Parallel sorting using gRPC

- Preetham Salehundam

About Parallelism

The main idea of this project is to parallelize a sequential execution of an algorithm.

What is Parallelism?

The ability to execute more than one command with in the same clock cycle.

- Possible?
- Yes, by using Multiprocessor systems and Multi-core CPU's.

Majority of the parallel execution environments have processes running on different cores of a CPU or on different processors.

Sharing memory is not always a viable situation in parallel execution environments. Hence, there is a need for a fast and efficient communication protocol between the processes.

Sequential Min-Max Sort

- Sorting based on positioning the elements based on minimum and maximum element in the list

Algorithm

- Step 1 : Set P to 0 and Q to N-1 where ‘N’ is the number of elements
- Step 2 : While P < Q
  - REPEAT 3 to 6
- Step 3: Min-sorting - Move lesser value items to the left of the list
- Step 4: Max-sorting - Move higher value items to the right of the list
- Step 5: Increment P and Decrement Q
- Step 6 : END While
- Step 7: Print sorted list

Min sorting

- Step 1: Set “**Min**” to item in P^{th} location of the list (min = list[p])
- Step 2: for each element from index P to Q

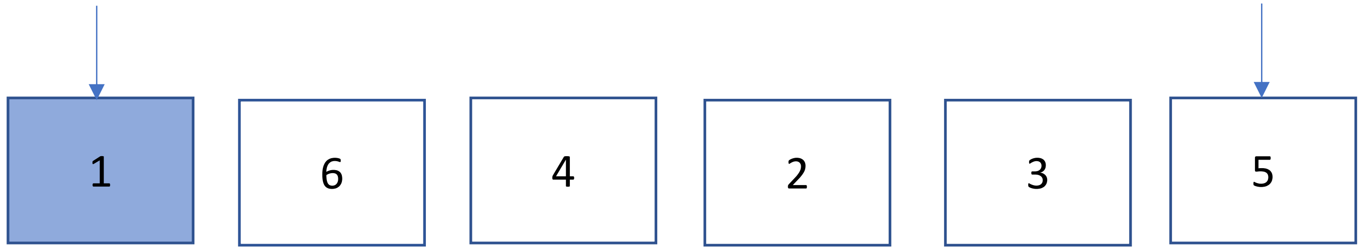
 - Compare with “**Min**”

- Step 3: if an item < **Min**
    - Step 3a : **Swap** the **corresponding item** with **Min**
    - Step 3b: Set **Min** to value of the corresponding item
- Step4 : Repeat steps 2 and 3 i.e. comparing and swapping

Max sorting

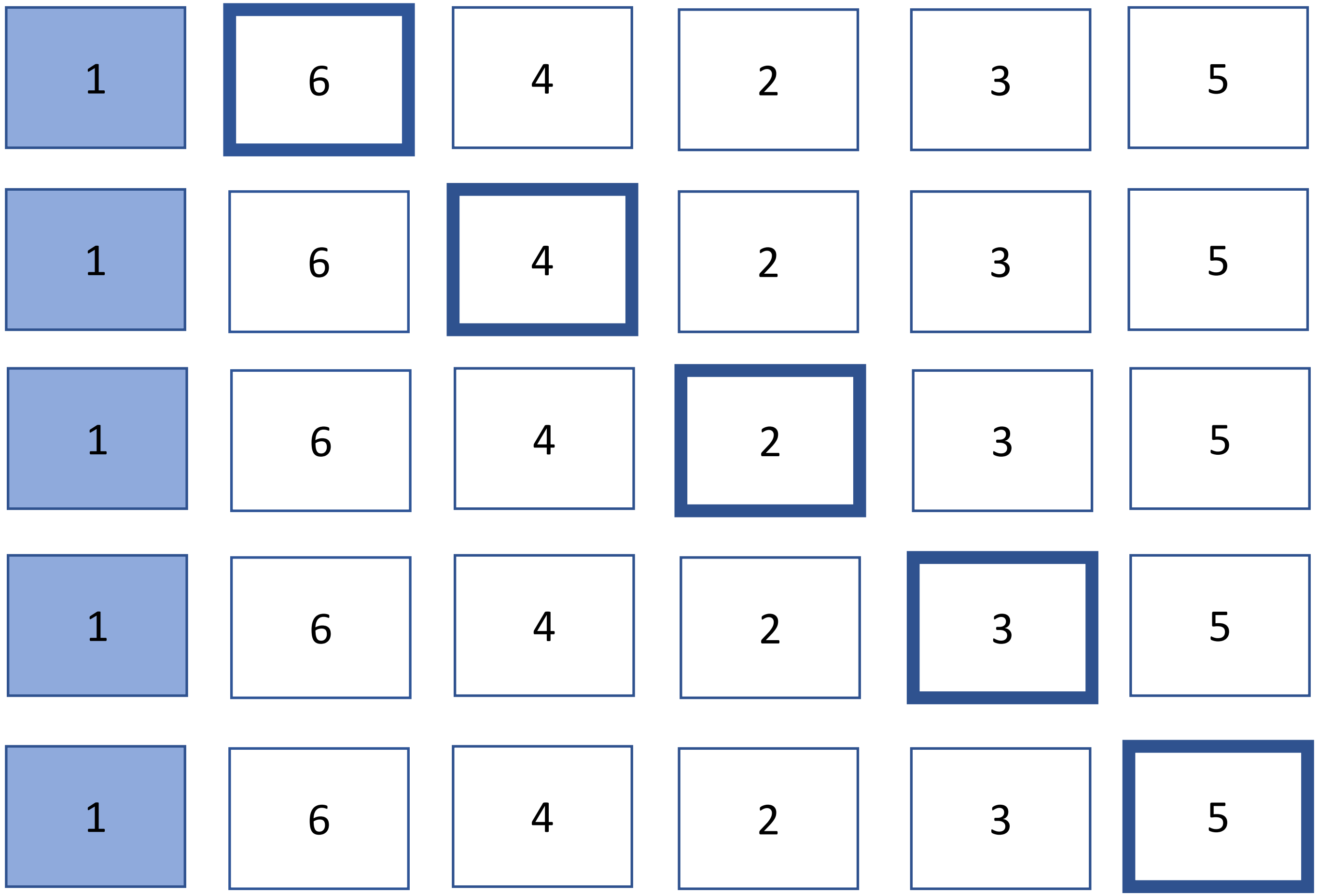
- Step 1: Set ‘**Max**” to value of an item at Q^{th} location in the list
- Step 2: for each element from index P to Q
  - compare elements with **“Max”**
- Step 3: if an item > **Max**
  - Step 3a : **Swap** the **corresponding item** with **Max**
  - Step 3b: Set **Max** to the value of the corresponding item
- Step4 : Repeat steps 2 and 3 i.e. comparing and swapping

