The Implementation of an MPI Algorithm for the Analysis of Global Precipitation Data

Aaron Fainman (1386259), Nathan Jones (1619191) & Taliya Weinstein (1386891) ELEN4020
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg South Africa

***Abstract*—The design and implementation of an MPI algorithm for the analysis of precipitation data are presented. Specifically, the algorithm finds a five-point summary of the data. Two algorithms have been designed, though only one was fully implemented. The first algorithm works by distributing the data to different processes, each of which uses a merge sort to order the data set. Since the data set is very large, the files are read in batches, filtered, and then sorted by each process. Between each batch, the sorted data is distributed to processes to store. After all batches have been processed, the zeroth process finds the position of the quartile values in the stored data and the necessary process sends the value. The second algorithm works by repeatedly finding the median of sorted arrays containing the elements of the 5-number summary from each process. Only the merge sort approach was implemented in practice. Testing took place on the University of the Witwatersrand's Jaguar cluster. Results proved that the algorithm is accurate but computationally inefficient. 90% of the computation time is spent reading and filtering the data, serially. This overshadows any trends that could be seen in how the merge sort scales with array size, or how computation time is affected by the number of processes. In the future, the data should be read and filtered in a distributed manner.**

I. Introduction

The analysis of climate data is a common application for parallel programming [1]. One such data set is the Global Precipitation Climatology Centre's daily precipitation data [2]. The full data set consists of daily rainfall in a region for every day of the year since 1891 [2]. The spatial resolution of the data ranges from 0.25º to 2.5º in both latitude and longitude. Analysing this data is no trivial task. Each year's worth of data is a few gigabytes in size [2]. However, proper analysis of this data provides the opportunity to glean insight into climate change and the state of the environment. This paper will investigate the design and implementation of a message-passing interface (MPI) algorithm for the analysis of the GPCC's climate data.

II. Project Background

A. Box and Whisker Plots

In statistics, a box and whisker plot graphically represents a *five number summary* for a set of data [4]. These values are used to assess how the data is distributed and are comprised of the minimum and maximum values, as well as the values at 25%, 50%, and 75% of the way through a data set in ascending order. In other words, given an ordered data set of size n, the first quartile (Q1) is found at the position $\lceil n/4 \rceil$, while Q2 and Q3 are found at positions $\lceil n/2 \rceil$ and $\lceil 3n/4 \rceil$ respectively. Unlike the mean, these measures of central tendency have the advantage of being nonsensitive to skewed data or outliers [4].

B. Algorithm Options for 5 Number Summary

The most direct approach to finding the required elements for a box and whisker plot is to sort the supplied data and then utilise the positions of the defined elements to extract the required values. There are many algorithms for sorting data. The simplest method is a selection sort. This compares two neighbours and swaps them, if the second value is greater than the first [5]. While simple, selection sort has a time complexity of $\mathcal{O}(n^2)$ [5]. A more complex algorithm is a quick sort. Here, data is compared to a pivot selected from the dataset. Data values smaller than the pivot are sorted to one side of the array and data values larger are sorted into the other [5]. The two sides of the array are then separated and a new pivot is selected. This continues until each sub-array is sorted [5]. Quick sort has an average complexity of $\mathcal{O}(nlog(n))$ and a worst-case complexity of $\mathcal{O}(n^2)$ [5]. Merge sort works by dividing data into smaller chunks, sorting each chunk, and then merging them back together. Merge sort has an average complexity of $\mathcal{O}(nlog(n))$ [5]. Some algorithms can do better than $\mathcal{O}(nlog(n))$ in their complexity. Radix sort has a complexity of $\mathcal{O}(kn)$, where k is some integer [5]. The algorithm works by sorting the data into buckets based off of the data's digits. First, the units are sorted, then the tens place, and so on until the largest digit of the algorithm is placed into a bucket. However, radix sort does not easily translate into float values. First, a conversion needs to take place to convert the float value, into an integer equivalent (such as the IEEE 754 representation) [6].

