An optimization-inspired approach to parallel sorting

Team Metropolis:
Jamshid 'James' Farzidayeri, JJ Lay, and Graham West

COMS 7900, Capstone

Abstract

We present a novel method influenced by ideas in the field optimization to efficiently sort large amounts of data in parallel on a cluster of computing nodes. We describe in depth the different challenges which beset such a method, including distributing/importing files, locally sorting the data on each node, uniformly binning the data, and exchanging data between the nodes. We also present the results of several different timing tests applied to the method. These tests demonstrate how the method scales when the number of files and/or the number of nodes is increased. Finally, we summarize the the greatest challenges in implementing the method, as well as the components of the method which were the most successful. We conclude with a discussion on ways in which the method could be improved.

Contents

1 The Method

1.1 Overview

As our first project for COMS 7900, the objective was to develop a system that would use MPI and the Babbage cluster to read in a significant amount of raw data and output that data in sorted sections. The data consisted of 500 files each with 20 million lines of data points. Each data point contained a integer number with three floating point coordinates. The project required the sorting of data by any of the coordinate columns of data points. Additionally, the project must output the sorted information using qsub and have the data stored in files listed from smallest to largest. Finally, the system must allow for testing the output for verification of the sorting and a method for process timing.

1.1.1 Workflow, project organization, and division of labor

After the assignment was distributed, the team members met to discuss and develop a brief outline of how the assignment would be handled. It was concluded that a prototype would be developed using MATLAB, and the final project would use C++ with C MPI calls. It was at that time Graham began working with the adaptive binning algorithm. As the project took shape, JJ set up GitHub and provided Graham and James with additional GitHub training. JJ also set up the structure for how the code would be presented, including general outline of the main program's structure and the outline for different files needed. Then, the team met and discussed the results of Graham's MATLAB prototyping. It was after the second meeting that we distributed work assignments and agreed upon variables and conventions. Numerous C++ functions were developed simultaneously which were then imported into a central `main` program via include statements. This minimized the number of merge conflicts and served to keep the code neatly organized. Regarding the workload distribution, JJ worked on file I/O and distribution, Graham developed binning and adaptation, and James worked on sorting and data exchanging. The code was assembled and debugged as functions were added.

1.1.2 Variables and conventions

Here we provide a helpful list of conventions, notations, and variable names used throughout this paper.

Counts:

- N: number of nodes
- W: number of workers $(N-1)$
- L: number of lines to read per file
- L_w: number of lines on the wth worker
- M: max number of allowed time steps

Indices:

- $m = 0, \cdots, M$ is the time step of the bin adaptation scheme (likely less than M)
- $n = 0, \cdots, W$ spans the nodes
- $w = 1, \cdots, W$ spans the workers
- $i = 0, \cdots, W$ spans the bin edges/indices
- $j = 1, \cdots, W$ spans the bin counts (this will occasionally subscript binI/E as well)
- $\ell_w = 0, \cdots, L_w - 1$ spans the lines on the wth worker
- $k = 0, \cdots, 3$ is the data column being sorted

Variables:

- $\text{data}^w_{4\ell+k}$: data point on ℓth line and kth column (0 indexed) on the wth worker (1 indexed)

- binE_j^m: bin edges (0 indexed) at time step m
- $\text{binI}_j^{w,m}$: bin indices on worker w (0 indexed)
- $\text{binC}_j^{w,m}$: bin counts on worker w (1 indexed w.r.t. w) at time step m
- $\text{binC}_j^{0,m}$: bin counts on head node (sum of worker binC's) at time step m

1.2 File I/O

There are two main steps in the initial file I/O phase. The first step is to identify the candidate data files and assign them to worker nodes. The second part is for the workers to import the data files into memory.

1.2.1 Distributing files

Rank 0 generates a list of data files to be processed by reading in all filenames from the specified directory. It looks for files matching the naming pattern `datafile00000.txt`, i.e., the string `datafile` followed by five digits, and ending with `.txt`. The list is stored in a C++ STL `vector` container.

Next the filenames are sent to the worker nodes in a round-robin manner as a means to evenly distribute the workload. This approach assumes that the files are equal in length. A more fair algorithm could accumulate the number of lines in each file and distribute chunks of data to each node to better balance the effort.

1.2.2 Importing files

Worker nodes import files using the `getline` function to read each line individually into a `string`. Fields are then parsed from the string based on their positions.

Field	Starting Pos	Ending Pos	Data Type	
Index	0	12	Ignored	
X	13	35	double	The `Index` field is calculated using the
Y	36	54	double	
Z	55	EOL	double	

value from the filename and a maximum number of rows in each file of 20,000,000. The starting index value for a file is:

$$I_0 = (Filename - 1) * 20000000$$

The index is then incremented for each row and used in lieu of the value from the file. By storing the index as a double, the need for a complex C `struct` is avoided, and the data can be stored in a one-dimensional array where contiguous blocks of 4 doubles is a single line from each file.

