Parallel Quicksort With Load Balancing

CS 566 Parallel Processing (2023 Spring)

-By Prof. Shantanu Dutt

Vikram Abhishek Sah
Meer Shah
Utsav Sharma
Jason Pereira

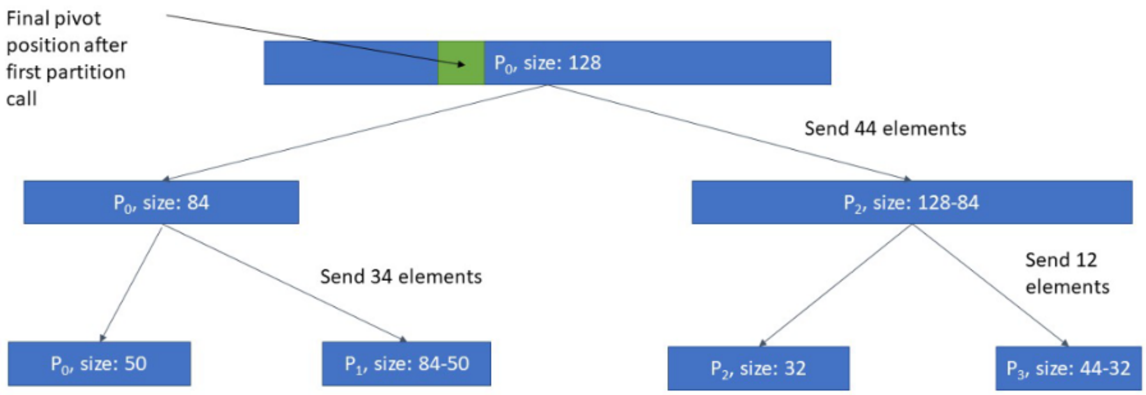
TABLE OF CONTENTS

Phases of the Algorithm

Our algorithm reads the input data from a text file in P0 and implements the below phases in sequence:

- Recursive pivoting and distributing the elements among the processors.
- Left to Right loading balancing among the leaves of the recursion tree.
- Right to Left loading balancing among the leaves of the recursion tree.
- Local sorting.
- Writing the output to file using sequential tokens.

Initial Data Distribution



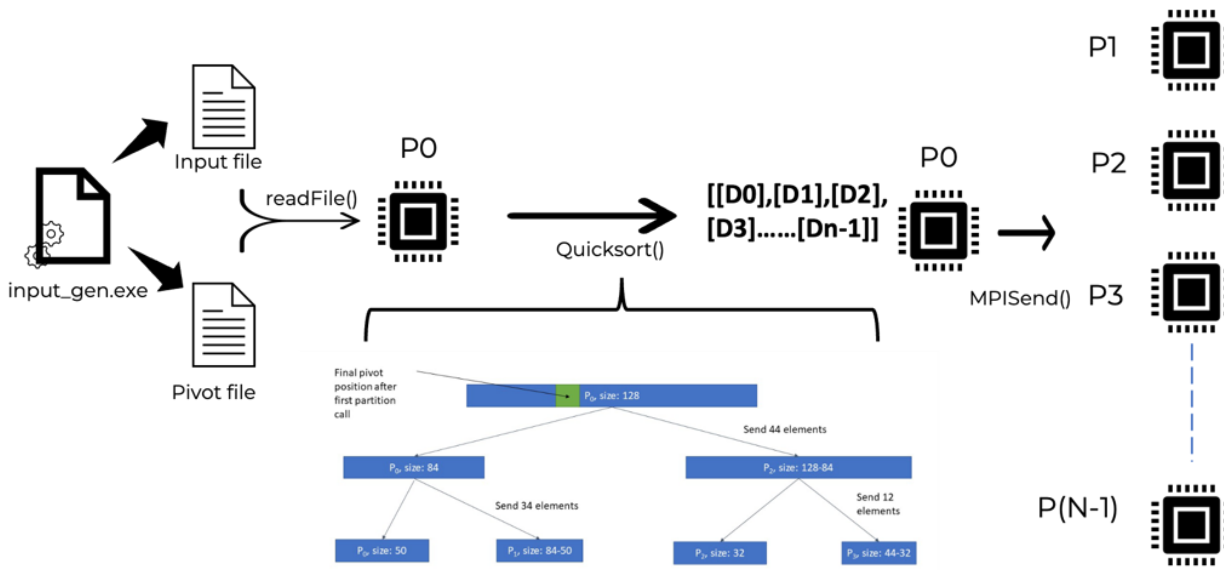
We are using a recursive pivoting approach to distribute the data. The examples here assume that P=8
Step 1: P0 loads the unsorted array into its local memory,
Step 2: P0 partitions the data on the basis of the pivot provided.
Step 3: The (low -> pivot) elements are kept at P0.
Step 4: The (pivot +1 -> high) elements are sent to its partner processor, which is P4 (or P/2i+1 processor).

Assuming a hypercube topology, steps 2 to 4 are repeated by the active processors in each iteration of the algorithm, till all P processors have a portion of the array.
Since we are assuming a hypercube topology, this distribution would complete in log P iterations.
Performing the pivoting and distribution in this manner instead of sequentially ensures that we are not bottlenecked by the sequential distribution on P0, which would then take another P-1 communication step to send the data to the other P-1 processors.

Distributing the algorithm in this manner guarantees that the subarrays being distributed to each processor are ordered from smallest to largest going from P0 to P-1.
This is because once the pivot is chosen, the elements in the array at each processor in each iteration are arranged as smaller or larger than the array, and then the higher portion of the array is sent to my_id+1.
This ensures that when the local arrays are sorted and written to file, the ordering of the numbers is maintained.

After the introduction of pivot generation file introduced in this project, we tweaked our algorithm To be able to distribute the data as per the defined pivots in the pivot file. This distribution algorithm entailed the following steps.

- Generating the Input and the pivot files from the input_gen.exe file provided.
- Reading these files into P0 and applying the pivoting logic against the pivots in the pivot file and running the left and right partitioning algorithm of quicksort to split the data recursively for P processors.
- Once the partitioning is done we send the data sequentially to the P-1 processors from P0.




```cpp
void quicksort(vector<int>& arr, int low, int high, vector<int>& pivots) {
    if (low < high) {
        int p = partition(arr, low, high, pivots);
        vector<int> v;
        for (int i = low; i <= p; i++) {
            v.push_back(arr[i]);
        }
        alldata.push_back(v);
        if (p == -1) return;
        // quicksort(arr, low, p - 1);
        quicksort(arr, p + 1, high, pivots);
    }
}
```

```cpp
int partition(vector<int>& arr, int low, int high, vector<int>& pivots) {
    // Use middle element as pivot
    // int pivot = arr[(low + high) / 2];

    if (p > pivots.size() - 1) {
        return -1;
    }
    int pivot = pivots[p++];
    int pivotIndex = find(arr.begin(), arr.end(), pivot) - arr.begin();

    // Initialize pointers
    int i = low;

    // Move pivot element to the end of the range
    std::swap(arr[pivotIndex], arr[high]);

    // Partition the range
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            std::swap(arr[i], arr[j]);
            i++;
        }
    }

    // Move pivot element to its final position
    std::swap(arr[i], arr[high]);
```