However, the process of sorting a large quantity of data for the extraction of 5 numbers is an unnecessary effort in most instances. quickselect is an alternative algorithm that is able to identify the k_{th} smallest element in an unordered list. It is important to note that, although similar, quickselect is not the same as quick sort. The algorithm works by comparing a randomly chosen pivot from the data set to the rest of the unordered data thereby forming three distinct groups: values greater than the pivot, values less than the pivot, and values equal to the pivot. The required k_{th} smallest element position is then compared to the size of each of the three groups to determine which group is most likely to contain the data value. This group subsequently becomes the new unordered dataset whereby the selection algorithm recursively operates. A new pivot is chosen in each iteration until only a single value remains - representing the desired element in the k_{th} position [7].

The quickselect algorithm has a time complexity of $\mathcal{O}(n)$ if the pivot is situated within the middle of the data set and $\mathcal{O}(n^2)$) if the selected pivot is situated at either end of the dataset. Ultimately, the best pivot to choose is the median as it enables the greatest elimination of data points in each step [8]. The median of the median algorithm is generally utilised to ensure that the median value of the dataset is selected as the pivot. This algorithm breaks the data into fixed batches and then finds the median of each batch. The median values of each batch are then split again to find the medians of a fixed number of batches. This continues until only the estimate median remains [7].

C. Distributed Computing Limitations

The sorting algorithms found are usually described as serial or shared parallel algorithms. While distributed computing has many advantages, it does introduce some complexity into the structure of the algorithms. At the beginning of the program, only one process (usually rank 0) has access to the data. This process then needs to redistribute the data to be sorted and be received back. Careful attention must be paid to how often communication occurs. This is often a bottleneck in MPI programs [1]. Additionally, it is desirable to have as much of the work parallelised as possible. This avoids the processes waiting idly, for a serial piece of code to finish [1]. For these reasons merge sort has been implemented. For starters, the sorting algorithm is able to deal with float values. Additionally, the size of the chunks of data is split based on the amount of data, and not the data itself. Quick sort splits data based on a pivot chosen from the data set. A poor choice of pivot can lead to poor load bearing in the algorithm, worsening computation times [9]. Merge sort's fixed batch sizes make it preferable in this respect.

Each step of the quickselect algorithm poses a performance challenge to its parallel implementation [10]. The pivot selection step requires communication between the processes to ensure that all processes are using the same randomly chosen pivot. This pivot selection occurs in each iteration of quickselect. Since pivot communication between different processes would require synchronization - the algorithm's performance would be impacted [11]. Additionally, synchronization performance impairment considerations are also prevalent once each process has finished organising its assigned data according to the pivot. Each of the three organised groups would need to be sent back to the coordinating process, merged, and subsequently compared to the desired k_{th} element. This sorting, together with the recursive nature of the algorithm, introduces substantial serial sections into the overall parallel implementation [10]. According to Amdahl's law, the speedup of any parellilized program is limited by the amount of serial code included in the program [12]. Thus, due to quickselect's high volume of serial code, it is ruled out as a viable algorithm for generating a scalable solution for producing a box and whisker plot.

III. Design Context

A. Requirements and Success Criteria

The goal of this project is to implement a parallel algorithm that makes use of the MPI library to find the five-number summary of the rainfall data for a given range of years within the GPCC database. Success will be defined as an algorithm that is both accurate and scalable.

B. Assumptions

It has been assumed that days or locations where no data readings were made do not contribute to the calculation of the five-number summary, and have thus been ignored. The issue of "zero" precipitation has been dealt with in two ways by giving the user the choice to include or ignore these values. Including these values results in a box plot that provides a holistic view of global rainfall, while ignoring them results in a box plot that indicates the spread of rainfall data across the globe *only when it does rain in each area*. Finally, for the sake of simplicity, in the case that one of the quartiles falls between two data points, the first of these points is chosen rather than taking the average of both points. This is justified by the fact that with very large data sets, these two values are unlikely to differ by any significant amount. Outliers are assumed to be any value outside 1.5 times the interquartile range above and below the median [4]. These have been dealt with by displaying the non-outlying maximum and minimum, and the outlier maximum and minimum. The non-outlying maximum and minimum are taken as exactly 1.5 times the interquartile range above and below the median respectively. Since the data set is large it is assumed the actual non-outlying maximum and minimum are very close to this value.

