Dylan Dunsheath Homework 8 Description CS288

**Problem 1**

This code is a representation of the count sort algorithm for a binary file with parallel Messaging Passing Interface among the processes. The file/program takes three different libraries: stdio.h, stdlib.h, and mpi.h; specifically for input/output, dynamically allocating the array, and mpi (which was installed by sudoing mpich). Outside of any functions, I also defined a constant called MAX which took a value of 1000 (used for the count array) and a global variable called arr. Arr is essentially a global integer pointer that is used to be sorted from the function countElems. I will explain the main function then the countElems; in order to retain the reader’s understanding

The **main function** of the program is what drives the program to even work and is a program that takes in an argument which is done by executing:

* mpirun -n [number of processes] ./<executable>

When the program is started, it calls MPI\_Init(&argc, &argv) which is used to start MPI and a process group called MP\_COMM\_WORLD. I then declared a variable called world\_size which is used within the MPI\_Comm\_size method to get the number of processes that the user wants to pass to the program and then another variable called world\_rank; which gets the rank for each processs: this is important because each process is identified by its rank (it’s specifically used to retrieve the rank of the calling process). After that, I check the world\_size/number of processes that were passed to the program. If it doesn’t allow for parallelism (or less than 2 processes), than notify the user and exit out; as it defeats the purpose to less than two processes for a parallel programming assignment. Afterwards, I declared an array of size 256 (as I don’t exactly expect the size needs to be even THAT big for testing my program and a variable called ‘size’ which is used for the size of my dynamically allocated array from a given file (how many #s there are). When it comes to the way the processes work:

* Process 0 (let’s call it the ‘ROOT’) process, is what interacts with the user to get the name of the file (which reads UP to 255 characters AND allows for white spaces in the name. So fiftyValues or ‘fifty values’ are valid names). It then continuously checks until a valid file is found and then exits out of the while (file == NULL) loop. In this process, we also use a method called fread to continuous;y read ONE number at a time until we reach the end of the file (the numbers are all integers of sizeof(int)) and we place these values into the num variable. Afterwards, we use the realloc from stdlib.h library to continuously increase the size of the array by one until we reach the end of the file and then adds it to the appropriate index. We then close the file and check/verify if the size of the array can be evenly distributed among the processes. If there’s a remainder, we can’t and essentially exit out of the loop.
* We then have to distribute the array among all the processes (size and array themselves), and receive it from the previous process. Each process calls the countElems function to sort the designated portion of the array

The main function then finalizes the mpi environment and prints in process 0 that the program is completed (this was just so we had verification to the user that it is terminated and so we had a proper message).

The **countElems** function was declared with the following parameters:

* (int \*arr, int size, int rank, int world\_size, char filename[])
  + Array itself, size of the array, rank, number of processes, and the file name

The given function declared starts with declaring MPI\_Request request and MPI\_Status status to manage communication between the processes. The program then calculates the number of elements that can be sorted by each process given the size of the array and the number of processes along with the start and ending index of the portion of the array each process is sorting based on the rank.

Afterwards, I then dynamically allocated an array called count that can hold a MAX of 1000 elements (not necessarily always going to hold that much) and looped through the array and continuously added/initialized 0 for each index of the array. Now, the process iterates over its portion of the array called arr and counts the occurrences of each element in the count array; effectively it is a representation of the frequency distribution of the elements in the portion of the array which is being sorted by each process.

* If the process is currently 0, we send the count array to the next process (process 1) using MPI\_Isend, wait for the count from the final process with MPI\_Wait() and then sort the array based on the received count array:
* Otherwise, we receive the count array from the previous process, update their count array by adding the received count array, and send it also to the next process (last process sends it back to process 0)

We then return NULL since we are not returning anything

**Problem 2**

This program is designed/was implemented to calculate the approximate value of pi with parallel processing by utilizing pthreads. In this program, I introduced a variety of libraries but pthread.h is what allows us to run the program as specifically intended with multithreads. Of course math.h allows us to find the absolute difference of the calculated pi and exact value of pi, stdlib.h allows us to convert the command-line arguments to integers (# of threads and # of terms), and stdio.h for the output.

In this program, I declared a structure called thread\_data that contains various information: the number of threads, an array of IDs, the total number of terms, and an array that stores the partial approx. of pi. Afterwards, I declared pthread\_mutex\_t that is absolutely required for thread synchronization and is for controlling access to shared resources and ensures that only one thread cana access the resource at a time and prevents race conditions and ensures the integrity of data. According to (<https://www.ibm.com/docs/en/aix/7.2?topic=p-pthread-mutex-initializer-macro>) this initializes a static mutex with default attributes and that the mutex is initialized at compile-time. With this, you don’t need to explicit define initialization functions like pthread\_mutex\_init(). I also used a counter to keep track of the current thread being processed.

Main Function:

* Within the main function: I start off by verifying the number of command-line arguments that were passed to the program (as it requires exactly 3). If 3 arguments were NOT passed, I notify the user with the format that’s required and exit the program. I then declare a variable called PI that holds the EXACT value of PI. The program then dynamically allocates memory for a structure called that’s of the structure previously mentioned (thread\_data) called data. This essentially hold the data needed for each thread. After the dynamic memory allocation is completed, it then converts argv[1] and argv[2] from a string to an integer using atoi() for the threads and number of terms respectively for the calculation (approximate) value of pi. I then used malloc that would create an array to store the approximations by doing the following:
  + data->pi\_approx = malloc(data->num\_threads \* sizeof(long double));
* The program then uses a line of code that dynamically allocates memory for an array of pthreads and another line to create the array of thread numbers; which will be assigned a unique identifier as shown in the for loop. After assigning a unique id for a thread, it creates the thread with pthread\_create which runs the CalcPiAprox function with (void \*)data as its argument.
* Afterwards, we wait for each thread to finish by creation another loop where we wait for each thread to finish before proceeding. The function pthread\_join is what waits for the thread before continuing.
* Essentially, now it’s just a matter of using a loop that runs based on the number of threads that goes through the data->pi\_approx array and finds the sum and adds it to the varialble pi\_approx that’s specific to main and outputs the approximate value of pi, exact, and the absolute difference between them

CalcPiAprox Function:

* This function is designed to find the partial sum of the approximation of pi and it successfully does so
* It’s important to declare a void function that takes an argument called (void \*[someArgName]) so it can take a void pointer called [whatever you call it], in my case; it takes a void pointer called threadarg for simplicity sake. In the program, it uses the data -> id[i] = I in the main function to take threads from 0 -> # of threads – 1. This function retrieves the thread number from the ‘thread\_data’ structure by using counter as its index.
  + long thread\_num = ((thread\_data \*)threadarg)->id[counter]; 🡪 Keeps track of the thread that is currently executing. After retieving it, the counter is incremented for the next thread.
* Afterwards, we have to retrieve the total number of terms (using ‘n’) and then similarly; the total number of threads from the thread\_data struct we declared. These are done via:
  + long n = ((thread\_data \*)threadarg)->n; 🡪 Gets number of terms
  + int num\_threads = ((thread\_data \*)threadarg)->num\_threads; 🡪 Number of threads
* It’s then important to retrieve the array which stores the partial sum approximations
* Afterwards, we then essentially begin calculating the partial sum approximation of pi for the terms assigned within the current thread and iterate over a subset of terms based on the thread #. The approximation is done using the Monte Carlo method with the for-loop and then the formula that was shown in the slides of MPI.
* This overall allows us to calculate the partial sum approximation of pi for the assigned subset of terms and gives us the specific result that is calculated; which changes based on the number of terms (threads increase the speed of the calculation).