In action
Pass 1
Min-sort



Min = 1

Compare with all items from 0th index to 5th index of the list



6 < 1 - False

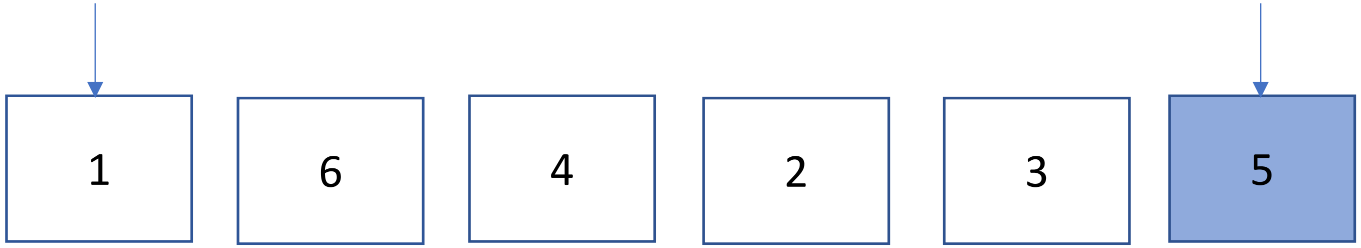
4 < 1 - False

2 < 1 - False

3 < 1 - False

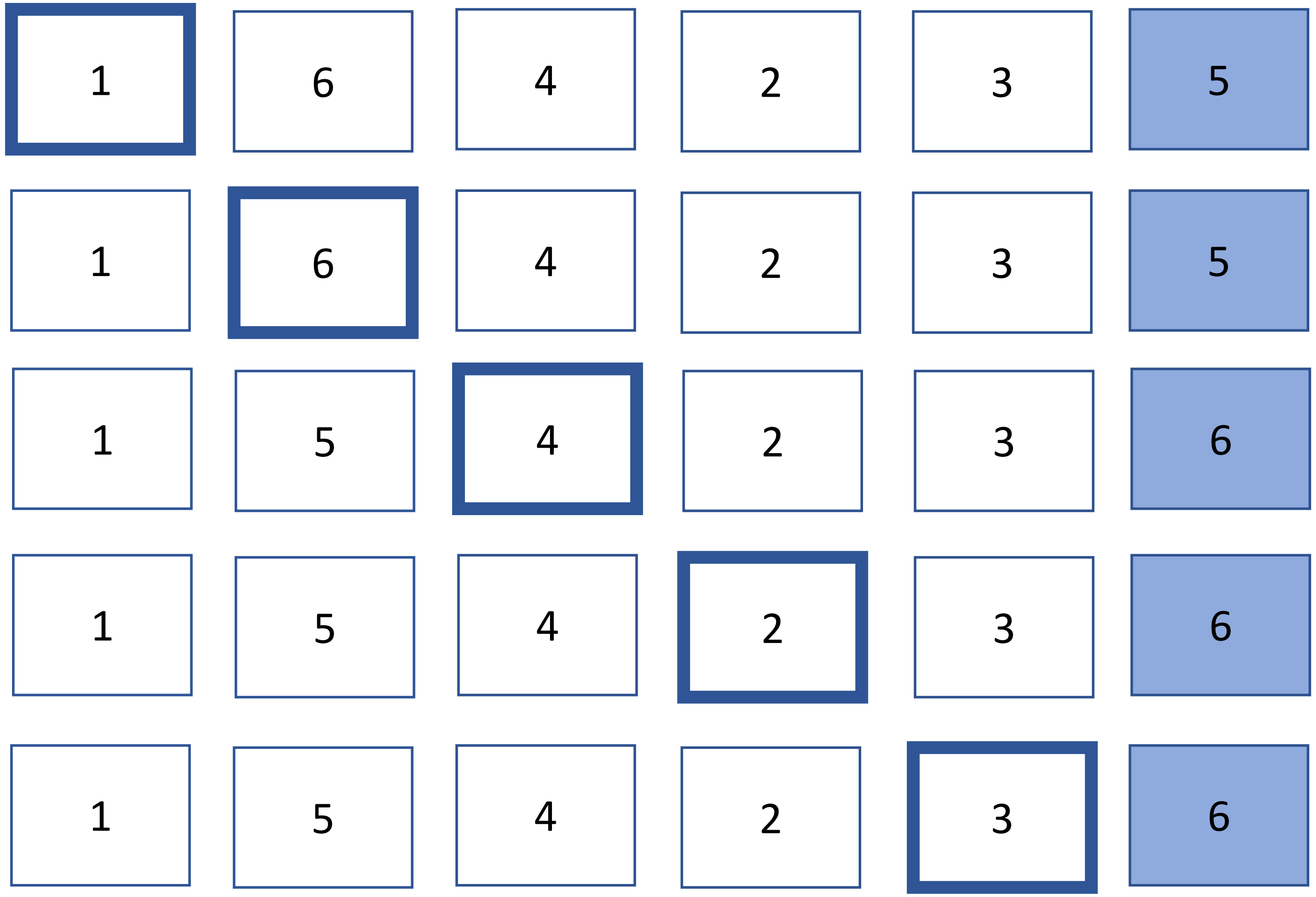
5 < 1 - False

In action
Pass 1
Max-Sort



Max = 5

Compare with all items from 0^{th} index to 5^{th} index of the list



1 > 5 - False

6> 5 – Swap 5 with 6