1.3 Sorting

In this section, we address the two options for sorting the data. Since the majority of the groups used qsort or some variation, we felt it would be interesting to have some alternative options for comparison. Therefore we offer two options while using our process: a linked list insertion sort and bubble sort. Both options require the user pass information in the following format.

```
void sort(double myArray[], int rows, int cols, int sortThisColumn)
```

Either version first uses the value in `sortThisColumn` to determine which column the user is requesting the sort on. Both options also make modifications to the existing `myArray[]`. The rows and column are used during the iteration process to determine the "shape" of the array during iteration. The limiting factor for both options is that the user is currently restricted to just the four columns. However, the function could be easily modified the accept more columns.

1.3.1 Linked list insertion sort

The faster of our two options by a significant margin is or linked list insertion sort. Inside this function is a one directional singly linked list made with the following node structure:

```
struct Node{
   double ll_location;
   double ll_x;
   double ll_y;
   double ll_z;
   Node *next;
};
```

The function initializes the linked with the first data point in the myArray[]. Then inserts each additional piece of data from `myArray[]` into the linked list using the traditional head, current, and previous pointer methods. After the method has iterated through all the data in `myArray[]` and has a sorted linked list, the data is then placed back into `myArray[]`.

1.3.2 Bubble sort

This function is the slower of the two methods and considered one of the least desirable of all sort functions. However, it is quite easy to implement, very robust, requires no additional large memory sections, and extremely reliable. This method simply sorts in place using the classic bubble sort process below.

```
for (int iii = 0 ; iii < (rows);iii++){
   for(int jjj=0; jjj<(rows-stop); jjj++){
         if(myArray[cols*jjj+sortByThisColumn]>
            myArray[cols*(jjj+1)+sortByThisColumn]){
               //perform swap
```

Although this will not be used as part of our regular process, we are interested in comparing its performance to other sorting methods. This method does does not attempt to resort the "last" space of the previous iteration. This is a slight reduction in the N^2 average that is usually expected from bubble sort methods.

1.4 Binning

We now discuss the optimization-inspired portion of the method. It has optimization properties since we need to find the bin edges such that the uniformity of the distribution is maximized. However, it is a constrained form of optimization since the bin edges must always maintain the positive volume constraint:

$$\mathrm{binE}_i^m < \mathrm{binE}_{i+1}^m \tag{1}$$

As we will see, the way we construct our adaptation formulae ensures that this constraint is always met.

1.4.1 Data binning w/ binary search

Since we use an iterative scheme to adjust the bin edges through time, we use equally-spaced bin edges between the global min and max of the data as an initial condition/approximation. Now, since each worker could theoretically have millions of data points, repeatedly looping through the data to find the bin in which each point lies would be inefficient. As such, we use the fact that the data is sorted to expedite the process significantly by using binary search. Each worker will have the same bin edges binE_i^m (where $i = 0, \cdots, W$), but the the location of the edges with respect to the data indices on each worker will likely be quite different if the data distribution across workers varies. With this in mind, for each worker, we search for the data indices at which each bin edge lies, giving us the new variables $\mathrm{binI}_i^{w,m}$ (note the superscript w, indicating that each worker has its own unique set of bin indices, as well as the superscript m, indicating the time step). To do this, we search for the index ℓ such that

$$\mathrm{data}_{4\ell+k}^w < \mathrm{binE}_i^m < \mathrm{data}_{4(\ell+1)+k}^w \tag{2}$$

giving $\mathrm{binI}_i^{w,m} = \ell + 1$. We also set $\mathrm{binI}_0^{w,m} = 0$ and $\mathrm{binI}_W^{w,m} = L_w$. We then calculate $\mathrm{binC}_j^{w,m}$:

$$\mathrm{binC}_j^{w,m} = \mathrm{binI}_j^{w,m} - \mathrm{binI}_{j-1}^{w,m}, \quad j = 1, \cdots, W \tag{3}$$

Lastly, all workers send their bin counts to the head node and we calculate the total bin counts:

$$\mathrm{binC}_j^{0,m} = \sum_{w=1}^{W} \mathrm{binC}_j^{w,m}, \quad j = 1, \cdots, W \tag{4}$$

1.4.2 Stopping criterion: the uniformity metric

Once the head node has calculated the total bin counts, it then determines how uniformly the is data distributed across the bins:

$$U^m = \max\left(\frac{\mathrm{binC}_{\max}^{0,m} - \mathrm{binC}_{\mathrm{avg}}^{0,m}}{\mathrm{binC}_{\mathrm{avg}}^{0,m}}, \frac{\mathrm{binC}_{\mathrm{avg}}^{0,m} - \mathrm{binC}_{\min}^{0,m}}{\mathrm{binC}_{\mathrm{avg}}^{0,m}} \right) \tag{5}$$

If U is below a set threshold (usually ≈ 0.1), then the the data distribution is deemed to be uniform in the sense that each worker will have within $\approx 10\%$ of the average data per worker; thus sorting on each worker will take roughly equal time.

