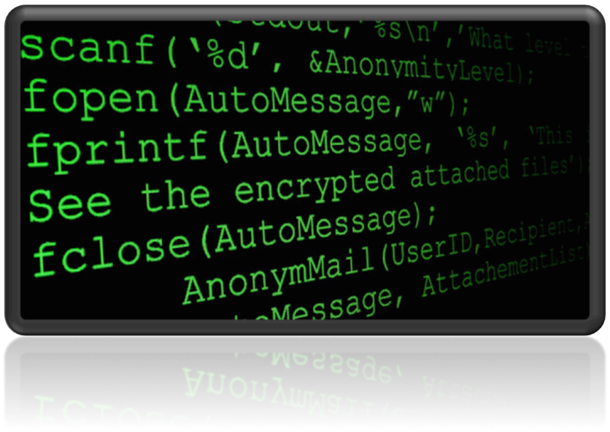
Comp 428/6281
Parallel Computing

Assignment Three


scanf('%d', &AnonymityLevel);
fopen(AutoMessage,"w");
fprintf(AutoMessage, '%s', 'This
See the encrypted attached files')
fclose(AutoMessage);
AnonymMail(UserID,Recipient,
AutoMessage, AttachementList

Due date: 11:55 PM, Tuesday, November 10.

Note: All submissions are made via Moodle (see "deliverables" section).
Late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied automatically.

After you upload the assignment, please verify that your submission is recorded as submitted.

Note: Graduate students will be required to solve additional problems and/or provide additional software components. These additional components will be clearly indicated in the description below.

Part A: Debugging

Total Points for Undergrads and Grads: 10

In this first question, your task is simple. You will be given a small MPI program that computes a matrix/vector multiplication. The logic of the computation itself is correct (so you don't really have to understand exactly how matrix multiplication works). However, I have introduced a small error into the program that unfortunately causes it to crash. You will see this as soon as you compile and run it – there will be a whole lot of nasty messages spewing out to the screen. But again, as bad as the error looks, there is only one tiny part of the code that needs to be fixed.

So all you have to do is find the bug and provide the update to eliminate the error. In addition to the source code for the application ("COMP428A3_debug.c"), I have included a text file ("Debug Program Output.txt") that shows what the output should look like.

Part B: MPI program

Total Points for Undergads: 30
Total Points for Grad Students: 40

In this assignment, you will get a chance to use a couple of the more sophisticated features of MPI: derived data types and non-blocking calls.

The task is fairly straightforward. You will be working with a set of data structures that represents weather events – specifically, rainfall amounts for certain dates and locations. The data will be stored in an array of C structures of the following form <date, location, amount>

Date will consist of the 3 integers:

1. Year (a **short unsigned int** in the range 1970 to 2015)
2. Month (a **short unsigned int** in the range 1 to 12)
3. Day (a **short unsigned int** in the range 1 to 31)

Location will consist of 2 integers:

1. Latitude (a **short signed int** in the range -90 to +90.
2. Longitude (**a short signed int** in the range -180 to +180.

Amount: a **float** value in the range 0.0 to 1000.0

Your application will perform the following steps:

1. Each process will generate 10 thousand rainfall events. Data will be randomly generated according to the appropriate ranges described above. These events will be stored in an array on each processor.

2. Our goal is to sort the data by rainfall amount, in ascending order. You will use the following approach. Each process will first send *about* 1/p of its rainfall events to one of the other nodes. The idea is for the events with the smallest amounts to be sent to process 0, the next smallest group to be sent to process 1, etc. Since the rainfall amounts are randomly generated, it is easy to split the data more or less evenly. For example, if p = 5, all processes would send values between 0.0 and 200.0 to process 0, and values from 200.0 to 400.0 to process 1, etc. **NOTE**: this does not mean that exactly the same number of events will be sent to each node. For example, there might be 203 values between 0 and 200.0 and 198 values between 200.0 and 400.0.

3. In order to send the data, you must first sort the data on each node so that you can then use MPI to send the data to the proper destination (i.e., approximately the first 1/p segment of the array would go to process 0).

4. Furthermore, you must use derived data types to efficiently send the data directly from its position in the local array.

5. Once the data arrives at the destination node, it will be sorted again. For example, process 0 will receive all the rainfall events between 0.0 and 200.0 from each process. This will be about 2000 events per process, for a total of about 10000 events.

6. You're not quite done. This is only half of the data. You will now repeat the previous steps, generating another 10000 events on each node. However, it makes no sense to wait for the first 10000 events to be sent across the network to the target nodes before proceeding with this step. So your initial distribution of the 10000 events will be done in non-blocking form. While that send is being done, you will generate the second group of 10000 events and sort them locally.