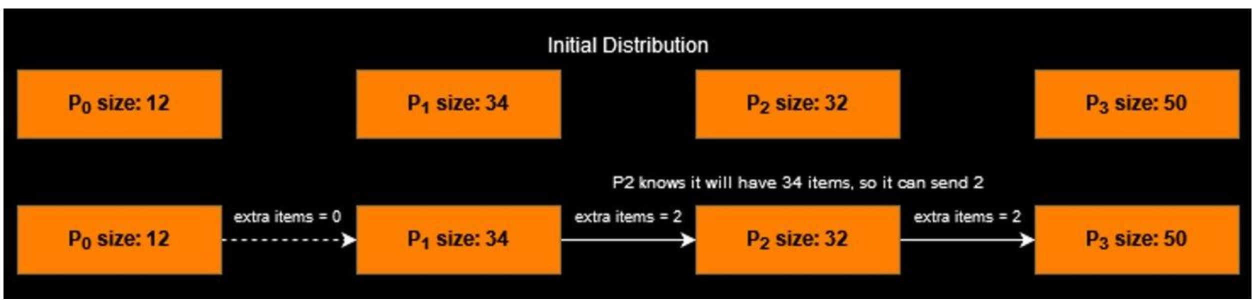
Load Balancing

Once the individual arrays have been distributed to the processors, the function loadbalancing() is called. This function is responsible for all elements being moved around in the processors to achieve optimal load balance.

The load balancing phase is divided into two steps which happens for each direction:
Step 1: Message passing to inform your neighbor of the number of extra elements that they are going to receive.
Step 2: Finding and sending those extra elements to your neighbor.
Looking at each step in detail:

Left to Right:

Step 1 :



In a while loop, where docontinue > 0,
P0 calculates its extra elements as: extraElementsSize = localArray.size() - optimalSize;

If negative, the value is set to 0 and extraElementsSize is sent to the next processor, i.e. P1.

Each processor, on receiving the numberOfElements, calculates its extraElementsSize = localArray.size() - optimalSize + numberOfElements;

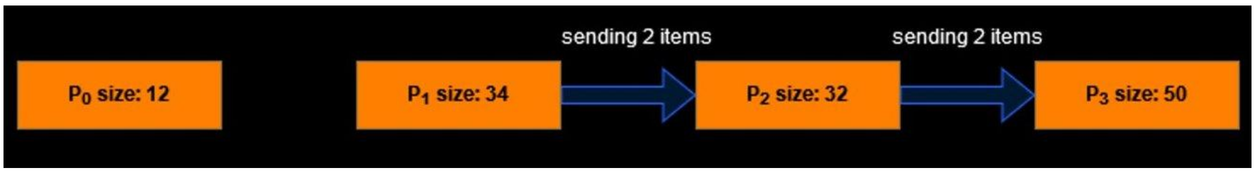
If negative, the value is set to 0 and extraElementsSize is sent to the next processor (myid +1).
If we don't have lessthan or greater elements in our localarray as extraElementsSize, we send 0, but do not change the value held in extraElementsSize, since we use that later.
Once this sequential communication reaches the P-1'th processor, we stop.

Then, each processor check if its extraElementsSize > acceptableImbalance * optimalSize.
We have set acceptableImbalance to 0.2, signifying 20%, but this is a tunable parameter.

If this condition is met, the variable mycontinue is set to 1. Else, it is set to 0.

We then do a reduction operation on P0, where we collect the sum of all the mycontinue into the variable docontinue. Then, do continue is broadcast to all the other processors from P0.
Next we check if we have reached the maximum number of rounds set in maxrounds, and if yes we break out of the outer while loop.

Step 2:



If docontinue is greater than 0, this means that those many processors want to continue to the next step of sending the messages.
To send the messages, at each processor parallelly, we check if we have extraElementsSize > 0 and < localArray.size(). If yes, we proceed to send the elements. To calculate the extra elements, we first find the KLargest elements in the array using a quickselect based function. Then we remove those elements from the array using a heap based remove function, and send the extraElements to the processor that is myid+1.
Then, we receive the elements and add them to our localArray.

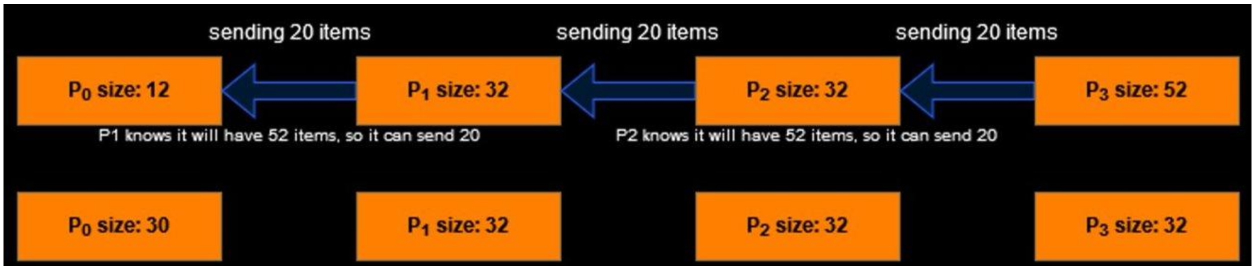
Right to Left:

Step 1:

Once we finish enough rounds on the left to right load balancing, we start load balancing from right to left, to ensure that we do not miss any load imbalance.
The steps are the same as the Left to Right load balancing, except that we send the data to myid-1, since we are going right to left.

Step 2:

Similar to Step 2 of left to right load balancing, we perform data sending in parallel on all processors. However, since we are sending from right to left, we are going to select the KSmallest elements. The remove and sending happens in much the same way as before.



Each of the ‘Left to Right’ and ‘Right to Left’ phases are repeated either for a set number of maximum iterations, or till no processor has sent a value in extraElementsSize > 0.

There is some sequentialization in the first message passing phase, but there is a lot more efficiency gained by parallelizing the actual sending of elements, since those take the bulk of the time to move around between processors.
After we either reach maxrounds on both Left to Right and Right to Left, or we reach a state of load distribution where the data on each processor is > acceptableImbalance*optimalSize, we terminate the load balancing and proceed to sort the data.

Insertion sort at each processor

At the end of all rounds of distribution and load balancing the data within the individual processors are sorted using insertion sort. Insertion sort would also be affected by the quality of the data. That is, the array where a lot of elements are in place, will take less time to sort. This matters because during the load balancing phase, while sending the elements to another processor, the order is changed towards sorted.

To determine the average efficiency of insertion sort consider the number of times that the inner loop iterates. As with other loops featuring nested loops, the number of iterations follows a familiar pattern: 1 + 2 + ... + (n - 2) + (n - 1) = n(n - 1) = O(n2).

