Sorting random numbers in distributed systems

Kanan Jafarli, Ibrahim Yunuslu

May 13, 2021

1 Application of Distributed Systems In Speeding Up Sorting Programs

This is a comprehensive algorithm in speeding up a computation intensive problem using distributed system concept. The program can be speed up by distributing the sorting efforts using numbers of process and thread, which can be located within a machine or spread to different machines. The load balancer balance the jobs and distribute them to three different servers, and then collects back the results. The three servers uses a merge sort algorithm and it also implements multithreading for concurrent job processing in giving faster results.

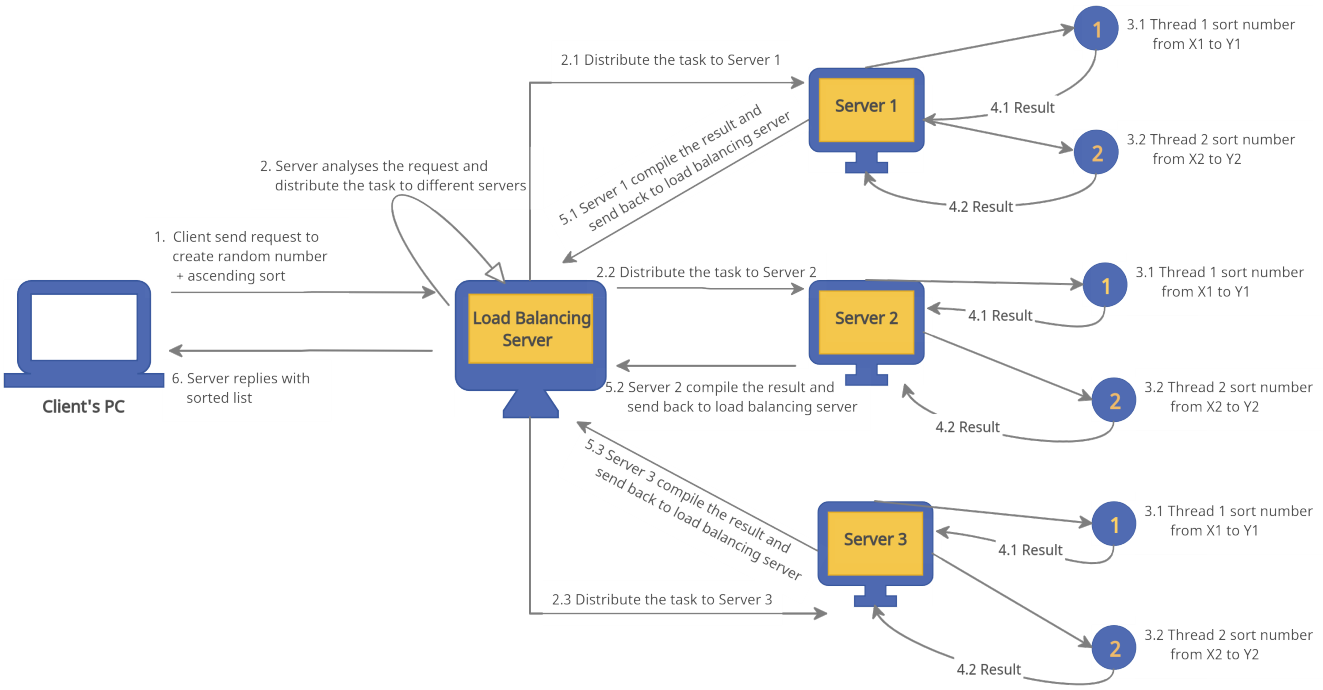


Figure 1: Structure, link to picture

Client sends a request to LoadBalancingServer which gives range boundaries (min and max values) to create random integer numbers in ascending sort. LoadBalancingServer analyses the request that generates random integers based on client defined range and distributes the random integers to the three different servers after balancing the numbers of random integers among servers. Each server also distribute the random integers to different server workers (threads) for giving faster results. Server workers sort numbers using merge sort and send back results to servers. Each server do final sort for the results obtained from server workers and send back to load balancing server. At the end, LoadBalancingServer collects back the results and replies to client with sorted list. Client can resend request to LoadBalancingServer again.