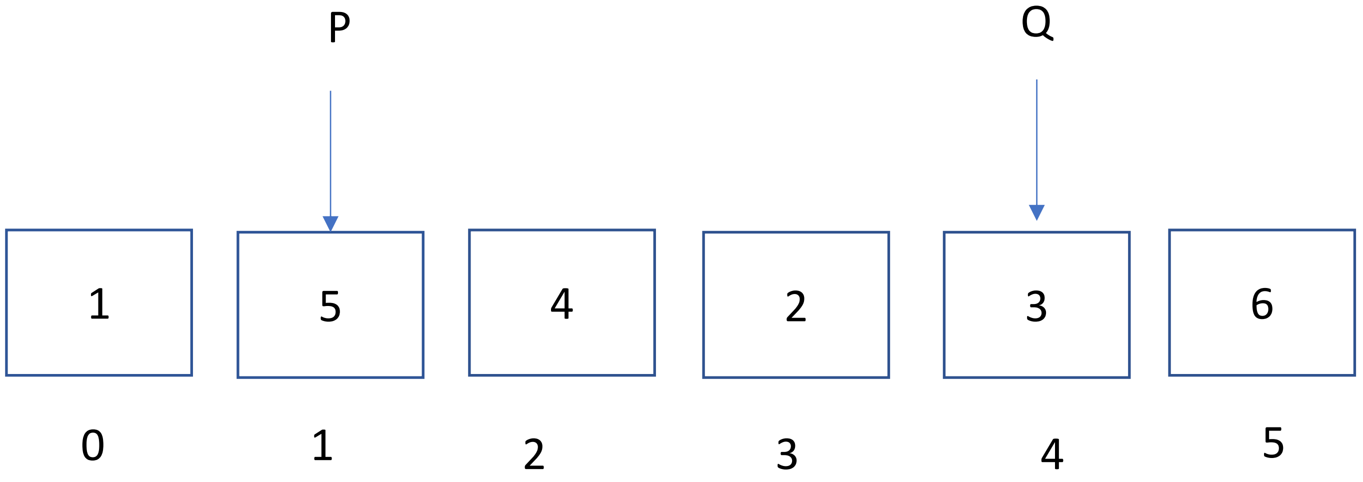
set **Max** = 6

4 > 6 - False

2 > 6 - False

3 > 6 - False

After First Pass



Increment P to index 1
Decrement Q to index 4

In action
Pass 2
Min-Sort

1	5	4	2	3	6	**Min** = 5

Compare with all items from 1st index to 4th index of the list

1	5	4	2	3	6	4 < 5 - Swap 5 and 4 Set **Min** = 4
1	4	5	2	3	6	2< 4 – Swap 2 and 4 Set **Min** = 2
1	2	5	4	3	6	3 < 2- False
1	2	5	4	3	6	END