Writing to file using sequential token passing

- To write the output to the file, we use a token based system where P0 opens the file, writes its output and closes the file. It then passes the token to my_id+1.
- This process is then repeated at all the other processors in order, till we reach P-1.

Optimization

Quickselect optimization

- At a particular point in the load balancing process, we need to find the k smallest elements from a vector of N elements. The standard ways to do this are using a simple brute force algorithm that is O(N**2) complexity. A better approach is to use a heap and reduce the complexity to O(Nlogk)
- However, since this function is called frequently in our algorithm, we needed to reduce it time complexity even further.
- For this we have used a variant of quickselect. Quickselect is an algorithm that positions the k smallest element in its correct position in an array by putting all the elements smaller than it to its left and all the elements larger than it to its right.
- It does this by choosing a random pivot and positioning it in its correct position. Then, it recurses on either the left or right depending on the value of k.
- It can thus find the k smallest elements in O(N) average time complexity, which is a huge improvement from N log k

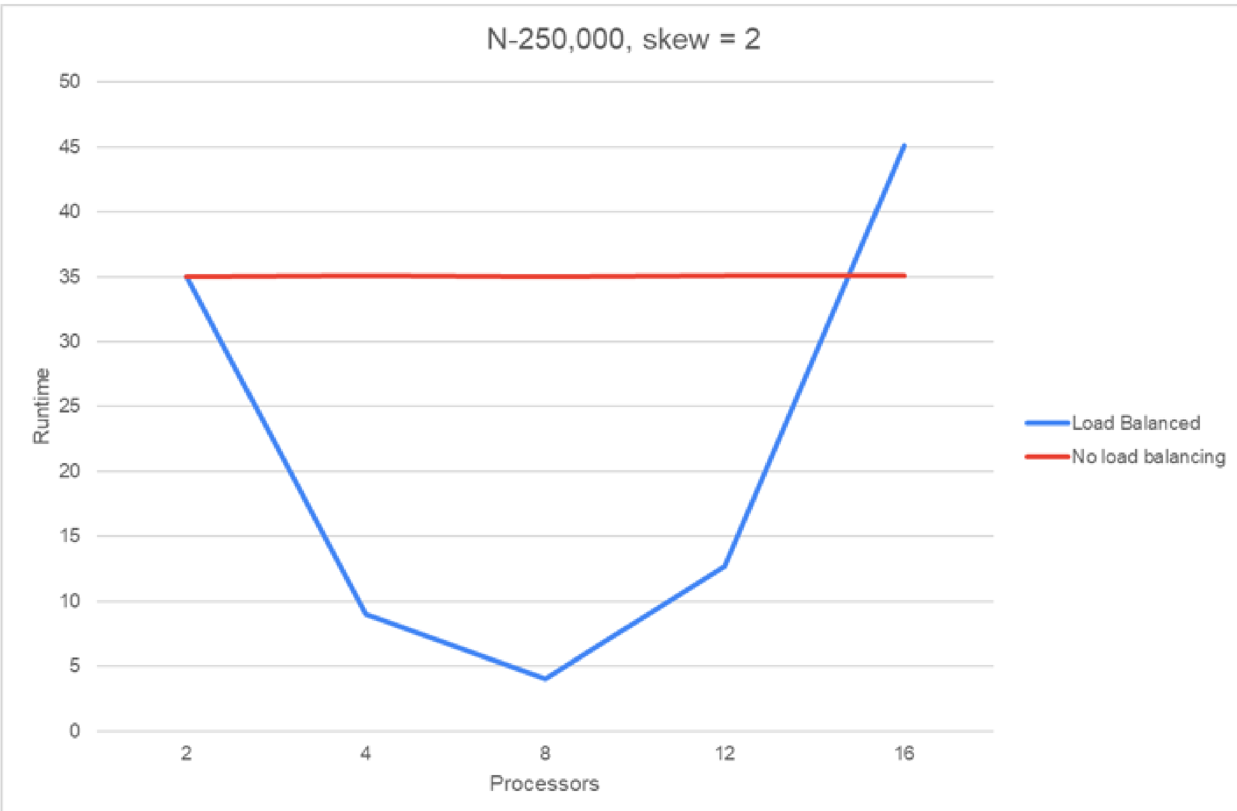
Using a hashmap to remove elements from a vector

- This function was written to remove the elements from the localArray that we were sending to the partner processor.
- We have ensured that we handle duplicates properly, and remove only those many occurrences of a number as are in the extraElements vector
- We use a hashmap to keep count of the elements that need to be removed from the original vector. We create this hashmap from the extra elements vector which is the vector containing elements to be removed.
- Then, we loop over the original vector and check if the current elements is present in the hashmap and its count hasn't gone down to 0. If it is 0, then we don't touch it because we've already removed the correct number of occurrences of that element.
- If it is not 0, then we skip adding that element to our result vector and decrement its count in the hashmap.
- Using a hashmap reduces the complexity of this function from O(N**2) to O(N).

