An optimization-inspired approach to parallel sorting

Team Metropolis:
James Farzi, 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

Introduction:

1.1 Overview

1.1.1 Workflow

We used C++ with C MPI calls
Used GitHub
Workflow description

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

1.2 File I/O

```
walk = RandomWalk(nStep, initPos, stdDev)
```

1.2.1 Distributing files

1.2.2 Importing files

1.3 Sorting

```
walk = RandomWalk(nStep, initPos, stdDev)
```

1.3.1 Linked list merge sort

1.3.2 Bubble sort

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 we maximum the uniformity of the distribution. However, it is a constrained form of optimization since the bin edges must always maintain the relation

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

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 as an initial condition/approximation equally-spaced bin edges between the global min and max of the data. 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. Now, 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 $\text{binI}_i^{w,m}$ (note the superscript w, indicating that each worker has its own unique set of bin indices). To do this, we search for the index ℓ such that

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

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

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

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

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

1.4.2 Stopping criterion: the uniformity metric

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

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

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 in the first step, we move on to adapt the interior bin edges ($i = 1, \cdots, W-1$):

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

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.

1.5 Exchanging data

1.5.1 Data swap

1.5.2 Cleanup

2 Testing

2.1 File I/O

2.2 Sorting

2.2.1 Linked list merge sort

2.2.2 Bubble sort

2.3 Binning

2.3.1 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.

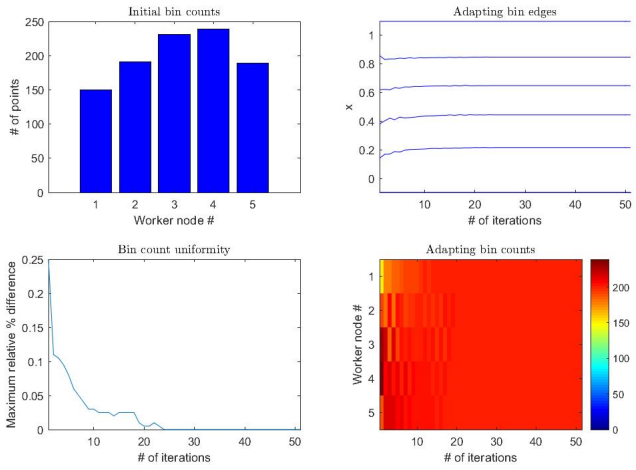

Figure 1: 5 nodes, 1000 data points, $\alpha = 0.475$

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), 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 sufficiently 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 since altering the bin edges on average won't have as drastic an effect on the bin counts. Also, the method performs worse (slower convergence, possible oscillatory behavior) with more workers (i.e., bin edges). There

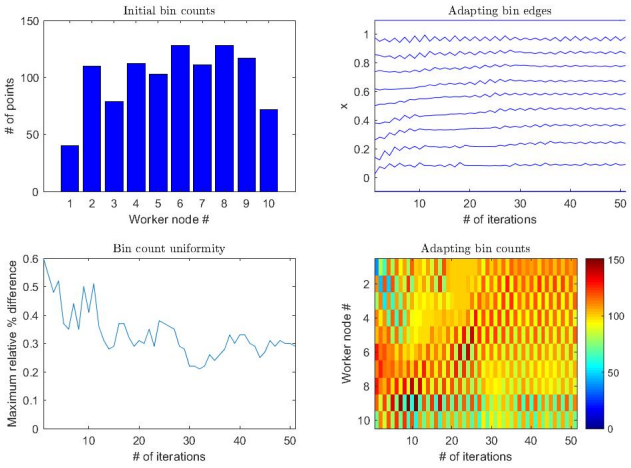

Figure 2: 10 nodes, 1000 data points, $\alpha = 0.475$

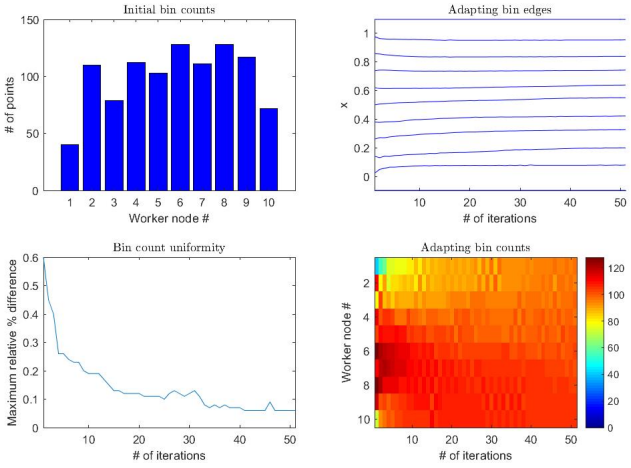

Figure 3: 10 nodes, 1000 data points, $\alpha = 0.25$

are two reasons for this. First, relating to the previous rule, since more granular divisions of the data counter the positive effects that large amounts of data have on performance. This being said, in practice, this effect is not significant when dealing with millions or 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), the bin edges near the boundary of the data range do not have much room to move since the interior bin edges must move first (since we must maintain the constraint). 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 (whicj still maintains the constraint, of course).

2.3.2 C++ runs

2.4 Exchanging data

3 Conclusions

We will now conclude with two discussions on 1) the most difficult 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

3.2 Future work