CS426
Spring 2020
Project 2
Due 01/04/2020 – 23:59

Part A:

MapReduce is an extremely popular method to process huge data in a parallel fashion. It is commercially popularized by Google but the idea goes to decades ago. The basic idea behind MapReduce is good-old higher order functions from functional programming languages like Lisp, ML etc. I am sure most of you took CS315 or similar classes and this is not new to you. But even if you didn't, the idea is quite simple. A higher order function is a function that may take another function as input and may return another function as output. Here is an example;

- (map square '(1 2 3 4))
  - Output: (1 4 9 16)

Here map is a higher order function because it takes another function as input, namely square. In fact map is a special higher order function because it is used to design other higher order functions. Map takes a function and a list of items and it will return another list where the elements are replaced with applying the input function to each element of the input list.

Another higher order function that is used as a building block is fold. Here is an example;

- (fold add '(2 4 6 8 10))
  - Output: 30

This one also takes a function and a list but it returns a number, where the result is obtained by accumulating all the elements of the input array by applying input function succesively. In fact this is similar to MPI_Reduce where reduction operation is MPI_SUM.

Last higher order function we are going to look at is filter. This is like map, but it takes only predicates as input functions. A predicate is a function that takes an element and returns a boolean, like even funcion that tells wheter input is even. Here is an example;

- (filter even? '(1 2 3 4))
  - Output: (false true false true)

If you have problems with these concepts, take your time to read the following documents before moving on. These are rather easy things for a senior CS student.

https://en.wikipedia.org/wiki/Higher-order_function

http://www.sicpdistilled.com/section/1.3-higher-order/

For this part, we are going to implement the above-mentioned higher order functions in MPI using limited function manipulation capacities of C, namely function pointers. You may have never used function pointers during all your C career, so you need to brush up your C**. In this homework C++ is not allowed because C++ has built-in support for most of these things and using it would be cheating. We want to learn how these things are implemented in C++ itself or on any other language.** Quicksort implementation in C standard library is a good example to study if you need further exercise in function pointers. It is called qsort.

Implement following functions using MPI:

1. **float *MPI_Map_Func(float *arr, int size, float (*func)(float))**
   This function takes a float array, size of the array and a function pointer to a function who takes a float and returns a float. It will apply that function to each ellement of the array.
   NOTE: Function you passed might be named "add" but in the function MPI_Map_Func, it will be known as func, its local parameter name.
2. **float MPI_Fold_Func(float *arr, int size, float initial_value, float (*func)(float, float))**
   This function will take again an array, its size, initial value and a function pointer again. The function pointer should point to a function that takes two floats and returns a float. İnitial values is the value that starts the accumulator.For example this function can be used as;
   - MPI_Fold_Func(arr, 100, 1.0, &multiply)

   Which means we multiply 100 elements of the arr and our initial value is 1.0. Remember 1 is the identity element for multiplication.

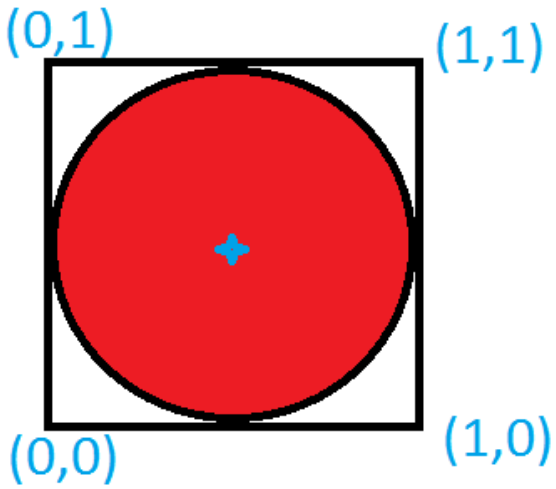
3. **float *MPI_Filter_Func(float *arr, int size, bool (*pred)(float))**
   This is similar to Map version but the function is now a predicate which takes a float and returns a boolean. You can use chars for booleans since C has no builtin boolean type. A better choice would be typedef'ing an enum of two integer values.

Implement these functions and use any communication scheme available in MPI as you want, MPI_Send, MPI_Recieve, MPI_Bcast, MPI_Scatter, MPI_Gather or MPI_Whatever... **Just don't use MPI_Reduce, MPI_Allreduce and you will be fine.** The communication scheme you chose will impact your code quality and readability and ease of writing. Do spend sime time to think on this one. Test your functions and make sure that they work reasonably well. Then move your implementation to a file called **helper.h**. We will use these functions in the next part.