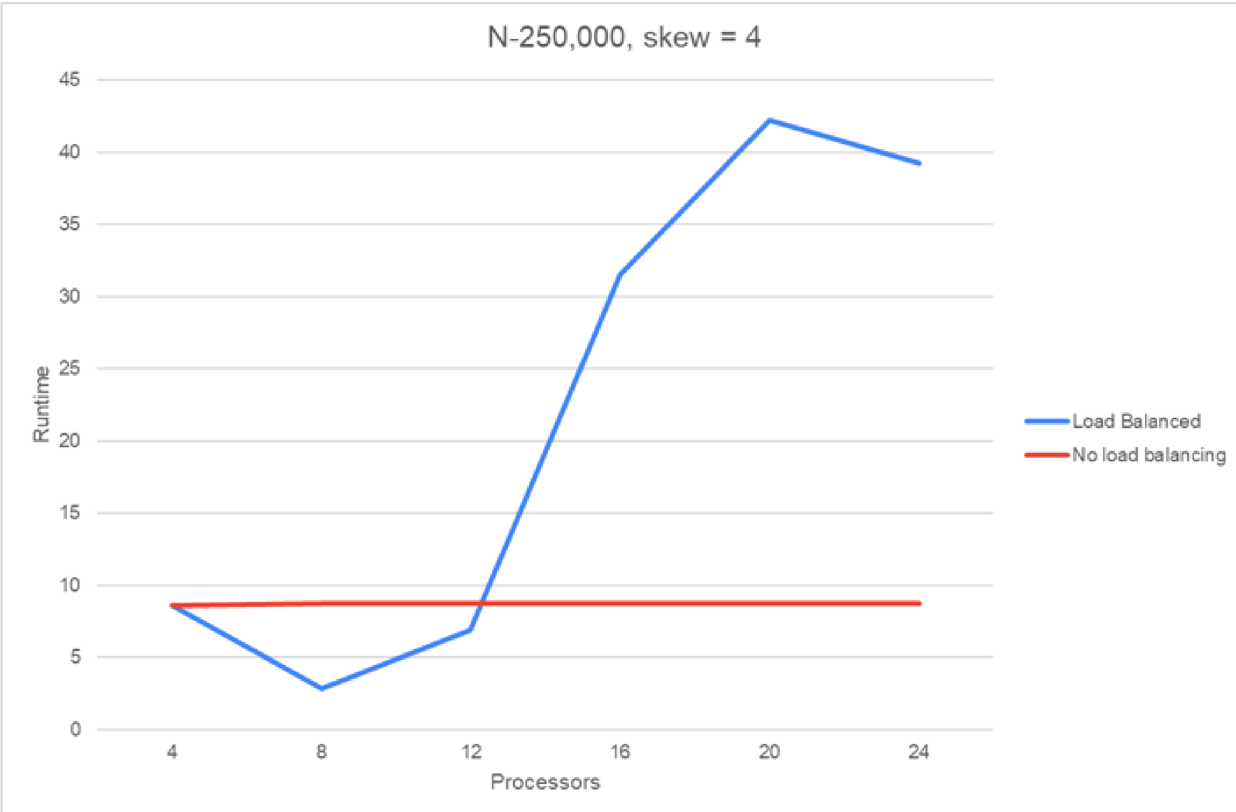
Readings

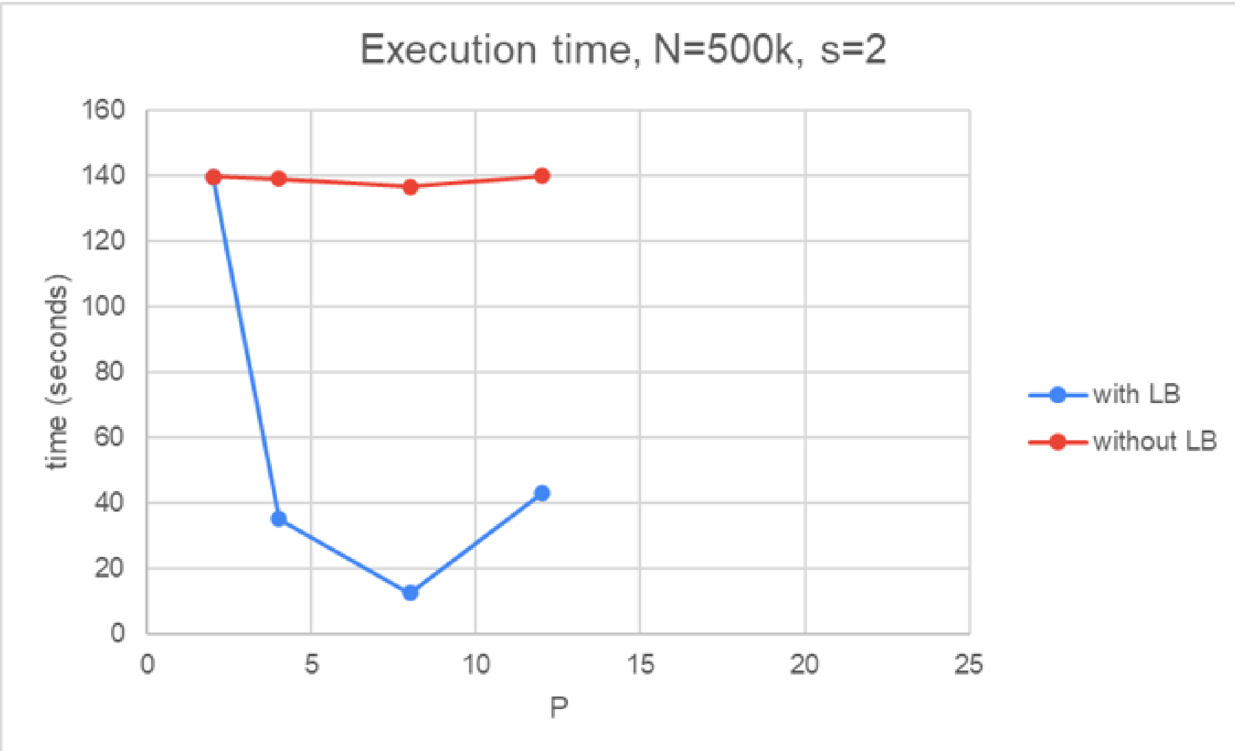
Results

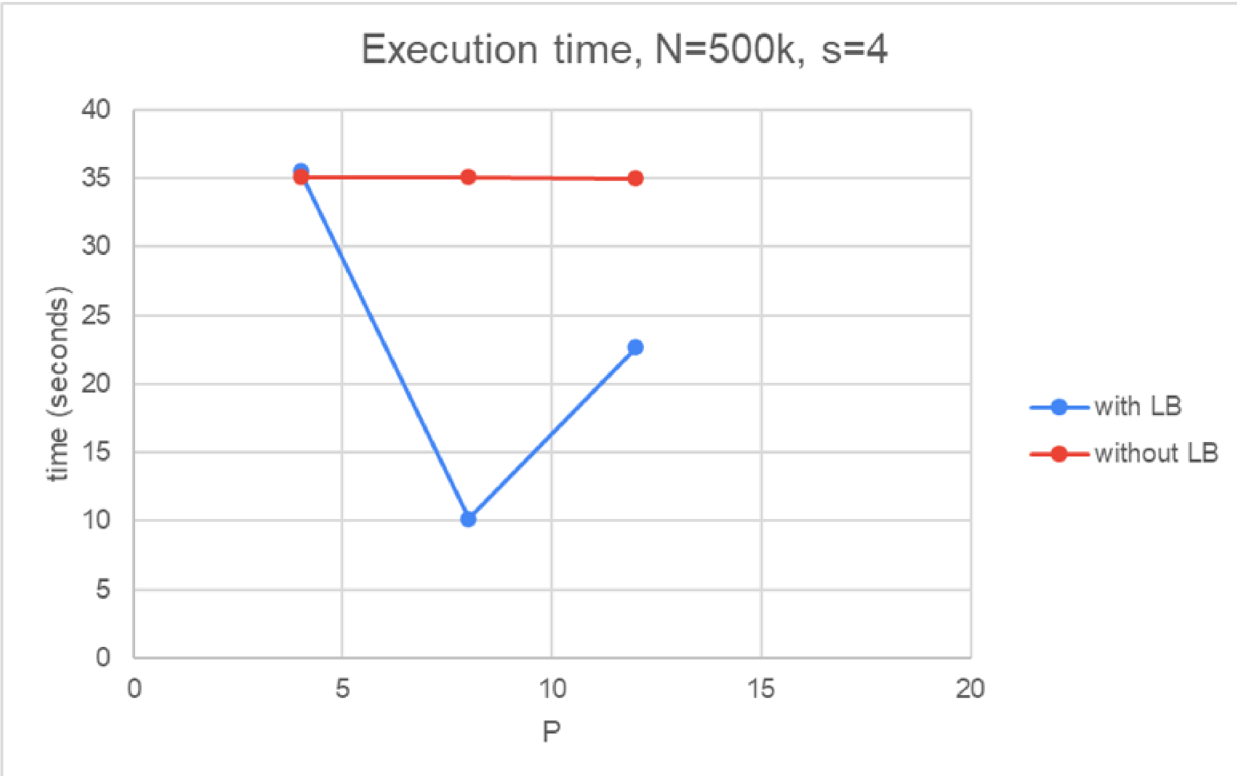
1.