C. Constraints

The primary constraint in meeting the success criteria was the limited time available to design, implement, and test our solution. This was due to the pressures of other university projects and the limited computing resources of the Jaguar1 cluster that needed to be shared amongst all groups. In addition, all group members were unfamiliar with the MPI library as well as other software tools such as SLURM and CMake. This limited the extent to which our design could be optimised, as the process of compiling and running our solution on the cluster had its associated learning curve. Finally, we were restricted to using open-source software solutions and the NetCDF data format used to store the GPCC rainfall data.

IV. Implementation

A. NetCDF Data Reading

The GPCC rainfall data is stored using the Network Common Data Form (NetCDF). This file format is used to store large multidimensional arrays of scientific data and is commonly used in climate and other geographical applications [13]. The netCDF-4 C++ library was used to read this data into our program. Developed by Lynton Appel of the Culham Centre for Fusion Energy (CCFE) in Oxfordshire [14], this library provides an object-oriented API built on top of the standard netCDF-4 C library. The `FileReader` class encapsulates the IO functionality of the program and is responsible for opening, reading, and storing the NetCDF files as chosen from the command-line input. In addition, the function `filterData` removes non-data entries, as well as zero-entries if the `filterZeros` flag is set by the user.

B. Algorithms

1) Sorting Algorithm: The sorting algorithm implemented is based on a merge sort. Process 0 (the process with MPI rank 0) reads in the data. Reading in is done in chunks. Process 0 then calculates the smaller batch sizes to distribute to all other processes. The sizes are within 1 unit for all processes. MPI's `Scatterv` is then used to distribute the batches to all other processes. A typical merge sort would continue to divide the processes up. However, within each algorithm, the single batches are sorted. This is done using C++'s `std::sort` algorithm. Data is sorted in-place with an average time complexity of $\mathcal{O}(nlog(n))$. Processes are then grouped in pairs. One process sends their data and the other receives and merges it into its dataset. Multiple iterations of this merge occur until all the data ends up back in process 0.

The data is then distributed to every other process to store. On the first reading of the file, process 0 divides the sorted data equally among the processes to store. It also stores the maximum element that each process stores. Thereafter, the data is distributed by only sending each process value up to its maximum. The process receives the data and merges it into its stored data. This means that over many iterations of file reads, the data will always be stored sorted. It also means different processes should store similar amounts of data (assuming the distribution of data in the first file read is representative of the file as a whole). There is one exception to this. Process 1 always stores 0s in the data (if 0s are not excluded). There are often a large number of 0s in the data set and having a single process store this data makes for simpler computations. Finding the box-and-whisker plot values can occur once the entire NetCDF file has been read. Process 0 stores how many elements it has distributed to each process. Based off of this, it calculates where the first, second, and third quartiles are stored. A broadcast takes place of which rank should send data to process 0 and the index thereof. Pseudocode for the algorithm is shown in Figure 1.

```
N = Number of file reads
While N
    if Process==0 read data
    if Process==0 Calculate batch sizes
    Scatterv batches of data from process 0
    Sort data
    while data not merged
        if Process % merge number==1
            send data to rank below
        if Process % merge number==0
            receive and merge data from rank above
        ++merge number
    end
    if Process==0 distribute sorted data
    else  received and merge sorted data for storage
end
if Process==0
    calculate where quartiles lie
    broadcast which process should send data
else receive broadcast
    send data to process 0 if required
if Process==0 Output data
```

Fig. 1: Merge Sort Algorithm

2) Median of Medians Algorithm: In an attempt to design an optimal 5-value summary for parallel implementation, a modified version of the median of medians pivot selection algorithm is designed. As demonstrated in Figure 2, this algorithm distributes the incoming data to the available processes which then sort the data into ascending order. Each process calculates its own Q2, Q1, Q3, minimum and maximum and returns these values to process 0. Process 0 then separates each of the summary data into its own array (i.e: all Q1s from all processes are in one array), sorts the values into ascending order, and calculates an overall 5-number summary from all the processes by finding the median of each of the generated arrays. This occurs for all batches of data making up the requested data stream. In the last batch of data - the composite arrays of all previously calculated values of Q1, Q2, Q3, minima, and maxima in each of the batches are then sorted in ascending order again with the final value for each of the 5 numbers being found through finding the median of each array.

