. The following cases will also be regarded as no submissions.

  * Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.
  * Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

  Final grade = F: The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee5501`
- Begging partial credits for no valid reasons will be treated as a cheating attempt, and such a student will lose all scores of the assignment.