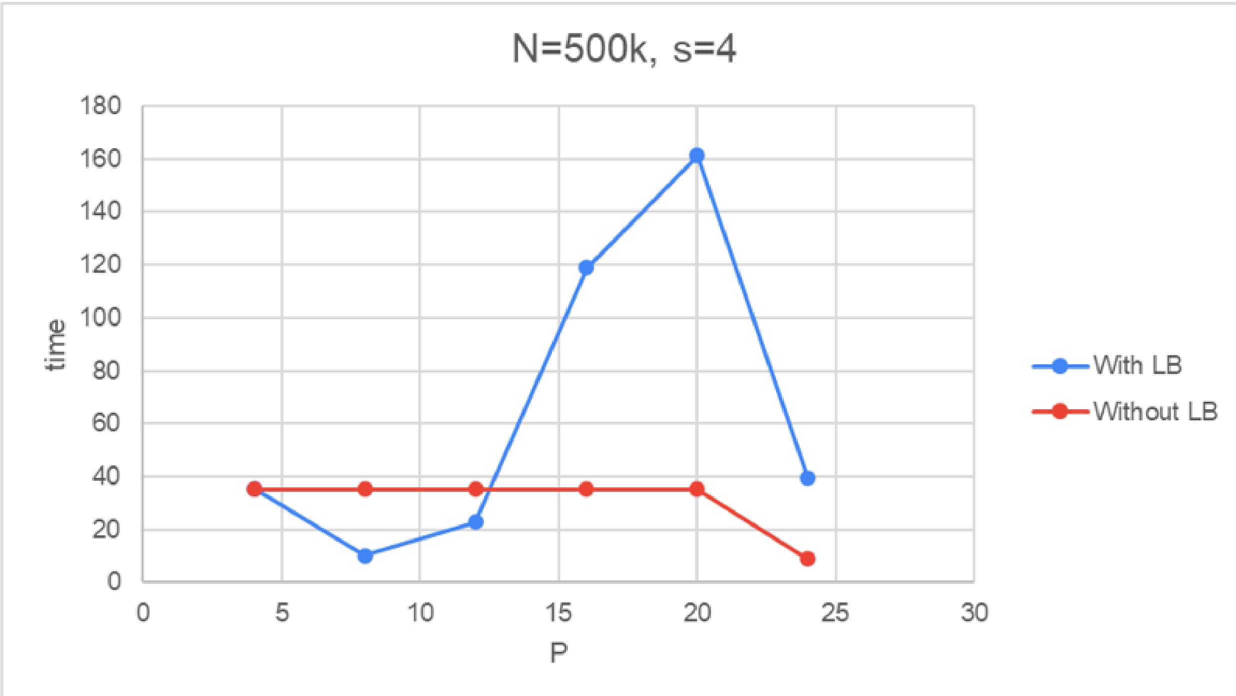


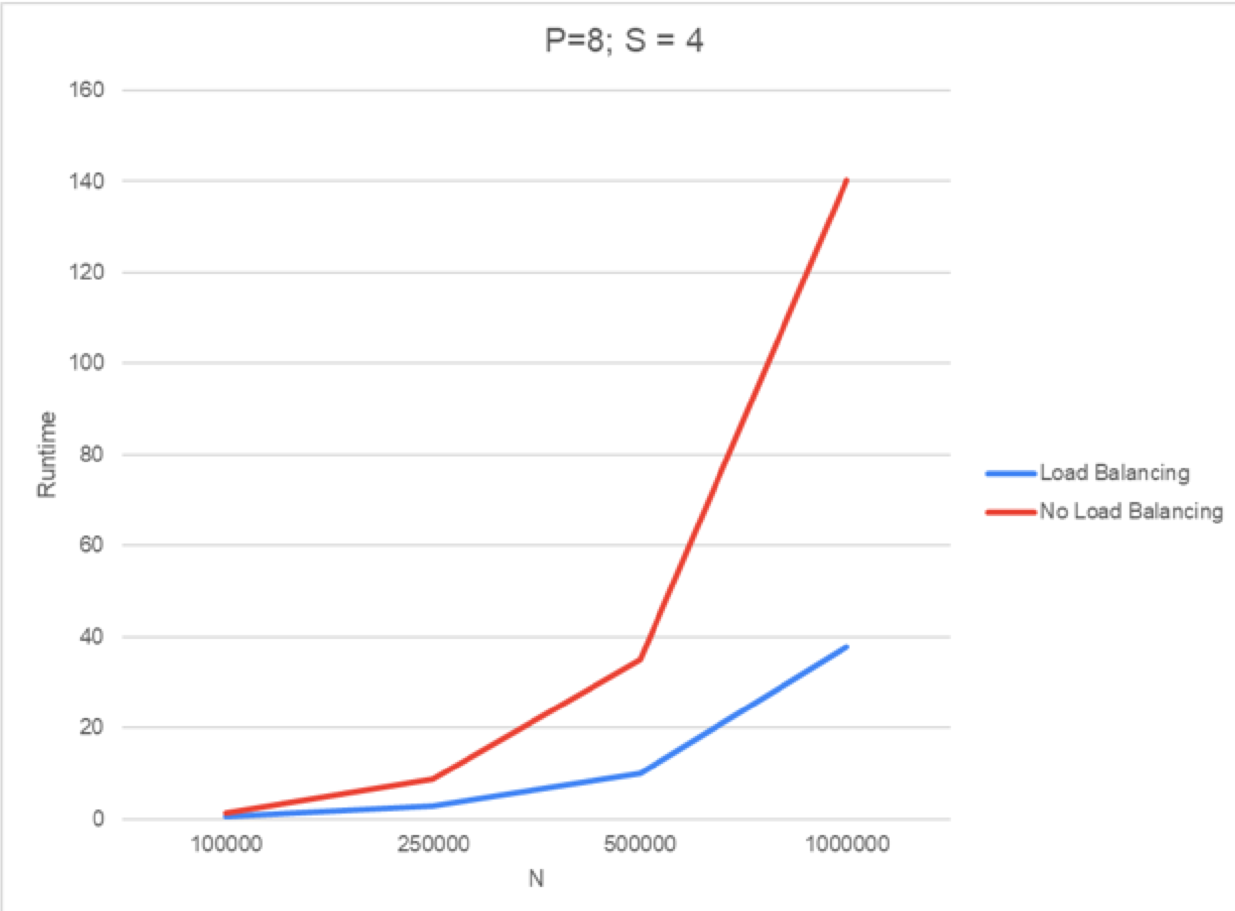
The above is a graph of how the runtime varies for different processor values for both Load Balanced and Non Load Balanced runs. We observe that the run time for loadbalanced runs has an inflexion point at 8 where the runtime for load balanced starts increasing as the number of processors increase.