If an odd quantity of data is sent through, then the true median is found. Yet, if an even amount of data is sent through then the lower value of the central 2 values is taken as the median. The algorithm also accounts for outliers by checking whether the overall absolute value of the minima and maxima, after taking the median of the final array set, are greater than the defined limits given in Equation 1 and Equation 2. If the calculated values exceed these bounds then the bounds are used to represent the outlier value.

$$lowerOutlierLimit < Q1 - 1.5 \times (Q3 - Q1) \quad (1)$$

$$upperOutlierLimit > Q3 + 1.5 \times (Q3 - Q1) \quad (2)$$

```
A = 1DArray
N = Overall datasize
d = data sent in each batch stream
While N
    for i = 0 to N/?
        Send ± equal data batch sizes, B, to each process
        y = Sort(d)
        values = [y[0], y[B/4], y[B/2], y[3*B/4], y[max]]
        Send v to process 0
        if Process == 0
            Receive values from other processes
            Order values from processes into 5 arrays
            Find the median of each of the 5 arrays
            Store medians of 5 values in new arrays
        end
        end
    Find the overall median of stored values in each Array
    if Max and Min are within appropriate bounds
        Use calculated max and min
    else
        Max = Q3 +1.5*(Q3 - Q1)
        Min = Q1 -1.5*(Q3 - Q1)
    end
end
```

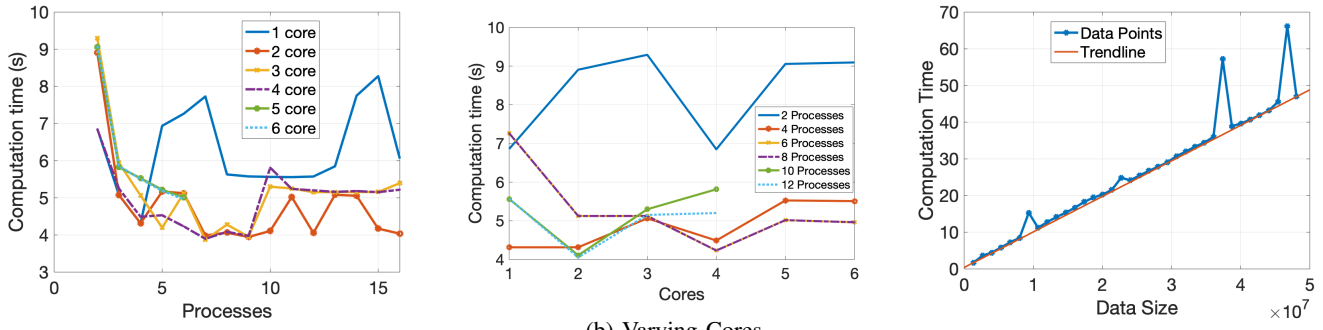
Fig. 2: Median of Medians Algorithm

Overall, this algorithm trades off the accuracy of the 5-number summary for simplicity of computation and reduced communication between processes. A potential bottleneck in performance could be the large amount of serial code required to be performed by process 0. Due to time constraints, this algorithm was unable to be fully implemented and subsequently was not able to be tested.

V. Testing and Results

A number of different tests were performed to investigate the algorithm designed. These were run on the University of the Witwatersrand's Jaguar cluster.

To check the accuracy of the results the algorithm was run on a subset of the NetCDF data, *full_data_daily_v2020_10_2019.nc*. The size of this data is 284 Mb and is small enough to be handled by a personal computer. MATLAB was used to check the algorithm's output against the actual output. Checking in MATLAB consisted of reading and filtering the data using MATLAB's built-in NetCDF functions, ordering the data using MATLAB's `sort`



(a) Varying Processes (b) Varying Cores (c) Varying Data Size

Fig. 3: Results from testing the algorithm's computation time. Figure (a) shows the computation time against the number of processes, for a constant number of cores and constant data size. Figure (b) shows the computation time against the number of cores while keeping the number of processes and data size the same. Figure (c) shows the change in computation time while varying the data size for a set number of cores and processes (2 and 8 respectively). Clear trends are difficult to infer from Figures (a) and (b). This is likely because the code is predominantly serial.