1.4.3 Adapting bin edges

If the data is not uniform on the first step, we move on to adapt the interior bin edges ($i = 1, \cdots, W-1$):

$$\begin{aligned} \Delta C &= 2.0(\mathrm{binC}_{i+1}^{0,m} - \mathrm{binC}_i^{0,m})/(\mathrm{binC}_{i+1}^{0,m} + \mathrm{binC}_i^{0,m}) \\ \Delta B &= \mathrm{binE}_{i+1}^m - \mathrm{binE}_i^m \\ \mathrm{binE}_i^{m+1} &= \mathrm{binE}_i^m + \alpha \Delta C \Delta B \end{aligned} \tag{6}$$

where $0 < \alpha < 0.5$. Each of these terms is designed the allow the bins to adapt the maximum amount possible without the bin edges becoming out of order. The quantity ΔB scales the maximum change to be within the current bin width. The quantity $0 \leq \Delta C \leq 1$ is a type of normalized gradient which will direct the bin edges toward regions with higher density. Lastly, α is a form of rate constant. It must remain less than 0.5 to maintain the constraint. It is usually set between 0.25-0.475.

1.4.4 Prototyping in MATLAB

Prior to our implementation in C++, we prototype the adaptation scheme in MATLAB as a proof of concept. For these tests, we read 1000 lines from one of the provided data files and applied the adaptive binning scheme to it, varying the number of nodes, iterations, and the value of α.

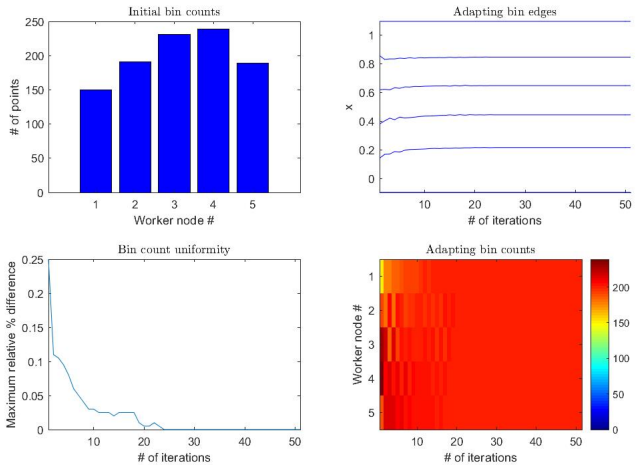


Figure 1: 5 nodes, 1000 data points, $\alpha = 0.475$

As can be seen from the Figure 1, with $W = 5$ and $\alpha = 0.475$, the method converges to a highly uniform distribution in roughly 20 steps. However, when increasing the number of workers to 10

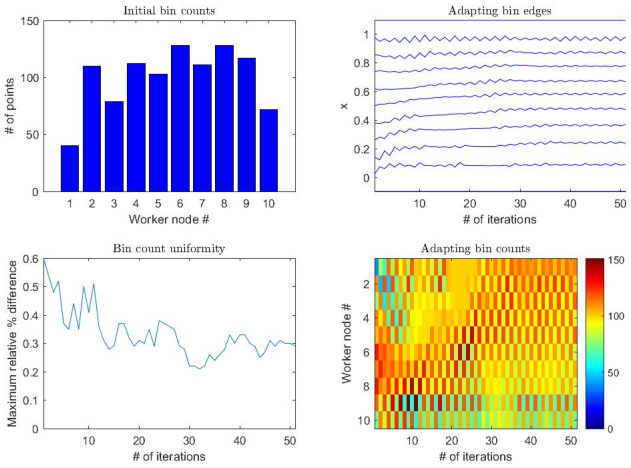


Figure 2: 10 nodes, 1000 data points, $\alpha = 0.475$

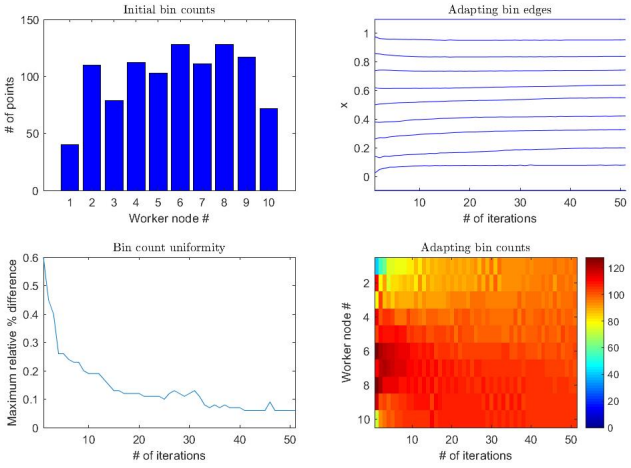


Figure 3: 10 nodes, 1000 data points, $\alpha = 0.25$

(Figure 2), the method struggles to converge in the same amount of time. This problem arises because adding more workers adds more bin edges. Since the test only uses 1000 data points, then there is only an average of 100 points per bin, which is not sufficiently smooth for the method to perform at its optimal level. As such, we must decrease α so that we can remove the wild oscillations present in the plots. This is what Figure 3 demonstrates. Though it still does not converge quite as fast as the 5 worker test, it is significantly better than the second test.