N-250,000, skew = 4
Runtime
45
40
35
30
25
20
15
10
5
0
4
8
12
16
20
24
Processors
Load Balanced
No load balancing

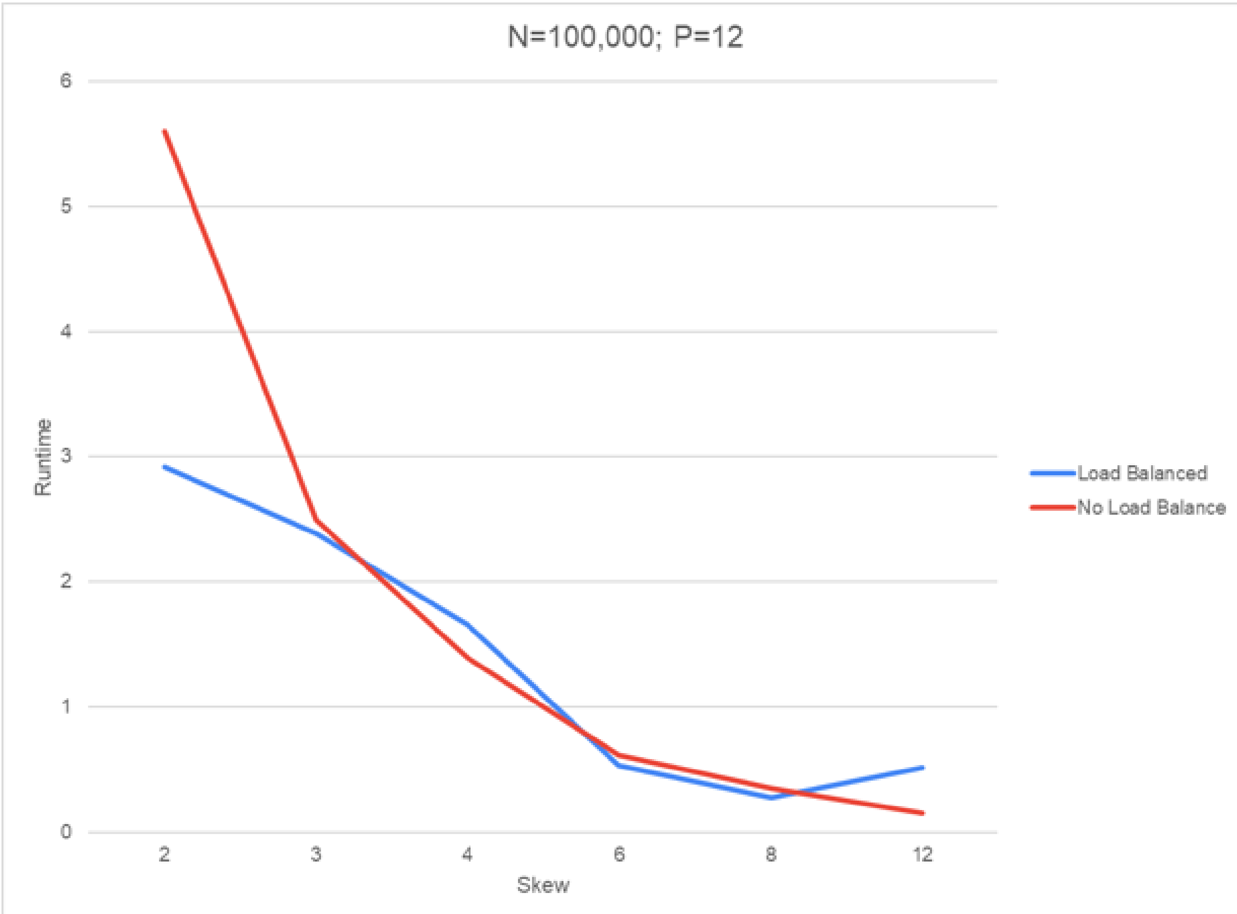

Execution time, N=500k, s=2
time (seconds)
160
140
120
100
80
60
40
20
0
0
5
10
15
20
25
P
with LB
without LB


Execution time, N=500k, s=4
time (seconds)
40
35
30
25
20
15
10
5
0
0
5
10
15
20
P
with LB
without LB

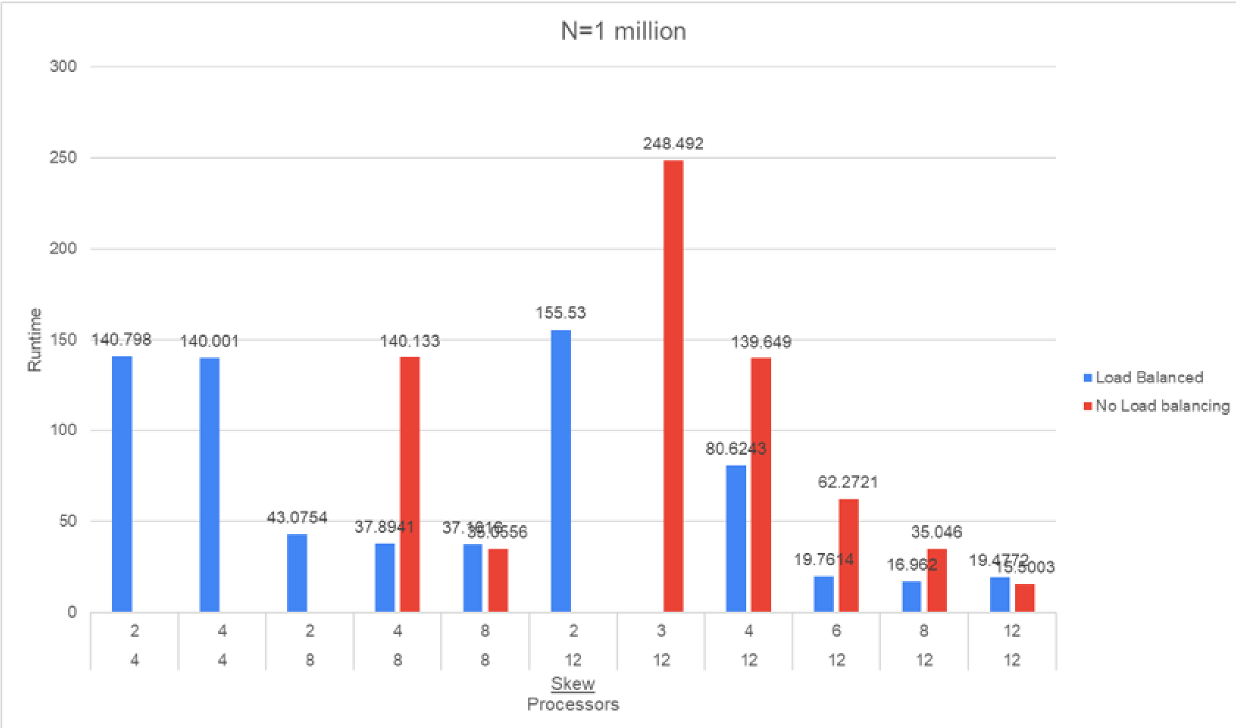

N=500k, s=4
time
P
With LB
Without LB
0
20
40
60
80
100
120
140
160
180
0
5
10
15
20
25
30


P=8; S = 4
160
140
120
100
80
60
40
20
0
Runtime
100000
250000
500000
1000000
N
Load Balancing
No Load Balancing

-



- For N=100K, and P=12, as the regular runtime becomes quite low, so does the difference.