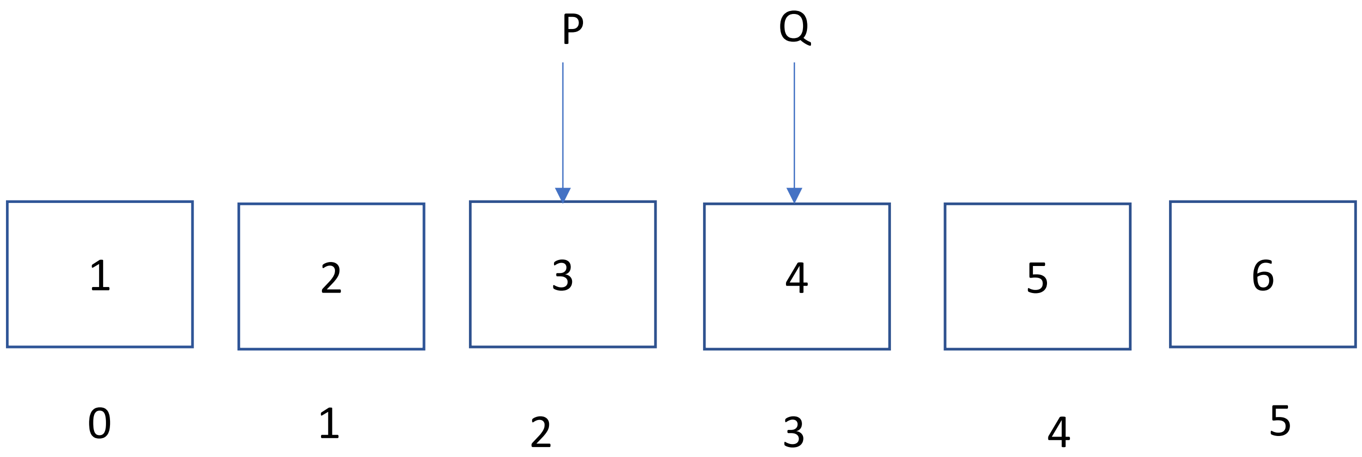
In action
Pass 2
Max-Sort

1	2	5	4	3	6	**Max** = 3

Compare with all items from 1st index to 4th index of the list

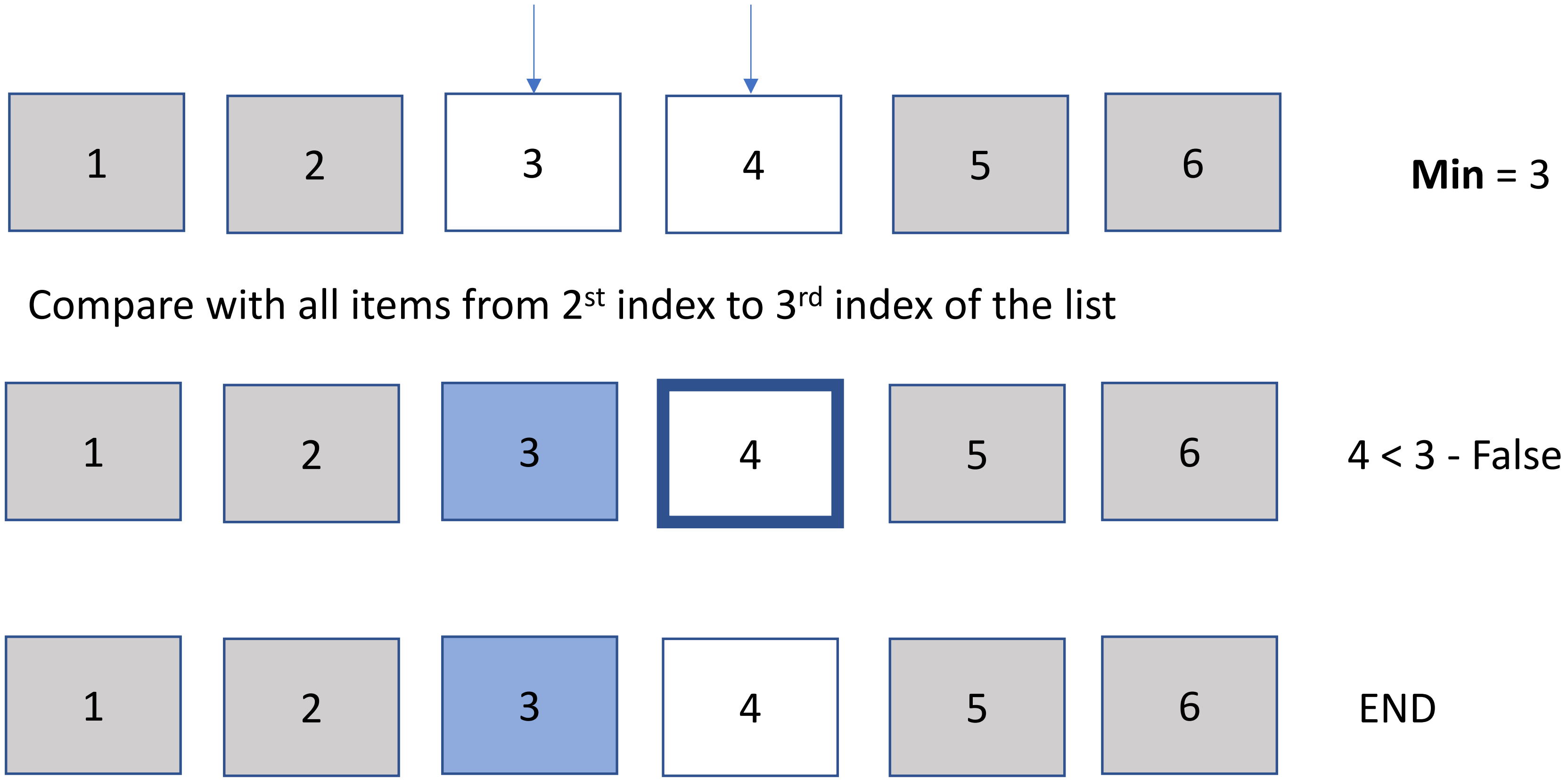
1	2	5	4	3	6	2 > 3 - False
1	2	5	4	3	6	5 > 3 – Swap Set **Max** = 5
1	2	3	4	5	6	4 > 5 - False
1	2	3	4	5	6	END