These tests demonstrate two general rules which describe the performance of the method.

First, the method performs better (faster convergence, less oscillations) with more data per bin (less variance in the sense of the central limit theorem), since altering the bin edges on a smooth distribution won't have as drastic an effect on the bin counts. This being said, in practice, loss of performance due to the distribution's granularity is not significant when dealing with billions of data points because no practical number of workers can make the data significantly granular. Second, since we use only a local bin updating scheme (a form of normalized gradient of the bin counts), bin edges in low-density regions of the data must wait for bin edges in high-density regions to move (i.e., propagate data to their bins) before they themselves can move. Consequently, this makes our method susceptible to very poor performance if there is an outlier in the data, because there will be a large number of bin edges between the outlier and the rest of the data which cannot move until some data is propagated toward them. A clear improvement which could be made to the method would be to replace the local bin count gradient with a global analogue (which still maintains the constraint, of course) or to use an integration-based scheme rather than a differentiation-based scheme.

1.5 Exchanging data

The exchanging data process for our design includes two steps: Data swap and Clean up. This is to allow flexibility in the main program during its implementation of the functions.

1.5.1 Data swap

When this function is called, it uses a from/to process, meaning that—given the following set of options—it will transfer a specific set of data from one node to another.

```
void swapArrayParts(double *pmyArray[], int *rowPTR , int *colPTR, int myrank, int
    numranks, int *binIPTR, int fromWho, int toWho)
```

The first task of the process to is pass information regarding the length of the data to be received from the array. If the worker that called the function is `fromWho`, it will pass its binI information to the worker who called the function as `toWho`. At this point, the `toWho` worker creates a temporary array that is large enough to fit both the current set of data plus the additional set of data it is about to receive. After that is completed, `fromWho` then passes the pointer to the beginning of the data point in `pmyArray` where `toWho` begins its length of data to transfer. Here, `fromWho` will exit the function and `toWho` will simply insert the data from its current `pmyArray` into the temporary array. Then in the remaining spaces inserts the information it just received from `fromWho`. Finally, `toWho` will increase the number of rows to include the additional data points. Then, `toWho` exits the function. At this point, `fromWho` still has all of its original data and `toWho` now has its original data plus the additional data sent over from `fromWho`. Since this is a one to one transfer, it can be implemented in any way taht the main program chooses. For the purposes of our group's needs, we chose the following method:

```
for( fromWho = 1; fromWho < numNodes; fromWho++ )
   for( int toWho = 1; toWho < numNodes; toWho++)
      if(toWho!=fromWho)
         if(myRank == toWho || myRank == fromWho)
            perform swapArrayParts
```

1.5.2 Cleanup

Once the swap function is complete, each worker will have all the information from each other worker appended to the end of their existing array. However, each worker will still have the information that it has passed to others and will now need to remove these data from their own array. This is where the cleanUp function is utilized. Whenever the cleanUp function is called, it receives the following parameters:

```
void cleanUp(double *pmyArray[], int *rowPTR , int *colPTR, int myrank, int
    numranks, int *binIPTR)
```

CleanUp first uses the information in `*binIPTR` to determine the amount and locations of the data that was given to other worker and the amount of information it should retain inside its array. Then we `malloc` a temporary array with the correct size. The information that should be kept is inserted into the temporary array while excluding the data that should not be included. Then, `*pmyArray[]` is pointed to the beginning of the temporary array and the row size is adjusted to reflect the new and correct size of the worker's array. The worker now exits the function with the correct size and information however the data is no longer sorted and a new sort must be completed. This function must be called individually for all workers used in the `swapArrayParts` method.

2 Performance

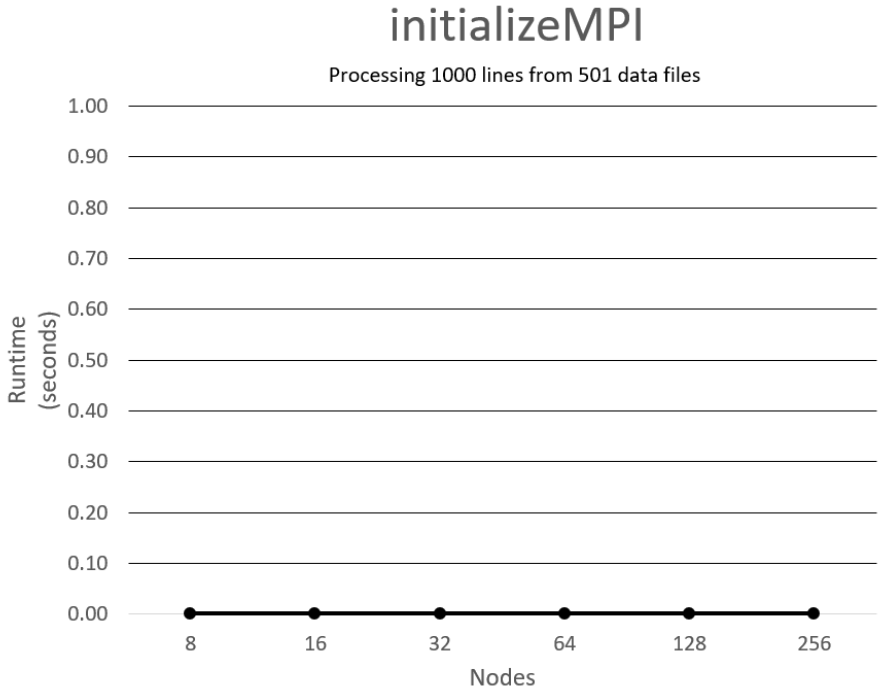
2.1 Methodology

The following sections report the performance of the code using differing numbers of MPI nodes. The *nodes* in each chart refers to the command line parameter:

`mpirun -np` *nodes* `./parallelSort`

2.2 initializeMPI

The `initializeMPI` function refers to performing the `MPI_Init` call, `MPI_Comm_rank`, `MPI_Comm_size`, and `MPI_Get_processor_name`. There is no reportable difference in performance based on the number of nodes requested.



2.3 distributeFileNames

This code reads the list of files in a directory matching the required pattern and distributes them to the workers using `MPI_Isend`. Timing ends when all nodes have completed their receives. As the number of nodes increases, the amount of time required grows logarithmically. (**Note:** the x-axis is logarithmic in powers of two.)

2.4 importData

This code performs a `while` loop which reads each line from each file. The line is stored as a `string` and parsed using `substr` and `stod`.

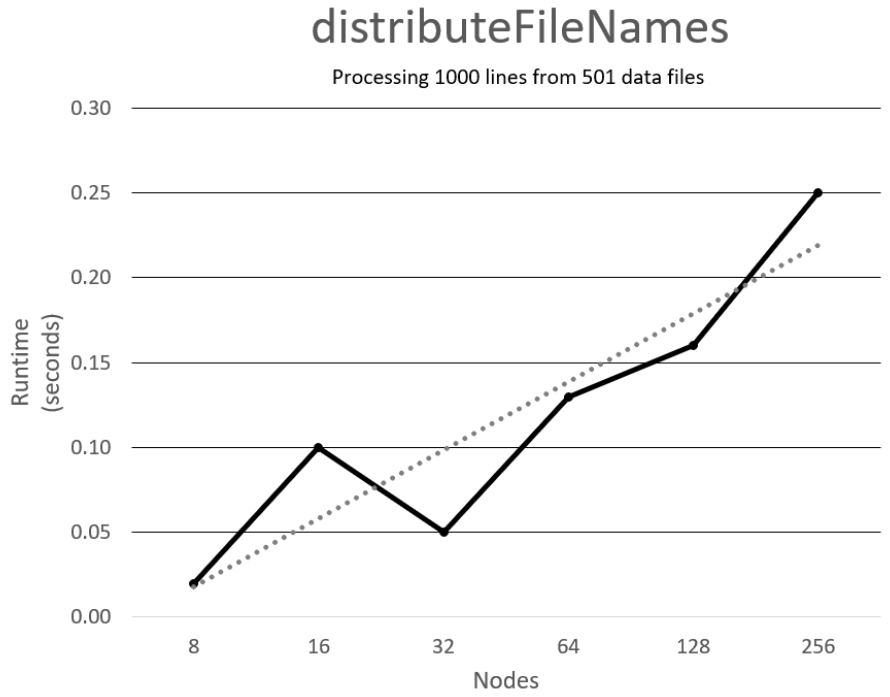
The amount of time required to import files drops exponentially as the number of files any single node must process is reduced.