- For N = 1 million, is where we see the most significant results - sometimes getting close to 4x improvement.

Time Complexity Analysis

The time complexity of the algorithm can be calculated taking into account the following factors:

- **Distribution time:** In the worst case, where the pivot ends as the last element of the array in each partition, one processor in each round of distribution will be sending $O(N)$ elements to another processor. Since there are $log(P)$ rounds, the time complexity will be:

$$T_D = O(cNlog(p)) \tag{1}$$

- **Load balancing time:** Load Balancing Worst Case Analysis:
  Key Points:
  - Operation has to be sequential. (0 to P-1 and back)
  - Total of N elements are passed at each stage
  - Number of stages = P

  Communication Time = $O(NP)$
  Merge Time = $O(NP)$
  Finding K minimum/maximum elements (K=N) = $O(PNlogK) = O(PNlogN)$
  Removing extra elements = $O(NP)$

$$T_B = 3 * O(NP) + O(PNlogN) \tag{2}$$
$$T_B = O(PNlogN) \tag{3}$$

- **Local sorting time:** At the end of the load balancing phase, each element has N/P elements. Since we're using insertion sort, the time complexity of the local sort phase is:

$$T_S = O((\frac{N}{P})^2) \tag{4}$$

- **Merging time:** Merging P N/P ordered sorted lists sequentially takes $O(N)$ time.

$$T_M = O(N) \tag{5}$$

Total time complexity T is:

$$T = T_D + T_B + T_S + T_M \tag{6}$$
$$T = O(cNlog(P) + PNlogN + (\frac{N}{P})^2 + N) \tag{7}$$

Entire Data:

All of the readings can be found at the following link in a better to read form - https://uic365-my.sharepoint.com/:x:/g/personal/mshah229_uic_edu/EY7p0pRpHC9Cuut07f5EsdMB7Uz32sxva70mcjIGTT2jSg

Type of Program	N	Processors	Skewness	Runtime
Load Balancing	100000	2	2	5.54
No Load Balancing	100000	2	2	5.51
Load Balancing	100000	4	2	1.4922
No Load Balancing	100000	4	2	5.50514
Load Balancing	100000	4	4	1.447
No Load Balancing	100000	4	4	1.38466
Load Balancing	100000	8	2	1.05696
No Load Balancing	100000	8	2	5.59481
Load Balancing	100000	8	4	0.602175
No Load Balancing	100000	8	4	1.39682
Load Balancing	100000	8	8	0.578465
No Load Balancing	100000	8	8	0.347145
Load Balancing	100000	12	2	2.91945
No Load Balancing	100000	12	2	5.59555
Load Balancing	100000	12	3	2.38817
No Load Balancing	100000	12	3	2.49622
Load Balancing	100000	12	4	1.6569
No Load Balancing	100000	12	4	1.39625

Load Balancing	100000	12	6	0.529486
No Load Balancing	100000	12	6	0.618961
Load Balancing	100000	12	8	0.276068
No Load Balancing	100000	12	8	0.347291
Load Balancing	100000	12	12	0.51197
No Load Balancing	100000	12	12	0.155601
Load Balancing	100000	16	2	8.44833
No Load Balancing	100000	16	2	5.59959
Load Balancing	100000	16	4	5.86795
No Load Balancing	100000	16	4	1.39707
Load Balancing	100000	16	8	0.675798
No Load Balancing	100000	16	8	0.347316
Load Balancing	100000	16	16	0.547045
No Load Balancing	100000	16	16	0.0873961
Load Balancing	100000	20	4	7.64855
No Load Balancing	100000	20	4	1.39692
Load Balancing	100000	20	5	4.23646
No Load Balancing	100000	20	5	0.887523
Load Balancing	100000	20	8	2.13299
No Load Balancing	100000	20	8	0.347645
Load Balancing	100000	20	10	0.65537
No Load Balancing	100000	20	10	0.221174
Load Balancing	100000	20	20	0.558142

No Load Balancing	100000	20	20	0.0555611
Load Balancing	100000	24	4	7.16518
No Load Balancing	100000	24	4	1.39653
Load Balancing	100000	24	6	3.23457
No Load Balancing	100000	24	6	0.617354
Load Balancing	100000	24	8	1.75384
No Load Balancing	100000	24	8	0.347241
Load Balancing	100000	24	12	0.755834
No Load Balancing	100000	24	12	0.155843
Load Balancing	100000	24	24	0.577911
No Load Balancing	100000	24	24	0.0388093
Load Balancing	100000	32	4	7.87344
No Load Balancing	100000	32	4	1.40259
Load Balancing	100000	32	8	2.09472
No Load Balancing	100000	32	8	0.34722
Load Balancing	100000	32	16	0.420257
No Load Balancing	100000	32	16	0.087415
Load Balancing	100000	32	32	0.606666
No Load Balancing	100000	32	32	0.0216634
Load Balancing	100000	40	8	2.30913
No Load Balancing	100000	40	8	0.347257
Load Balancing	100000	40	10	1.48944
No Load Balancing	100000	40	10	0.221291

Load Balancing	100000	40	20	0.345107
No Load Balancing	100000	40	20	0.0555696
Load Balancing	100000	40	40	0.626836
No Load Balancing	100000	40	40	0.0137861
Load Balancing	100000	48	8	2.46442
No Load Balancing	100000	48	8	0.347965
Load Balancing	100000	48	12	1.13905
No Load Balancing	100000	48	12	0.155445
Load Balancing	100000	48	24	0.345217
No Load Balancing	100000	48	24	0.0387962
Load Balancing	100000	48	48	0.644868
No Load Balancing	100000	48	48	0.00976515
Load Balancing	100000	56	8	2.58263
No Load Balancing	100000	56	8	0.347404
Load Balancing	100000	56	14	0.904606
No Load Balancing	100000	56	14	0.113093
Load Balancing	100000	56	28	0.316591
No Load Balancing	100000	56	28	0.0279541
Load Balancing	100000	56	56	0.658193
No Load Balancing	100000	56	56	0.00716996
Load Balancing	100000	64	8	2.66924
No Load Balancing	100000	64	8	0.347216
Load Balancing	100000	64	16	0.745795