7. You will now repeat the process of sending data to the target nodes, based on the rainfall ranges. At the end of this step, each process will have two sorted buffers.

8. Before completing our rainfall sorting program, we must combine the two sorted buffers on each node. We could do this by concatenating the two buffers together and then sorting it but this would be inefficient since we have already sorted each list. Instead, you will merge the two arrays in a single linear time pass. For simplicity, you can create a new "empty" array and then merge the two lists into this new array.

9. Finally, you will send the sorted lists back to process 0 so that they can be combined into one large sorted array of rainfall events. You do not need to sort this final array since it will already be in sorted order by process ID (i.e., P0's values will come first, then P1's values, etc.). To show that it has worked, you will print out the first 10 and last 10 events from the final sorted array. It might look like the following:

```
** Rainfall Events **

First 10:

Year    Month Day      Lat     Long    Amount

1987   06      23      +43      -143    0.26
1997   03      21      +13      +167    0.79
2014   12      07      +74      +034    0.81
2003   04      14      -12      +111    0.93
1971   07      19      +09      -001    0.98
1973   06      29      -71      -124    0.99
1982   03      31      -09      -003    0.99
1999   01      16      -22      -103    1.00
1998   03      12      +34      +098    1.04
1977   11      20      +67      +179    1.09


Last 10:

Year    Month Day      Lat     Long    Amount

1977   07      20      +13      -143    999.05
2008   05      17      +23      +167    999.09
2015   10      09      +89      +034    999.22
2011   12      12      -10      +111    999.34
2006   01      29      +09      -001    999.34
```

```
1981    01      22      -79     -124    999.36
1989    02      30      -82     -003    999.45
1973    08      07      -06     -103    999.87
2014    11      10      +32     +098    999.88
1989    10      24      +17     +101    999.98
```

And that's it.

Grad version. Graduate students will follow the same general application logic. However, the sorting of data will be slightly more complex due to the fact that the rainfall amounts will be heavily skewed (the other data values can just be randomly generated).

Creating useful skew patterns can be somewhat tedious. Since this isn't the point of the assignment, I will provide a skew generator for you. It will produce values in the range 0.0 to 1000.0, with numbers skewed towards the high end of the range. Each time the application is used, it will produce a slightly different degree of skew. The code for the skew generator is in a file called **zipf.c**. You can run that program directly to see how it works. For your application, you can just use the logic in the main method of zipf.c to produce your skewed values.

Because of the skew pattern, you cannot simply send regular ranges of data to each node. For example, 90% of the rainfall amounts might be between 800.0 and 1000.0. As such, you need to determine the ranges that should be sent to each node. For example, the ranges could be 0.0 – 506.0, 506.0 – 818.0, etc.

To do this you will use a two-step mechanism. On each process, you will sort the vector of events by rainfall amount. You will then take (p * 10) sample amounts from the list at even intervals. In our case, for p = 5 this would be 50 samples per process, at locations 0, 200, 400, 600, etc).

Each process will then distribute its list of samples to process 0. It will receive 250 samples if p = 5. Process 0 will then sort the samples and select ***p*** elements from the list to serve as the final split points. In our case, this would be the samples at position 0, 50, 100, etc. The idea is that these will give a good estimate of where the split points should be for the whole data set.

Process 0 will now distributes these split points back to the other process who can now proceed to distribute the rainfall values to the proper nodes.

There is one final thing to keep in mind. When you send the sample amounts to process 0, you will just send the rainfall amounts themselves, not the whole rainfall events. To do this, you must use a second derived data type that can extract the rainfall amounts directly from the rainfall event array.

And that's it.

Running the code. It is expected that the code will be tested on 5 processes. Of course, it should also run on 2, 3, or 4 processes without crashing.

Deliverables:

Part A: For the Debug questions, you will simply submit a fixed version of the COMP428A3_debug source file.

Part B: The main source file should simply be called A3.c. Yu can use other source files as required.

You will also need a `makefile` to compile your code. If you are using PTP, then you can use the Managed Makefile approach and allow the PTP plugin to produce the `makefile` for you.

You should also include a README text file to indicate what parts of the programming assignment work or don't work. This may make it easier for the grader to give you points if there are problems/limitations with your solution.

Final Submission: All files should then be combined into a single zip file as indicated below and submitted to Moodle.

LastName_FirstName_A3.zip **(e.g., Smith_John_A2.zip)**