# Final Project

CS 484 - Parallel Programming

May 4, 2020 @ 23:59

( Typeset on : 2020/01/19 at 14:44:23)

# Introduction

## Learning Goals

You will learn the following:

- Histogram sorting in a distributed memory environment
- Hybrid OpenMP + MPI parallelization and performance analysis

# Assignment Tasks

The basic workflow of this assignment is as follows:

- Clone your turnin repo to VMFarm [1]
  Your repo should be visible in a web browser and clonable with git from
  `https://gitlab.engr.illinois.edu/sp20-cs484/turnin/YourNETID/sortproj.git` .
- You may need to iterate:
  - Implement the algorithms / code for the various programming tasks.
  - Build and test on VMFarm.
  - Check in and push your complete, correct code.
  - Benchmark on Campus Cluster (`scripts/batch_script.run`)
- Check in any benchmarking results that you wish to and final versions of code files.
- Write and check in your writeup.

# 1 Distributed Memory Histogramming Sort

For this project, you should use MPI as well as OpenMP. You are free to choose how to combine the two for performance, but you should at least have as many MPI ranks as the number of physical nodes used in benchmarking.

You will start with data consisting of unsigned 64-bit integers, whose type is defined as `dist_sort_t`, that is distributed non-uniformly across MPI ranks. You can find the function prototypes in `solution.h`, and corresponding function bodies in `solution.cpp`. You may also want to refer to `main_sort.cpp`, where the overall execution flow of the program is defined.

You will first balance data by redistributing it across processes without worrying about sorted-ness, and then sort it in parallel using all MPI ranks with a repeated histogramming algorithm. At the end, each MPI rank should hold data that is locally sorted, and any data on rank `i` should be smaller than the smallest data item on rank `i+1` (i.e. data should be globally sorted). The entire set of data items, including any duplications, must remain the same.

## 1.1 Rebalancing Data

Your first task is to balance data across all the ranks without worrying about sorted-ness. This should be implemented in the `rebalance()` function. The first two parameters are the original data array (randomly generated) and the number of elements in the array. The last two parameters are pointers to the array that will contain the rebalanced data and the number of elements that will be in the rebalanced array. This function should be implemented such that all ranks have roughly the same number of data elements, where each data element originates from one of the imbalanced arrays (the total dataset should remain the same).

For a precise description of pre and post conditions, see the comments in `src/solution.h` .

[1] You are free to use the Docker container uiuccs484parallelprog/cs484_student , but we will provide no support for Docker itself.

### 1.2 Histogramming

You will implement the `findSplitters()` function which attempts to find a histogram with (roughly) equally counts in its bins by varying bin boundaries. You will use the repeated histogramming algorithm. There will be N splitters for N bins, with the last splitter being the maximum value in the entire dataset. Do not assume that N will be equal to the number of ranks, it may not be. After calling `findSplitters()`, the values in `splitters` shall be monotonically increasing, `counts[i]` shall contain the global number of elements in all the data arrays such that `splitters`$[i-1] < x \leq$ `splitters`$[i]$. That is:

$$\texttt{counts[i]} = |\{x| \quad \texttt{splitters}[i-1] < \quad x \leq \texttt{splitters}[i], x \in \texttt{DATASET}\}|$$
$$\texttt{counts[0]} = |\{x| \qquad\qquad\qquad\quad x \leq \texttt{splitters}[0], x \in \texttt{DATASET}\}|$$

over the entire, distributed dataset.

You should read the paper [1] for more details on the iterative process of histogramming and finding the splitters. For those of you that would like to optimize the algorithm even further, we recommend reading [2]. Both papers have been uploaded to the Wiki, in Week 13 of the Lecture Schedule page.

This problem is much simpler if, internally, you consider it as a problem of finding equally spaced quantiles, rather than equally weighted bins. Nevertheless, output must conform to this specification. (It is easy to transform between the two problems.)

For a precise description of pre and post conditions, see the comments in `src/solution.h` .

### 1.3 Moving the Data Elements

Finally, you will implement the `moveData` function that utilizes the `splitters` and `counts` arrays to send the data elements to the right ranks. At the end, each machine should have numbers that fall in the half-open interval

$$(\texttt{splitters[i-1]}, \texttt{splitters[i]}]$$

, with the zeroth rank holding all data in the closed interval

$$[0, \texttt{splitters[0]}]$$

. The final elements should be placed in `*recvData` (you should allocate memory for it and store the new pointer) and the number of elements in `*rDataCount`.

For a precise description of pre and post conditions, see the comments in `src/solution.h` .

### 1.4 Local Sorting

You have been provided a correct, working, single-threaded sort algorithm in `sort()` . You may replace this with your own sort implementation if you feel you can do better.

*Note that while no points will be awarded for the correctness of your sort(), points will be deducted for an incorrect sort() function.*

## 2 Benchmarking

The project will be graded on correctness as well as the performance of each phase, and good use of OpenMP and MPI primitives. You may create and test alternate implementations of some or all phases using competing MPI primitives. If you choose to do so, your writeup should compare the different approaches.

More specifically, once histogramming has identified the correct splitter keys, data needs to be moved from every process to every other process to where it belongs. For this purpose, you may use the following three alternatives (or more if you can think of any), and compare them:

- All-to-all collectives
- Regular point-to-point messages
- One-sided messaging (using `MPI_Put`, etc.)

Whether or not you choose to explore all the options, your final program should choose one, and you should highlight that in your writeup.

We have provided you with a batch script perform unit tests of your implemented functions and time your program. **This time, however, make sure to modify or add lines with `s_mpirun` in the benchmarking part of the script if necessary, according to how you mix MPI and OpenMP.** The current script assumes that you are not using OpenMP and are allocating 1 MPI rank per CPU core. As always, you should run this on the campus cluster. The timing result will be stored in `writeup/benchmark.txt`.

You should vary the number of CPU cores and physical nodes used in the program and plot these results, evaluating the performance and scalability in your writeup.

# 3 Submission

Your writeup must be a **single PDF file** that contains the tasks outlined above.

*Please save the file as "*`./writeup/proj_<NetID>.pdf`*" , commit it to git, and push it to the* **master** *branch of your turnin repo before the submission deadline.*

You must also commit at least the following code files. These files, and only these files, will be copied into a fresh repo, compiled, and tested at grading time.

- `src/solution.h`
- `src/solution.cpp`

Nothing prevents you from altering or adding any other file you like to help your debugging or to do additional experiments. This includes the benchmark code. (Which will just be reverted anyway.)

It goes without saying, however, that any attempt to subvert our grading system through self-modifying code, linkage shenanigans, etc. in the above files will be caught and dealt with harshly. Fortunately, it is absolutely impossible to do any of these things unaware or by accident, so relax and enjoy the assignment.

# References


[1] L. V. Kale and S. Krishnan, "A comparison based parallel sorting algorithm," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 3, pp. 196–200, Aug 1993.

[2] V. Harsh, L. V. Kalé, and E. Solomonik, "Histogram sort with sampling," *CoRR*, vol. abs/1803.01237, 2018.