No Load Balancing	100000	64	16	0.0878298
Load Balancing	100000	64	32	0.299933
No Load Balancing	100000	64	32	0.0216451
Load Balancing	100000	64	64	0.666353
No Load Balancing	100000	64	64	0.00539923
No Load Balancing	250000	24	12	0.947535
Load Balancing	250000	24	24	1.64216
No Load Balancing	250000	24	24	0.228313
Load Balancing	250000	32	4	43.5108
No Load Balancing	250000	32	4	8.74679
Load Balancing	250000	32	8	10.3371
No Load Balancing	250000	32	8	2.1798
Load Balancing	250000	32	16	1.94636
No Load Balancing	250000	32	16	0.539523
Load Balancing	250000	32	32	1.65149
No Load Balancing	250000	32	32	0.135485
Load Balancing	250000	40	4	46.2123
No Load Balancing	250000	40	4	8.74591
Load Balancing	250000	40	8	11.6138
No Load Balancing	250000	40	8	2.17924
Load Balancing	250000	40	10	7.12992
No Load Balancing	250000	40	10	1.39583
Load Balancing	250000	40	20	0.961335

No Load Balancing	250000	40	20	0.347228
Load Balancing	250000	40	40	1.68091
No Load Balancing	250000	40	40	0.0874233
Load Balancing	250000	48	8	12.5067
No Load Balancing	250000	48	8	2.17938
Load Balancing	250000	48	12	5.25961
No Load Balancing	250000	48	12	0.969368
Load Balancing	250000	48	24	1.20055
No Load Balancing	250000	48	24	0.233133
Load Balancing	250000	48	48	1.71096
No Load Balancing	250000	48	48	0.0616214
Load Balancing	250000	56	8	13.1734
No Load Balancing	250000	56	8	2.18003
Load Balancing	250000	56	14	4.08479
No Load Balancing	250000	56	14	0.710046
Load Balancing	250000	56	28	1.04518
No Load Balancing	250000	56	28	0.1776
Load Balancing	250000	56	56	1.72385
No Load Balancing	250000	56	56	0.0452905
Load Balancing	250000	64	8	13.6984
No Load Balancing	250000	64	8	2.1797
Load Balancing	250000	64	16	3.27793
No Load Balancing	250000	64	16	0.539927

Load Balancing	250000	64	32	0.949882
No Load Balancing	250000	64	32	0.135214
Load Balancing	250000	64	64	1.74114
No Load Balancing	250000	64	64	0.0346379
Load Balancing	500000	2	2	139.688
No Load Balancing	500000	2	2	139.65
Load Balancing	500000	4	2	35.1825
No Load Balancing	500000	4	2	138.939
Load Balancing	500000	4	4	35.5071
No Load Balancing	500000	4	4	35.0569
Load Balancing	500000	8	2	12.5496
No Load Balancing	500000	8	2	136.588
Load Balancing	500000	8	4	10.1619
No Load Balancing	500000	8	4	35.0626
Load Balancing	500000	8	8	10.0051
No Load Balancing	500000	8	8	8.75394
Load Balancing	500000	12	2	42.8729
No Load Balancing	500000	12	2	139.731
Load Balancing	500000	12	3	35.5006
No Load Balancing	500000	12	3	62.2641
Load Balancing	500000	12	4	22.6673
No Load Balancing	500000	12	4	35.0508
Load Balancing	500000	12	6	5.88473

No Load Balancing	500000	12	6	15.5125
Load Balancing	500000	12	8	4.56528
No Load Balancing	500000	12	8	8.74953
Load Balancing	500000	12	12	5.80705
No Load Balancing	500000	12	12	3.88015
Load Balancing	500000	16	2	169.774
No Load Balancing	500000	16	2	139.733
Load Balancing	500000	16	4	118.907
No Load Balancing	500000	16	4	35.0697
Load Balancing	500000	16	8	5.43033
No Load Balancing	500000	16	8	8.74935
Load Balancing	500000	16	16	4.63222
No Load Balancing	500000	16	16	2.18304
Load Balancing	500000	20	4	161.286
No Load Balancing	500000	20	4	35.0607
Load Balancing	500000	20	5	83.2098
No Load Balancing	500000	20	5	22.3923
Load Balancing	500000	20	8	39.1089
No Load Balancing	500000	20	8	8.74964
Load Balancing	500000	20	10	4.62737
No Load Balancing	500000	20	10	5.59351
Load Balancing	500000	20	20	4.09455
No Load Balancing	500000	20	20	1.39561

Load Balancing	500000	24	4	39.2411
No Load Balancing	500000	24	4	8.75759
Load Balancing	500000	24	6	16.5955
No Load Balancing	500000	24	6	3.87862
Load Balancing	500000	24	8	8.44129
No Load Balancing	500000	24	8	2.17906
Load Balancing	500000	24	24	3.856
No Load Balancing	500000	24	24	0.968924
Load Balancing	1000000	4	2	140.798
Load Balancing	1000000	4	4	140.001
Load Balancing	1000000	8	2	43.0754
Load Balancing	1000000	8	4	37.8941
No Load Balancing	1000000	8	4	140.133
Load Balancing	1000000	8	8	37.1616
No Load Balancing	1000000	8	8	35.0556
Load Balancing	1000000	12	2	155.53
No Load Balancing	1000000	12	3	248.492
Load Balancing	1000000	12	4	80.6243
No Load Balancing	1000000	12	4	139.649
Load Balancing	1000000	12	6	19.7614
No Load Balancing	1000000	12	6	62.2721
Load Balancing	1000000	12	8	16.962
No Load Balancing	1000000	12	8	35.046

Load Balancing	1000000	12	12	19.4772
No Load Balancing	1000000	12	12	15.5003
Load Balancing	1000000	16	4	460.76
No Load Balancing	1000000	16	4	139.684
Load Balancing	1000000	16	8	15.4982
No Load Balancing	1000000	16	8	35.049
Load Balancing	1000000	16	16	13.8014
No Load Balancing	1000000	16	16	8.75285
No Load Balancing	1000000	20	4	139.678
Load Balancing	1000000	20	5	320.91
No Load Balancing	1000000	20	5	89.3359
Load Balancing	1000000	20	8	148.786
No Load Balancing	1000000	20	8	35.0539
Load Balancing	1000000	20	10	12.3209
No Load Balancing	1000000	20	10	22.3554
Load Balancing	1000000	20	20	11.1502
No Load Balancing	1000000	20	20	5.60126

Conclusion

- From our experiments, we can see that the advantages of load balancing are highly dependent on the distribution and amount of data.

- For cases where we have a large number of processors such that each processor has relatively low data sizes, the runtime with load balancing is sometimes not better than without load balancing.

- For cases where we have fewer processors when compared to the input size, we see good improvement in runtime compared to without the load balancing.

- Therefore, we can surmise that the load balancing is necessary only when there is a very high imbalance of data across the processors, and it might be better to not balance when there are a low number of processors/low number of optimal elements per processor

- The overhead caused by the load information exchange (LIE) and subsequent message passing is not profitable if the data has a skew > 2. This means that our algorithm has scope for improvement for more balanced loads.

- Load balancing is also not profitable when the number of elements to be sorted is < 500k. In this case, we're better sorting locally than doing message passing

- The current algorithm has a number of tunable parameters like the number of rounds to stop before proceeding with the insertion sort. We can further experiment with adjusting these parameters to achieve better results.