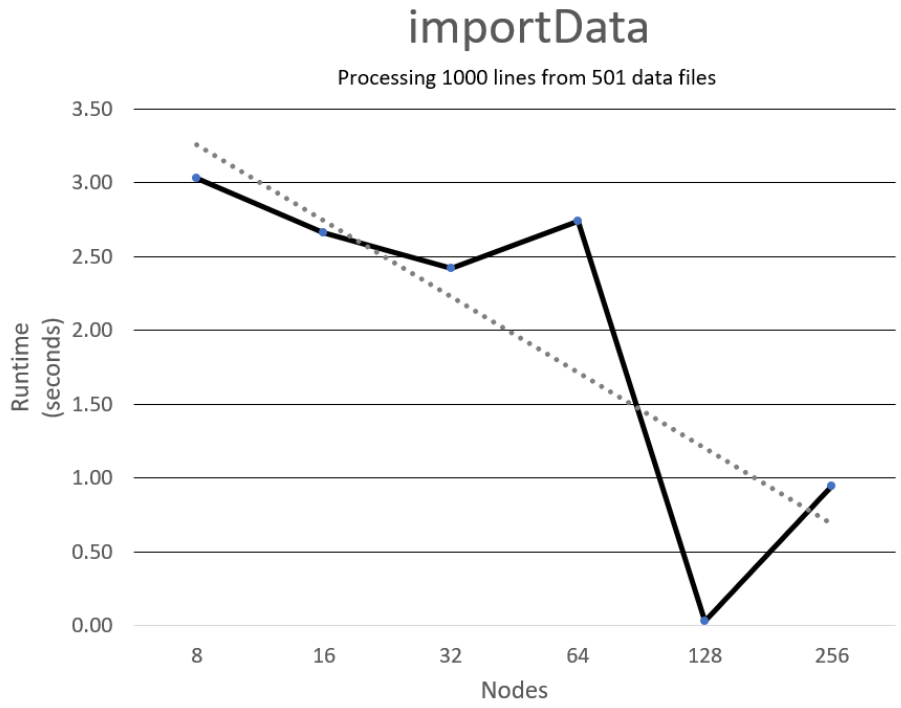
2.5 sortData

The `sortData` function reorganizes the data in ascending order by the selected field. Like `importFiles`, the time to perform the sort falls exponentially since the amount of data is reduced.

2.6 adaptBins

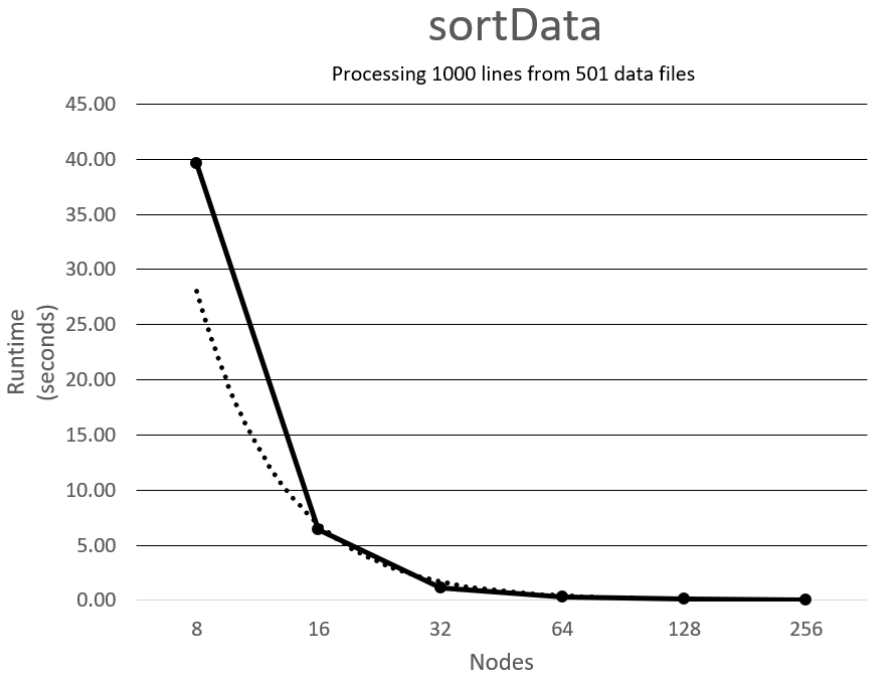
The `adaptBins` portion of the code appears to perform well until 256 nodes. Other sections of the code were experiencing significant performance issues at this scale, and it is possible that the problem may be related to the cluster's workload rather than the algorithm. Unfortunately, the method to validate this is to run the code without other users on the system.

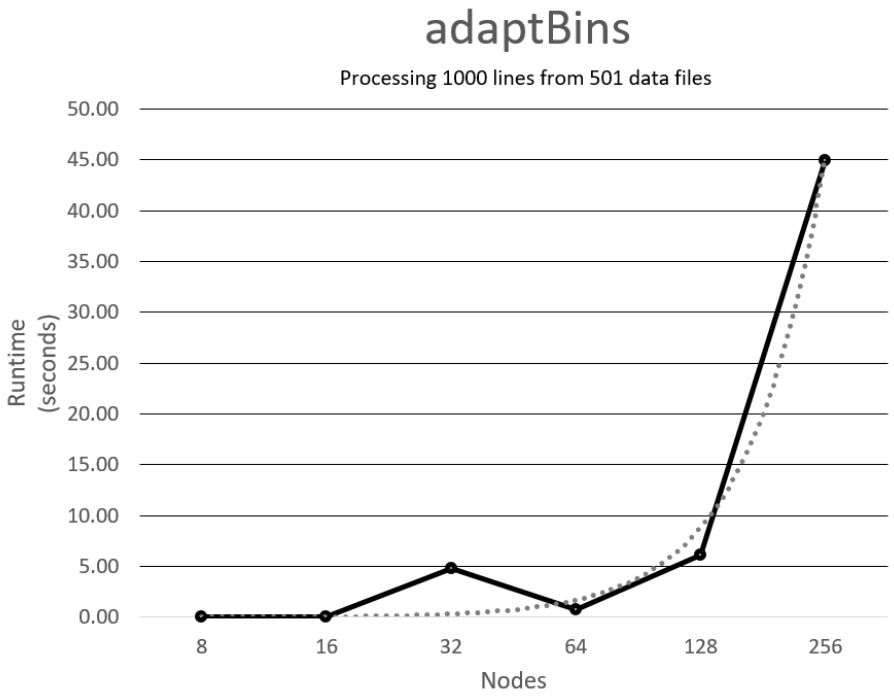




2.7 swapData

The `swapData` suffers from the same performance issues that `adaptBins` does. Both routines are heavily network dependent. Again, this routine needs to be tested on the cluster without any other workload to determine if the times reported are due to the algorithm's design or the network.



