**CSE-443/543: High Performance Computing**

**Homework #04**

Max Points: 80

**Notes:**

1. **In the Analysis section this homework has an extra question for CSE543 students.**
2. **This is not an easy program to write and debug. I strongly suggest that you begin development immediately!**

**Objective**: The objective of this exercise is to:

* Build familiarity with compiling and running parallel code using MPI
* Build experience with compiling code at the command line and constructing batch shell scripts to run MPI jobs
* Build familiarity with the concepts of parallel program performance discussed in class and compare the performance of serial and MPI versions of the same program.
* Learn about C++ preprocessor directives and how to use them

Fill in answers to all of the questions and complete the requested data analysis. You may discuss the questions with your instructor.

## Background

For this homework assignment I am giving you a serial implementation of a program that computes the [Discrete Fourier Transform](https://en.wikipedia.org/wiki/Discrete_Fourier_transform) (DFT) and you will parallelize it using MPI. If you are not familiar with the DFT, please review the linked Wikipedia article, section 37.1 in your textbook, or find other descriptions online or in a textbook. **Note:** You do **not** need to be an expert in Fourier transforms to do this lab, but rather you must understand it well enough to implement it correctly. Though this may feel strange, in the future you will very likely be developing programs for clients who have a deep understanding of their business that you will not be able to match. Your job will be to understand the requirements and write a program that implements them correctly. Consider this practice for the future.

You should also know that for many situations, the DFT is not the most efficient way to compute the Fourier transform. It is an algorithm and a different approach, the Fast Fourier Transform (FFT), is . The FFT is, however, more complex to implement and, except for large values of N, can actually be too fast to make it easy to time. Make sure you understand this: I want you to implement the **DFT**, **not** the **FFT** for this assignment.

In this homework you will be calculating the DFT of a function with a serial program and an MPI version of that program. The function you’ll be computing the DFT of is the sinc function, sin(ax)/x, defined by the following code (R and I denote the real and imaginary components of the function):

double a = 2.0;

oR[0] = a;

oI[0] = 0.0;

for (int n = 1; n < N; n++) {

oR[n] = sin(a\*n)/n;

oI[n] = 0.0;

}

The Fourier transform is designed to handle complex functions, i.e. functions that have both a real and an imaginary component. The original function I am giving you only has a real component but you should write your code so that it could act on a complex function.

This function has interesting behavior in both the real and imaginary components of its Fourier transform. If you run the serial code without modification you can determine the correct real and imaginary components of the output for different values of N to use while debugging your program. For your final timing runs you will be using N = 75000. As with previous assignments, for development and testing purposes you may want to perform runs with a smaller value of N.

For this lab, I am giving you my version of the serial DFT program (fourier\_serial.cpp) so you are all starting from the same point. The program introduces the concept of a [C++ preprocessor directive](http://www.cplusplus.com/doc/tutorial/preprocessor/). These tell the C++ preprocessor to take some action before sending the code to the compiler. An example of this that you have used in every C or C++ program you’ve ever written is the #include directive. This tells the pre-processor to insert the specified file to the source file before doing the compilation. This is a great benefit to programmers as it shortens code significantly and eliminates coding errors, at least for system-provided include files.

Another directive you have used in this class is #pragma, in particular #pragma omp. This instructs the compiler to process the code using OpenMP so that the appropriate multithreading instructions are included in the code it produces.

The macro I am introducing in my fourier\_serial.cpp code is #ifdef. This allows for conditional compilation of code based on whether a macro is defined or not. The specific example from my code might help you understand this better.

My code has an initialize method that computes the real and imaginary components of the sinc function and also zeroes out the arrays holding the two components of the Fourier transform. There are times when I’d like to print the values of the original function out, though generally I don’t want or need to. I could add a command-line option to the code to turn output on or off, but taking that route means that I have to supply it every time the code is run. Another approach is to use #ifdef, for example:

#ifdef OUTPUT

for (int n = 0; n < N; n++) {

cout << i << " " << oR[n] << " " << oI[n] << endl;

}

exit(0);

#endif

I can activate that code in one of two ways. First, at compile time I can add -DOUTPUT as an option to g++. The preprocessor then makes that code visible to the compiler and it is included. Alternatively, I can add a #define OUTPUT statement in the code to turn this on. This kind of approach is often used when developing code, in particular while debugging code. Having conditionally compiled code set aside with a #ifdef DEBUG is very common and can save you time. Try compiling and running the code with and without -DOUTPUT to see this in action.