After Second Pass



Increment P to index 2
Decrement Q to index 3

In action
Pass 3
Min-Sort



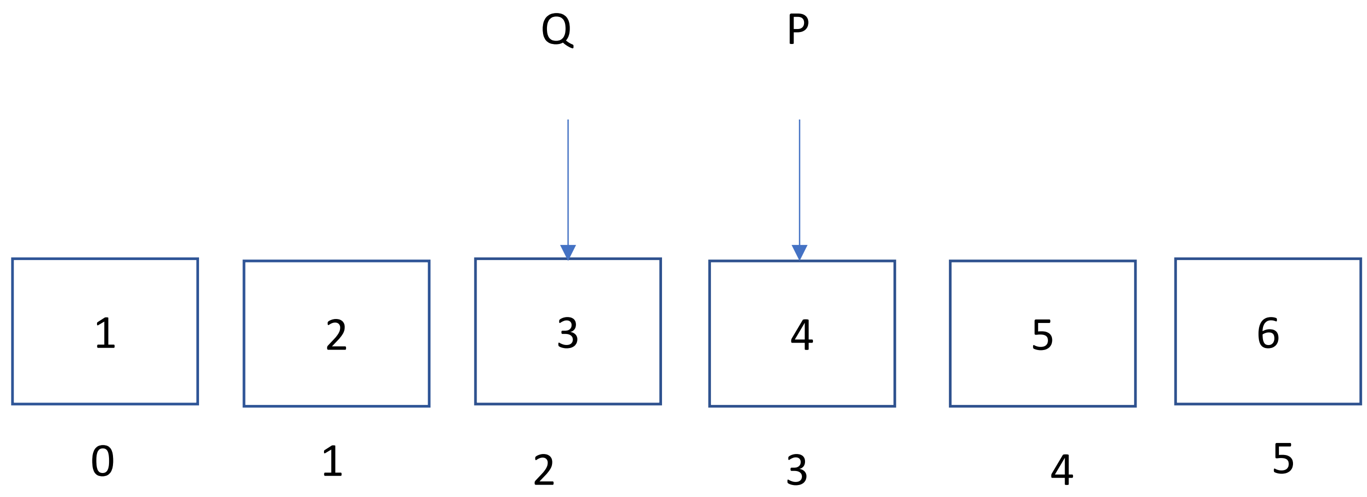
In action
Pass 3
Max-Sort

1	2	3	4	5	6	**Max** = 4

Compare with all items from 2nd index to 3rd index of the list

1	2	3	4	5	6	3 > 4 - False
1	2	3	4	5	6	END

After Third Pass



Increment P to index 3.
Decrement Q to index 2.