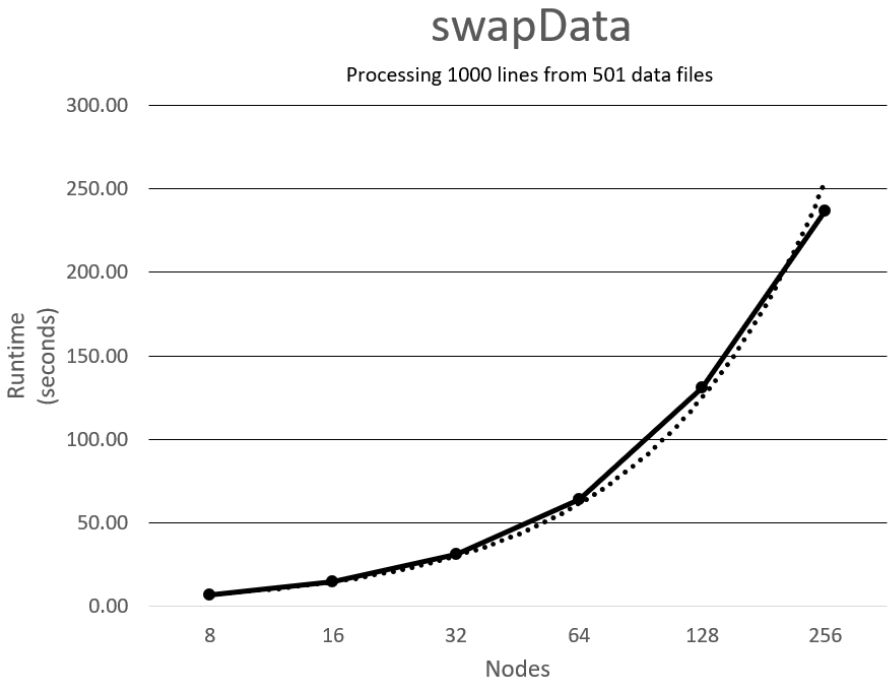


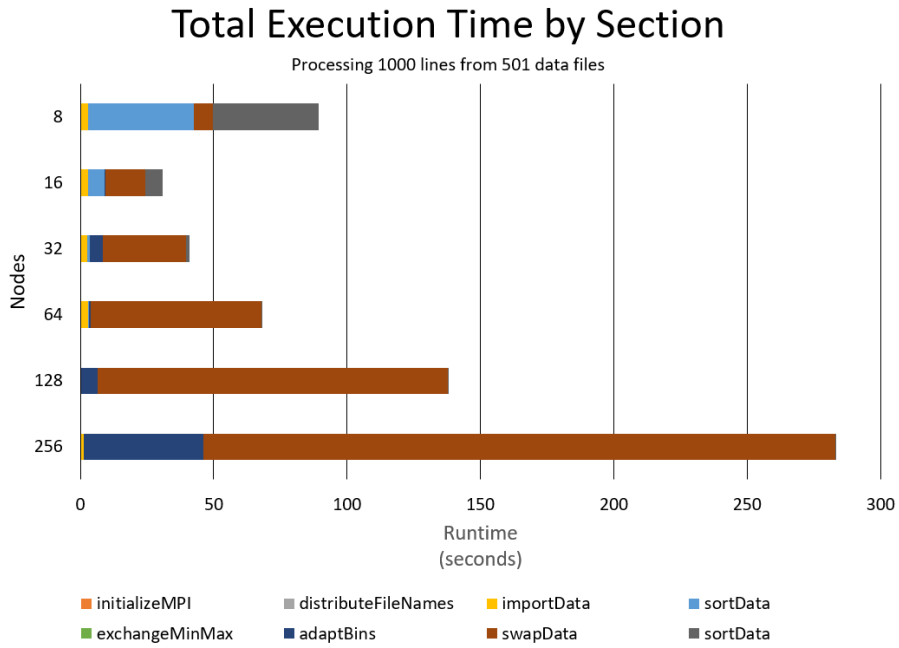
2.8 Overall

Here we present plots and tables which display the timing data of all methods simultaneous for easy comparison.