The DFT has two nested loops, an inner loop that goes over the variable n which represents time or space, and an outer loop that goes over the variable k which represents the frequencies. Each of these loops runs from 0 to N. To parallelize this code using MPI you should implement a manager/worker model. In the heart of the code, the manager will first give every worker a single value of k to compute the DFT for, then wait for results to come in. The manager must keep track of both how many k values it has handed out and how many it has received back. When both of those are equal to N, the computation of the Fourier transform is complete.

You should **not** assume that the workers will complete their tasks in any particular order so the manager must wait for a result from any task, with any value of k, before handing out a new value of k to that task. When you have sent all N of the k values to be worked on and received back all N real and imaginary components of the Fourier transform, you’ll need to signal the workers that their job is complete so they can send you information on how much CPU time they’ve consumed. You can take advantage of two things to make this job easier:

1. When computing the Fourier transform, the valid values of k are all greater than or equal to zero. So you can send k = -1 as a signal to the workers that it’s time to stop.
2. Remember tags? You can use the value of k that a worker is reporting on as the tag for the message the manager receives. That will save you having to send the value of k in an additional message. You can receive the message with any tag, then process the status variable to get the value of the tag.

You should copy fourier\_serial.cpp to fourier\_mpi.cpp to give you a starting point. This is an outline of how you should implement the DFT as I’ve described above. Note that you still have to write this, but the outline should hopefully help, and in addition, you have a functioning serial program to start with.

All tasks should:

* Check if a command line argument (N) has been provided and exit if it hasn’t
* Read the value of N from the command line
* Define arrays (oR and oI) for the original function
* Initialize MPI and find out the task’s rank, exiting if there’s a problem
  + Hint: A modified version of the doMPIStuff method from pingpong\_mpi.cpp would be helpful for this.
* Do worker or manager tasks based on their rank
* Finalize MPI
* Free up memory from the oR and oI arrays.
* Exit

In addition to the common tasks, the manager task (rank = 0) should:

* Define arrays to hold the Fourier transform
* Initialize the arrays for the original and Fourier transform functions
* Send the original function arrays to the worker tasks via an MPI\_Bcast call.
  + Note – the worker tasks do not need the Fourier transform arrays. They will be sent one value of k at a time and compute the real and imaginary components of the Fourier transform for that single value of k. In the language of the [Wikipedia article on the Discrete Fourier Transform](https://en.wikipedia.org/wiki/Discrete_Fourier_transform), they are computing Xk for their given value of k.
* Send every worker an initial value of k to begin working on, starting at 0.
* Wait for a results to be received
  + Determine the rank of the task that returned the value
  + Determine the value of k the value is for
  + Receive the real and imaginary components of the Fourier transform for that value of k
  + If there are still k values to compute, send a new one to the task that just sent you this value
    - If not, don’t send anything to the task. It will wait to receive the termination signal when every worker is done.
  + Increment the count of k values received
  + If there are still values to be received, wait for another one to arrive
    - If not, exit the “wait” loop
* Send all workers k = -1 to signal the end of their work.
* Determine the CPU for the manager
* Use MPI\_Reduce to accumulate the CPU time for all tasks.
* Determine the elapsed time for the whole run
* Write out the elapsed time and CPU time for the run to standard error (with cerr).
* Write out the Fourier transform (n, fR and fI) to standard output (with cout).
* Free up memory from the fR and fI arrays.