TABLE I: Average Computation Time (% total) for Different Sections of the Code

Code Section	Average Time (%)	Code Type
Reading Data	39.39	Serial (process 0)
Filtering Data	52.38	Serial (process 0)
Scattering Data	0.21	Communication
Sorting Data	0.76	Distributed
Merging Data	6.04	Distributed
Distributing data for storage	1.17	Communication
Analysing sorted data	0.09	Serial (process 0)

function, and finding the values at the relevant indices for each quartile. The MATLAB and the algorithm's output gave the exact same set of numbers: 0.01, 0.01, 0.14, 0.77, 3.43, 5.705, 562.59 mm/day for the absolute minimum, non-outlier minimum, quartile 1 value, quartile 2 value, quartile 3 value, non-outlying maximum, and absolute maximum.

The next set of tests looked at the computation time. The time was measured while adjusting the number of processes that MPI ran on (for a set number of cores), the number of cores (for a set number of processes), and the time when the data size to be processed was varied. These results are shown in Figure 3 Additionally, the average length of time of each section of code was measured. This average has been performed across core sizes varying from 1 to 6, and process sizes varying from 2 to 16. In total the average consists of over 50 tests. This is shown in Table I.

VI. Discussion

The graphs in Figures 3a and 3b, do not show a clear trend in how the algorithm scales with processes and processes on multiple cores. It is clear that 3 processes have better computation times than 2 processes. Not much more can be inferred from the two figures figures. Looking at the computation times in Table I shows that over 90% of the computation time is spent reading and filtering the data, a serial component of the code. This explains why there is no clear trend in the number of processes because the bottleneck in computation time is the serial aspect of the algorithm.

In fact, the actual substance of the algorithm is untested. From Table I, certain inferences can be made. The merge time takes the longest, followed by distributing the data for storage. However, these times pale in comparison to the length of time required to read and filter the data.

The decision to read and filter the data using a single process was a bad choice. However, the reason for it was to prevent processes from receiving only missing data or zeros. This is very possible considering these make up a large portion of each data set. Additionally, the thinking was that each process would receive the same amount of work to do if the data set was initially filtered leading to better load sharing in the sorting and merging sections. While this may be true, the code is obviously mostly affected by the poor load sharing in the reading and filtering sections.

The graph in Figure 3c shows the serial reading and filtering is the reason the computation time scales linearly with the amount of data. Unfortunately, the algorithm has not implemented an $\mathcal{O}(n)$ sorting algorithm. Rather, the likely $\mathcal{O}(nlog(n))$ computational complexity in the merge sort is being overshadowed by the very lengthy, but linear, reading and filtering.

This is not to say the work performed does not have value. For starters, the algorithm can accurately find the necessary values for the box-and-whisker plot from a large data set. As computationally inefficient as it may be, it is accurate. The bottleneck in the reading and filtering could be refactored and better distributed. This could be done without affecting the majority of the code written since reading and filtering have been implemented in their own class. Finally, the work serves as a good case study for Amdahl's law, which shows that the speedup for parallel programs is limited by the serial portions of code [12]. Regardless of the number of processes run on the cluster, the computation time is largely unaffected.

VII. Ethical and Social Implications

Recently, there has been an increase in concern raised for the environment, alongside calls for decisive leadership in combatting climate change [15]. Access to good information is critical in making impactful climate change decisions. The use of the Global Precipitation Climatology Centre's precipitation data set and parallel programming techniques is one way to obtain such information. It is hoped that exploring these data

sets, and data-intensive computing in general, will allow for more effective decisions around climate change.

VIII. Future Improvements

The obvious improvement in the code is to refactor it so that the reading and filtering of data is no longer serial, but rather distributed amongst processes. It is expected that this will have a major improvement on the code's computation time.

The merging of data in the merge sort appears to be the next major bottleneck in the algorithm. This is not unexpected, considering that merging involves two steps: communications between processes where data is sent off and received, and merging of the received sorted data into a process's own sorted data. This happens for every iteration of the file read. It would be worthwhile exploring ways of reducing this time. For example, it may be better to merge once at the end of all the file reads.