3 Conclusions

We will now conclude with two discussions on 1) the most challenging and most successful aspects of our method and 2) ways of improving the both the method's performance/efficiency and our



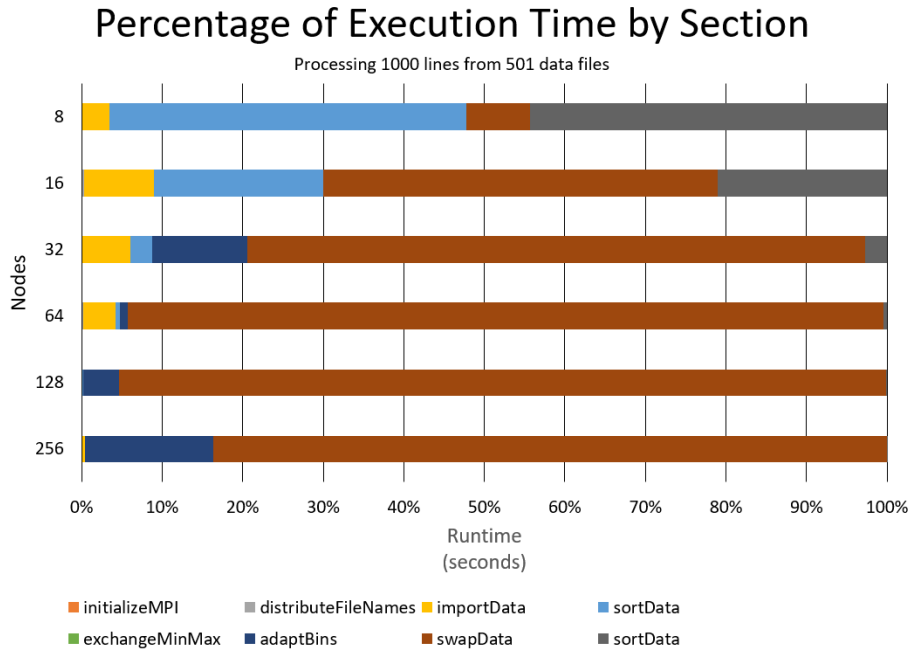


workflow as a group.

3.1 Challenges and successes

Integration The integration proved challenging due to a lack of specifications for the individual functions. While we agreed on *what* the function would perform, we did not discuss parameters and return values.

- Identifying structure of parameters and return values
- Needed better planning of data types up front



Nodes	initializeMPI	distributeFileNames	importData	sortData
256	0.00	0.25	0.95	0.03
128	0.00	0.16	0.03	0.12
64	0.00	0.13	2.74	0.35
32	0.00	0.05	2.42	1.12
16	0.00	0.10	2.66	6.46
8	0.00	0.02	3.03	39.61

Nodes	exchangeMinMax	adaptBins	swapData	sortData
256	0.00	44.96	236.75	0.03
128	0.00	6.08	131.23	0.12
64	0.00	0.72	63.98	0.35
32	0.00	4.81	31.26	1.12
16	0.00	0.02	15.05	6.46
8	0.00	0.01	7.00	39.61

Nodes	Total
256	282.97
128	137.74
64	68.27
32	40.78
16	30.75
8	89.28

Git We had a basic knowledge of Git but not in the context of working with a group. Rather than creating individual branches for each team member, we attempted to work from the *master* branch. This resulted in the loss of work on at least two occasions.

- Resolving merge conflicts
- Basic Git workflow

Coding During debugging, we added print lines and other code to help identify problems. None of this code was removed.

- Forgot to remove debug controls while collecting performance data

Testing Testing on large sets of data was challenging due to the long runtimes. If a crash occured, the program had to be restarted, resulting in repeating the data import section.

- Long runtimes when testing with full dataset
- Failures during long runs

3.2 Future work

Shared memory Rather than rely solely on MPI, the code could be modified such that a single MPI process is launched on each physical host and pthreads is used for interprocess communication on the host. This would allow a single copy of the data to be stored on the node and shared among the local threads.

- Combine with `pthreads` to leverage shared memory

Utilize All-to-all and Reduce MPI operations We chose to leverage `MPI_Isend` and `MPI_Recv` for data transfers rather than the more efficient all-to-all and reduce operations. This could improve our data swapping routine.

File I/O The import routine is very inefficient due to its use of C++ string operations to parse the data. A major improvement would be to use `fscanf` to read the file and immediately export the results as binary. If the program has to be executed again, it can check for the binary version. Also, `pthreads` would allow multiple files to be read simultaneously.

- Use `fscanf` to read files
- Read multiple files in parallel on single host
- Balance data distribution at a row level rather than a file level to deal with differing file sizes
- Intelligent distribution of rows to nodes as a "presort"

Adaptive Binning Our local time stepping scheme—though efficient for smooth, reasonable datasets—can perform poorly when given certain pathological cases. Also, its very structure limits its rate of convergence. A method which uses linear or spline interpolation and integration instead of normalized gradients would likely have superior convergence properties.

- Method susceptible to pathological cases
- Structure of method is self-limiting
- Using global, integral/interpolation-based method instead

Recoverability Data could be exported during major milestones (e.g., convert text to binary, after sorting, and after exchanging. If the program fails, it can check for the checkpoint files and restart at the milestone.

- Output data at checkpoints to avoid repeating successful operations

Testing A framework such as `CMAKE` would allow automated testing and simplify including unit tests to verify code works. The performance data was collected manually, and the process could be easily automated using Python or Bash.

- Utilize `CMAKE` to automate build and testing
- Utilize scripts to run various configurations and automate performance data collection