Part B:

In this part, you are going to implement another method to approximate π. Simple statistics can be used to calculate π and this time we will use this probabilistic approach. Here is the idea.

Imagine you inscribed a circle into a square as following;



Note that the ratio of areas of circle and square is $\pi r^2/4r^2$ which is $\pi/4$. If we thought of this as game of darts, randomly throwing darts to this picture would hit the circle with the chance of $\pi/4$. Your program is going to do the same. You will think of the square as unit square. Throwing a random dart means chosing two random numbers, each for x and y coordiantes. Let's say you picked 0,5 and 0,5. This means you just hit the bull's eye etc. You will develop a function to determine if your result is in the circle or is it a miss. Then you are going to repeat the experiment many times. If you do this long enough, the ratio of hits to the total number of experiments should give an approximation to $\pi/4$. This is called a Monte Carlo simulation. Your program will take as input the number of experiments to be performed. Then you will distribute to each process how many experiment it should perform and carry out the task. Here comes the crucial part, you need to use MPI_Map_Func, MPI_Filter_Func and MPI_Fold_Func you implemented above to calculate and process experiments. You will include your helper.h. Once you complete Part A, implementing this will be very easy, just think a little and you will find. Also implement a serial version for this pi approximation strategy in another file called serial.c. We will use serial version to compare. **Don't forget to implement timing for your results.**

What to Upload:

Two C files, named helper.h and main.c. Another small serial implementation of above pi approximation algorithm called serial.c. We will use this serial implementation to compare. Write a little makefile to compile your program. The output names of your compiled executables should be main-serial and main-parallel. (ie. add –o main-serial etc. to you compiler flags) The compiled program will just take one command line input, number of experiments to perform, and it will just output your approximation to pi. Write a little shell script to compile and run your program that calls make and your program with inputs 1000, 10000, 100000, 1000000. Name this script run.sh. So in the end, when I call run.sh, your program should be compiled using makefile, script should call parallel and serial version four times on these inputs and output your approximations for pi for each number of experiments.

Create a report where you discuss the following points:

- What kind of parallelism, does your strategy apply for approximating pi? Discuss in terms of data and task parallelism.
- What was your communication scheme, which MPI communication functions did you use and why.
- How MPI address space is used between processes? Are they threads, or are they different OS level processes? What would be in a distributed environment where you had many different computers connected together? Would function pointers, pointers to part of the code segment of virtual memory space, still work if we had a truly distributed environment.
- Discuss your timing results and put your graphs according to number of processes.
- If you prepare your reports in LaTeX, send me the Tex file.Reports in LaTeX will get you bonus points. It will make me magically ignore your otherwise easily noticable mistakes, of course up to a certain point. In short, you won't get points higher than 100 but I will be a little more tolerant while grading if you prepared reports in LaTeX.

Put all relevent code, makefile, shell script and your report into a zip file.

Email: berkay.gulcan@bilkent.edu.tr

Email subject: CS426_HW2 (Without subject it will not be evaluated).

Zip File name: yourname_lastname_p2.zip

No Late Submission Allowed!

Do not use other names for your files and definitely do not use wrong name in the mail topic. It should be exactly CS426_HW2.

The code is shorter for this homework compared to first one but it might require you to make more preparation and thinking. So do start early.

Some Clarifications

- Changing the signature of the map or other functions is not permitted.
- These are generic functions that work for any data type.
- So in a language like C++ or Java, your map function would be implemented generically for a type variable T lets say. Then you could just call map with a list of points or whatever data type you have. But since C is not well-suited for generic programming, I wanted you to implement these functions only for float data type.
- Implementations in part A has nothing to do with part B. These functions are classic building blocks used in data processing. The job of the programmer is not to change these functions according to their needs but to find a way to express their problems in terms of combinations of these functions.
- So rather than changing the functions in part A, you should think about "how can I formulate this problem as a float list processing task". Maybe you should not insist on using map contrary to its rules but you should find a way to incorporate other filters into your solution? There are multiple ways to express same task as combinations of map filter fold etc. You need to find a way using these functions.
- Implementing effectively Part B is your job. You can measure the performance according to data parallelism and task parallelism and compare. Our goal is to obtain a SIMD execution using MPI which may be represented as pipelined parallelism.
- In short, you need to implement part A as building blocks, useful abstractions without changing them. In part B, you need to find a way to use these abstractions as if they were black boxes to you and you need to formulate the problem accordingly.