A hybrid OpenMP-MPI approach may also be useful to explore. This would make use of MPI to coordinate work between processes, and the shared parallel library OpenMP for parallelising within a process. The use of OpenMP could reduce the computation time within a process.

Finally, due to the potential of the Median of Medians algorithm to reduce communication between the processes, which could improve performance, it is recommended to investigate this algorithm further.

IX. Conclusion

Two algorithms have been designed for analysing NetCDF precipitation data using MPI. The first algorithm works by sorting the data, using a merge sort, and then finding the indices of the desired quartile values. The second algorithm works by repeatedly finding the median of sorted arrays containing the elements of the 5-number summary from each process.

The algorithm meets the first success criterion of being accurate. However, it fails in scalability. The problem is not with the algorithm itself, but in the way data has been read and filtered. Most of the algorithm's time is spent in this portion of the code. This was clear in the linear scaling of computation time with array size. This was also the reason why there was no clear change in computation time with a number of processes, a demonstration of Amdahl's law, and the limits of computation speedup in programs with too much serial code. The reading and filtering only occurred in a single process. Distributing this task would likely result in major improvements in computation time. The merge sort was shown to spend a lot of its time in the merging of data. This step involved one process receiving data from a neighbouring process, and then merging it into its own data. In the future, it would be worth exploring whether to merge only at the end of all the data reads.

References

[1] P. Pacheco, 'Parallel Hardware and Parallel Software', in An Introduction to Parallel Programming, Burlington, MA: Morgan Kaufmann Publishers, 2011, pp. 1–10.

[2] Schneider, Udo; Becker, Andreas; Finger, Peter; Rustemeier, Elke; Ziese, Markus, "Gpcc full data monthly product version 2020," 2020, https://opendata.dwd.de/climate

[3] _environment/GPCC/html/fulldata-monthly_v2020_doi_download.html (accessed May 10, 2021).

[4] J. Jones, "Stats: Measures of position," 2020, https://people.richland.edu/james/lecture/m170/ch03-pos.html (accessed May 6, 2021).

[5] J. Paton, "Sorting," http://pages.cs.wisc.edu/ paton/readings/Sorting/ (accessed May 06, 2021).

[6] P. Tardiman, "Radix sort revisited," http://codercorner.com/RadixSortRevisited.htm (accessed May 06, 2021).

[7] D. Panigrah, 'Design and Analysis of Algorithms - Lecture 3'. Duke University, Durham, North Carolina, 2014.

[8] A. Desai, '"Median of median" on medium', Medium, Mar. 09, 2019. https://medium.com/@amit.desai03/median-of-median-on-medium-5ed518f17307 (accessed May 15, 2021).

[9] D. R. Ramkumar, "Implementation of parallel quick sort using mpi," course notes for CSE633, https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ramkumar-Spring-2014-CSE633.pdf (accessed May 06, 2021).

[10] C. Siebert, 'Scalable and Efficient Parallel Selection', in Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part I, vol. 8384, R. Wyrzykowski, J. Dongarra, and J. Wasniewski, Eds. Berlin Heidelberg: Springer, 2014, pp. 202–213.

[11] P. Pacheco, 'Distributed-Memory Programming with MPI', in An Introduction to Parallel Programming, Burlington, MA: Morgan Kaufmann Publishers, 2011, pp. 83–134.

[12] P. Pacheco, 'Parallel Hardware and Parallel Software', in An Introduction to Parallel Programming, Burlington, MA: Morgan Kaufmann Publishers, 2011, pp. 29–70.

[13] K. Hafen, "Read netcdf data with python," *Towards Data Science*, 2020.

[14] L. Appel, 'netcdf-cxx4', GitHub, May. 10, 2021. https://github.com/Unidata/netcdf-cxx4.

[15] A. Freedman, "More than 11,000 scientists from around the world declare a 'climate emergency'," https://www.washingtonpost.com/science/2019/11/05/more-than-scientists-around-world-declare-climate-emergency/ (accessed May 12, 2021).