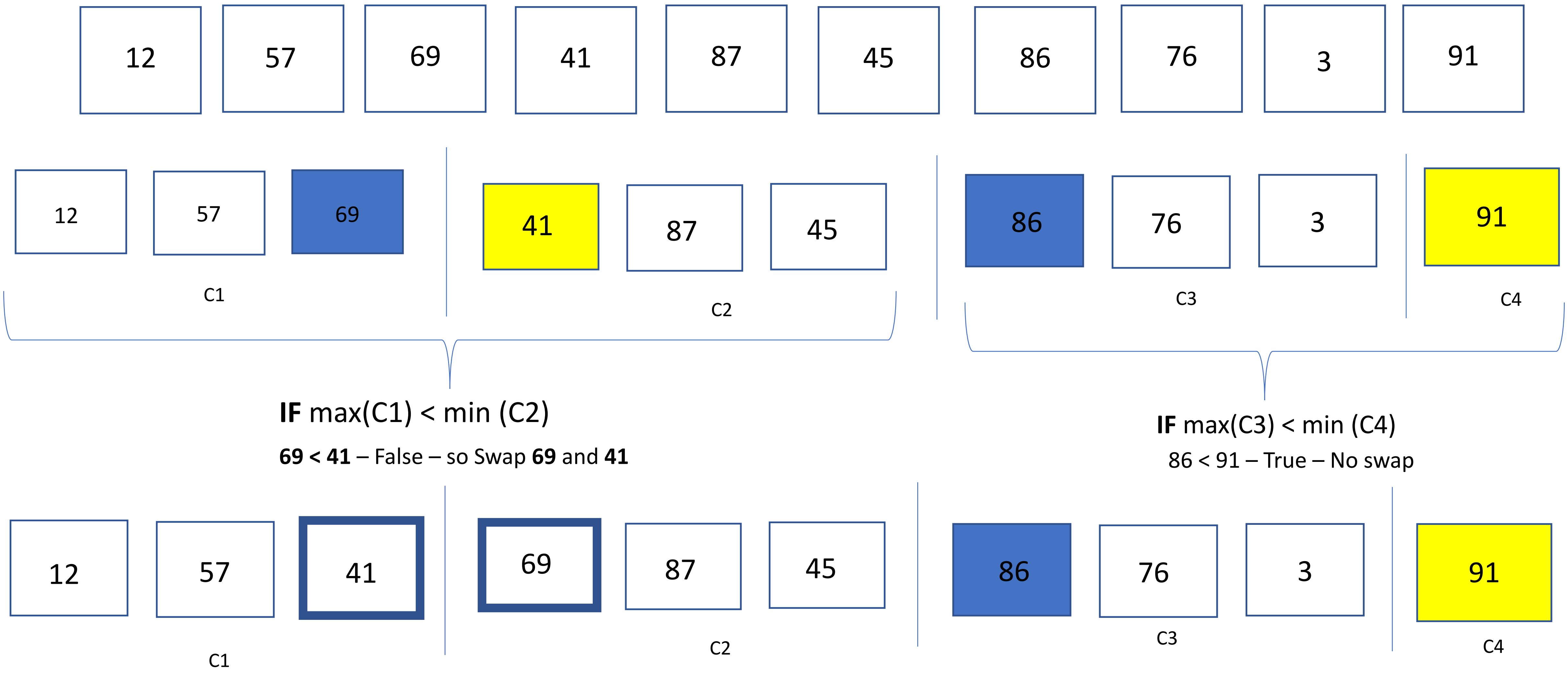
This **violates the P<Q condition** and the algorithm stops.

The list is thus sorted.