2 Merge Sort

Merge Sort is one of the most popular and an efficient sorting algorithms that is based on the principle of *Divide and Conquer Algorithm*. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

2.1 Algorithm

```
MergeSort(arr[], l,  r)
If r > l
  1. Find the middle point to divide the array into two halves:
        middle m = l+ (r-l)/2
  2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
  3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
  4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

2.2 How Merge Sort works?

To understand merge sort, we take an unsorted array as the following

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

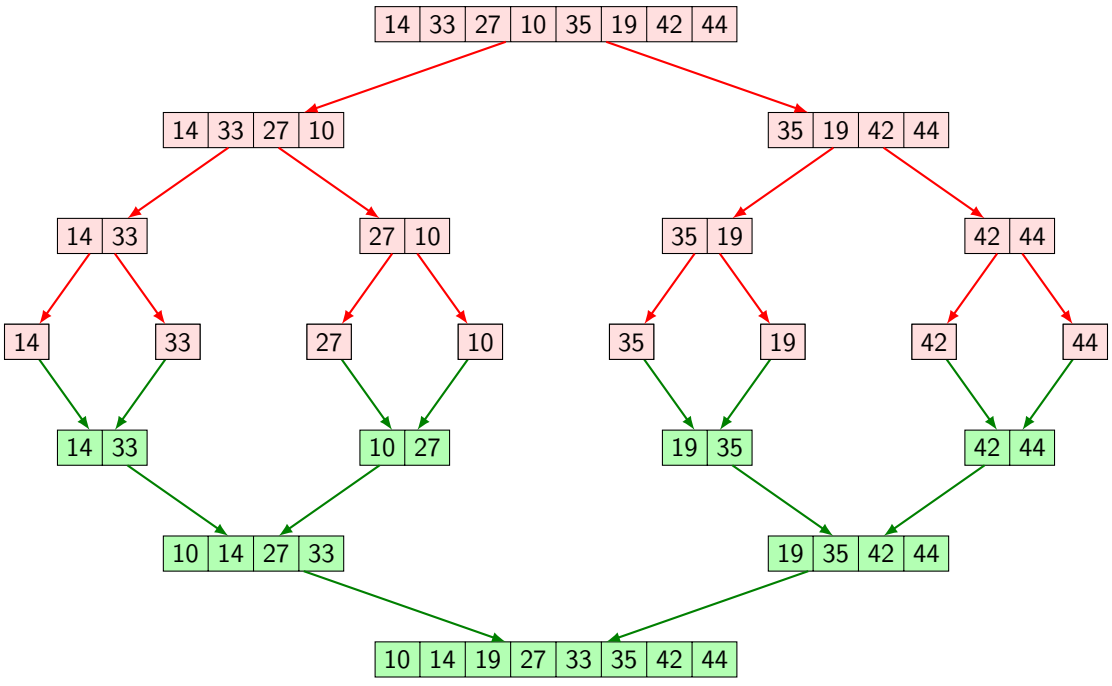
We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this.



3 Code explanation

3.1 How to run

```
// to compile programs
javac Client.java LoadBalancingServer.java Server1.java Server2.java Server3.java

// to execute the generated main class file
java Client
```

3.2 A list of the main functions

- **generateRandomNum(...)** - It is the function of *LoadBalancingServer* class. It takes min and max integer values as arguments and generate random integers in an array. Then this function pass generated integers into *getSortedResults*.
- **getSortedResults(...)** - It is the function of *LoadBalancingServer* class. It takes an 1d array of generated integers as argument and distributes tasks to three different servers for to get sorted results. Firstly, creates 3 1d separate integer arrays and load integers from generated array to these arrays. Then the function creates 3 different servers and distributes 3 separate integer arrays among servers.
- **mainDriver(...)** - It is the function of *Server* classes. It takes 1d array as argument, It creates two different server workers or threads and distributes array of integers to these threads. Finally, this function pass threads into *finalMerge* for sorted and also records computation time.
- **finalMerge(...)** - It is the function of *Server* classes. It takes two 1d array as arguments and sorts them in ascending. Finally, compiles results to *LoadBalancingServer*.
- **mergeSort(...)** - It is the function of *ServerWorker* class which extends *Thread* class. It takes 1d array as argument and sorts this array using merge sort algorithm.

3.3 Error cases

- **definable range format :** boundaries (min and max values) must to integer numbers, program returns *InputMismatchException* error and prints *"You have entered an invalid data."* on the terminal, when boundaries are string and non-integer numbers. Max value must be bigger than min value, otherwise program prints *"MAX must be greater than MIN"* on the terminal and wait for new max value.
- **resend request format :** for to send another request enter *1* or enter *2* for to exit program. If entered value is different from 1 and 2 or is string, program returns *InputMismatchException* error and prints *"You have entered an invalid data."* on the terminal.

4 Architecture

Figure 2: UML Class diagram

Class	Description	Attributes	Methods
Client	Send request to create random numbers in ascending sort.	min and max values of array	method for to enter min and max values of array
LoadBalancing Server	Analyses the request and distribute the task to different servers, then collects back the results.	-	methods for to generate random numbers, distribution and to get result from server
Server	Distribute the task to different server workers (threads), then sort results obtained from workers and send back to load balancing server.	1d array	methods for distribution and final merge
ServerWorker	Compile the result and send back to server.	1d array	methods for merge sort and method for to get array (like getter)

Table 1: Different classes, their attributes and methods