In addition to the common tasks, the worker tasks (rank > 0) should:

* Declare double variables for the real and imaginary components of the Fourier transform. These should not be arrays, just variables as the unit of work for each worker will be computing the Fourier transform for a single value of k at a time.
* Declare an int variable for k and initialize it to something greater than or equal to zero.
* Receive the original function arrays from the manager task via an MPI\_Bcast.
* Begin a while loop that runs until k is no longer >= 0.
* In this loop:
  + Receive a value of k to work on.
  + If k is greater than 0
    - Compute the Fourier transform for that value of k. (I.e. Xk)
    - Send the real part of the Fourier transform back to the manager task, using k as the tag.
    - Send the imaginary part of the Fourier transform back to the manager task, using k as the tag.
    - Check for another value of k
  + If k is equal to -1, end the loop
* Determine the CPU time used by the task and send this back to the manager task using MPI\_Reduce.

Notes:

1. Writing an MPI program that complies with CSE coding standards is very difficult. I was able to do it with one modification: Because the arguments to MPI methods can get “wordy”, I ran cpplint.py with the --linelength=120 argument. You do not have to make your code compliant, but if you do there are **2 extra credit points** awaiting you.
2. The program you submit must compile without errors or warning messages when using our standard set of arguments (-O3 --std=c++17 -Wall). It must also be the program you used to generate your results. For a previous assignment, some of you submitted a program that could not have been used to generate the results you presented! You will not get any credit for this assignment if that’s the case!
3. This shouldn’t need to be said by now, but the code you submit must be your own. You cannot share code with classmates, nor use code you have downloaded from another source. You may, however, use code from the fourier\_serial.cpp and pingpong\_mpi.cpp programs I have released to you.
4. As discussed in class, in order to run the program you will have to invoke mpirun with the command line option --use-hwthread-cpus. Though this is only necessary for more than 4 tasks, use it when running the program for all values of the number of tasks.
5. You may get some warning messages from MPI. As long as your program is running you can ignore them. Error messages are different and if your code isn’t running you have a problem you can’t ignore.

In terms of planning your work for this assignment, doing all five timing runs for the serial program should take about 12 minutes. If your parallel code is running correctly, all of your MPI timing runs (2-8 tasks and 5 runs for each number of tasks) should take less than 40 minutes total, so count on a bit more than an hour to do the timing runs for everything. Recall that you **do not** have to do these runs in one sitting and you have two weeks to complete your program and do the timing runs. Also note that if you are getting significantly longer times you may have an issue with your code, and if your run times are significantly less, make sure you are doing your timing for N = 75000. Finally, your MPI code should give exactly the same answer as the serial program.

## Experiment

Since you have 11/14 of a semester’s experience doing experiments in this class you should know the routine by now.

# Apparatus (platform for experiment)

The experiments documented in this report were conducted on the following platform:

|  |  |
| --- | --- |
| Component | Details |
| CPU Model | model name : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz |
| Main Memory (RAM) size | MemTotal: 8056076 kB |

# Observations – Homework 04

Enter your timing information for this lab in the supplied Google spreadsheet.

# Analysis – Homework 04

Compute the average elapsed and CPU time and their associated standard errors in the supplied spreadsheet. Then, compute the parallel speedup, parallel efficiency and parallel cost based on the average values, entering the results in the spreadsheet.

## For CSE543 students only:

Create graphs of the real and imaginary components of the Fourier transform of the sinc function used in this study. Each of your two graphs should contain axis labels and a title. You should add a tab to the Homework04 spreadsheet to hold your raw data for the graphs, and put the resulting graphs in the Homework04 tab below the Parallel Performance table.

# Submit files to Canvas

When you complete the lab, download this document from Google Drive as a Microsoft Word (.docx) file with the naming convention Homework04 - MUid.docx (example: Homework04 - ferrenam.docx).  When you complete the homework, download your Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Homework04 – Results - MUid.xlsx (example: Homework04 - Results - ferrenam.xlsx)

Then, submit the following files to Canvas:

1. The Microsoft Word file you downloaded from Google Drive.
2. The Microsoft Excel file you downloaded from Google Drive.
3. Your fourier\_mpi.cpp program.