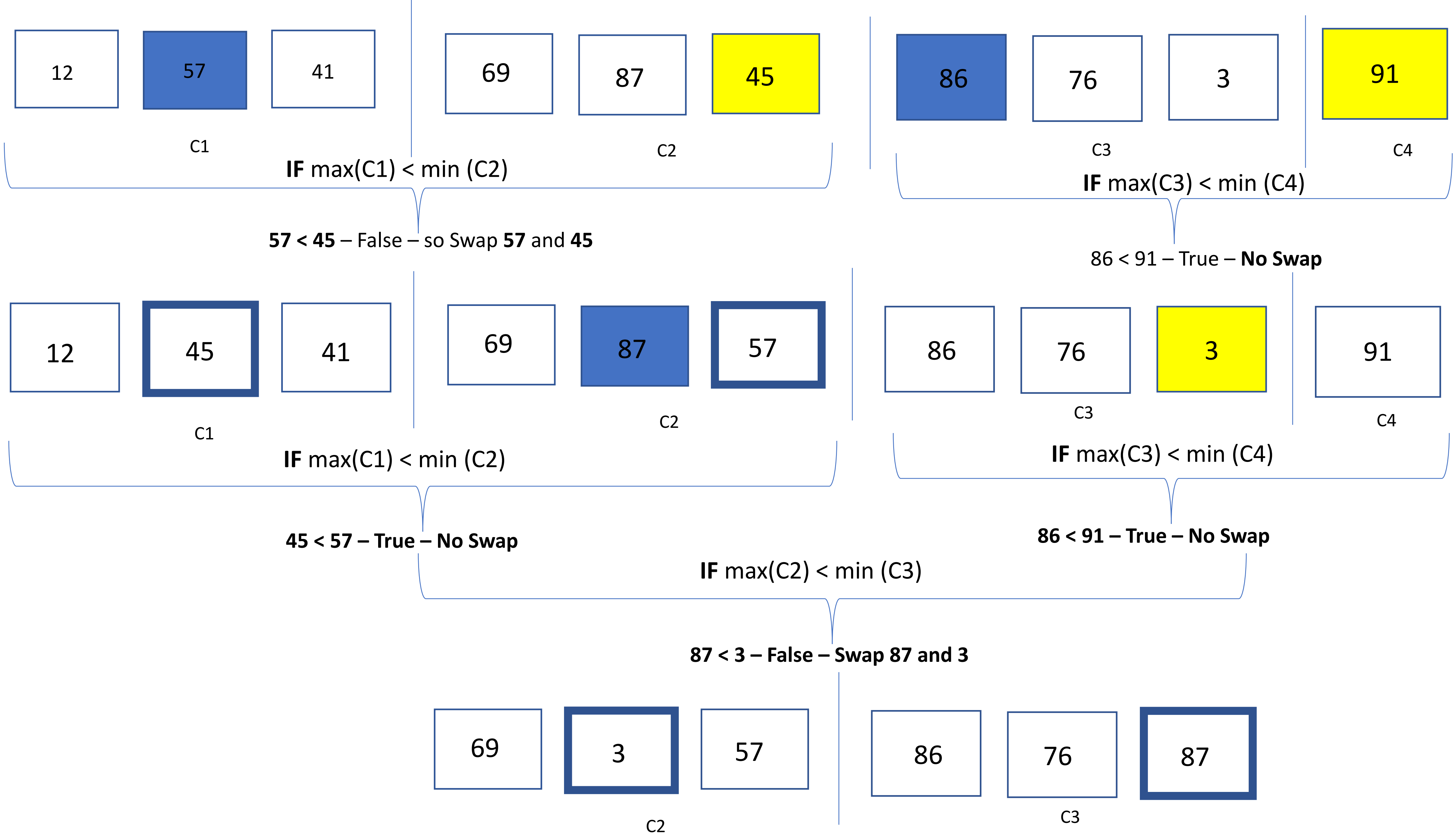
Parallel Sorting Algorithm

- **Step 1** — Split the given list into N-parts
- **Step 2** — Index the parts/chunks (Resource Hierarchy)
- **Step 3** — While all pairs of chunks satisfy max(chunk n) > min(chunk n+1) Repeat 4 & 5
- **Step 4** — Choose a pair and while max(chunk n) > min(chunk n+1)
- **Step 5** — Swap the max element of chunk N with min of chunk n+1
- **Step 6** — End when max(chunk n) <= min(chunk n+1)
- **Step 7** — Sort individual chunks

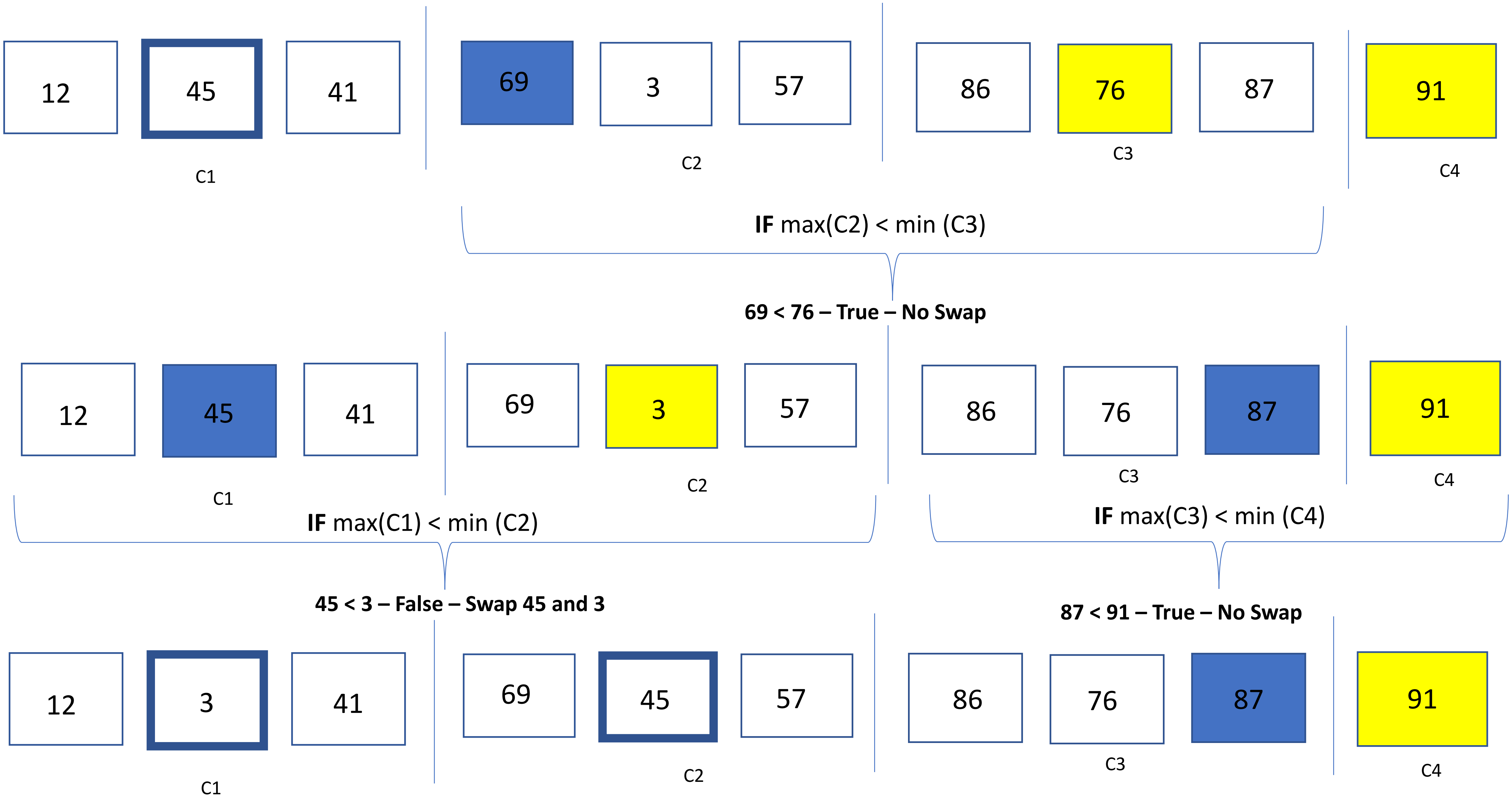
THE SHUFFLE

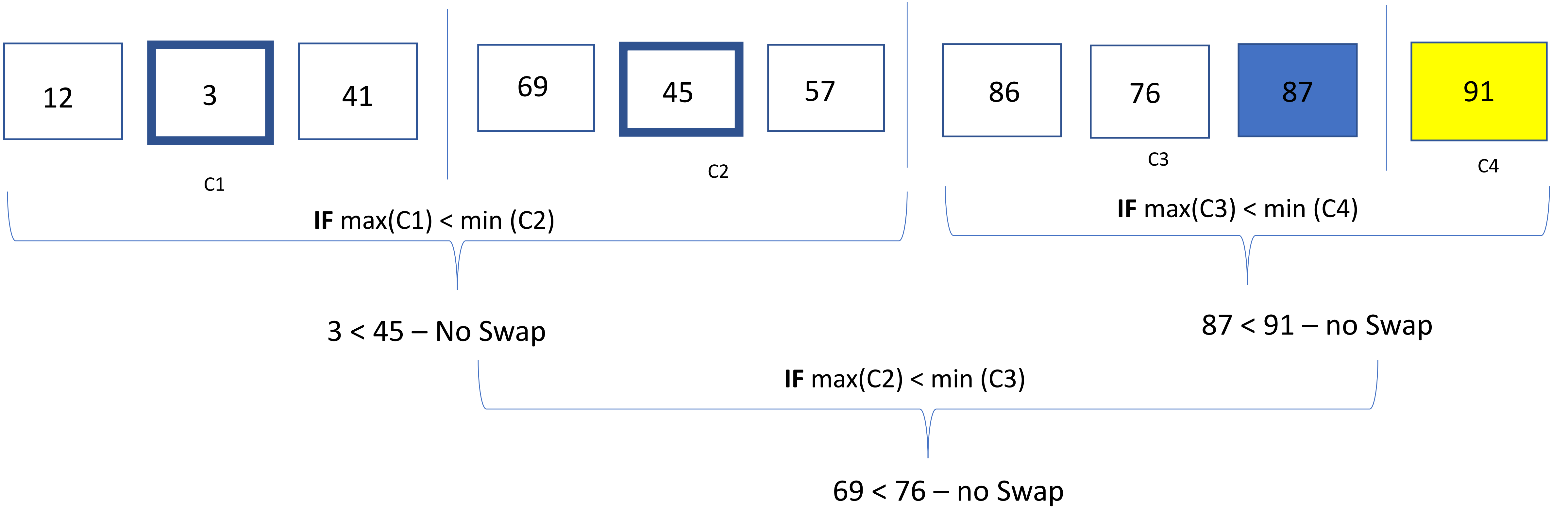


12 57 69 41 87 45 86 76 3 91
12 57 69
C1
41 87 45
C2
86 76 3
C3
91
C4
IF max(C1) < min (C2)
69 < 41 – False – so Swap **69** and **41**
IF max(C3) < min (C4)
86 < 91 – True – No swap
12 57 41
C1
69 87 45
C2
86 76 3
C3
91
C4

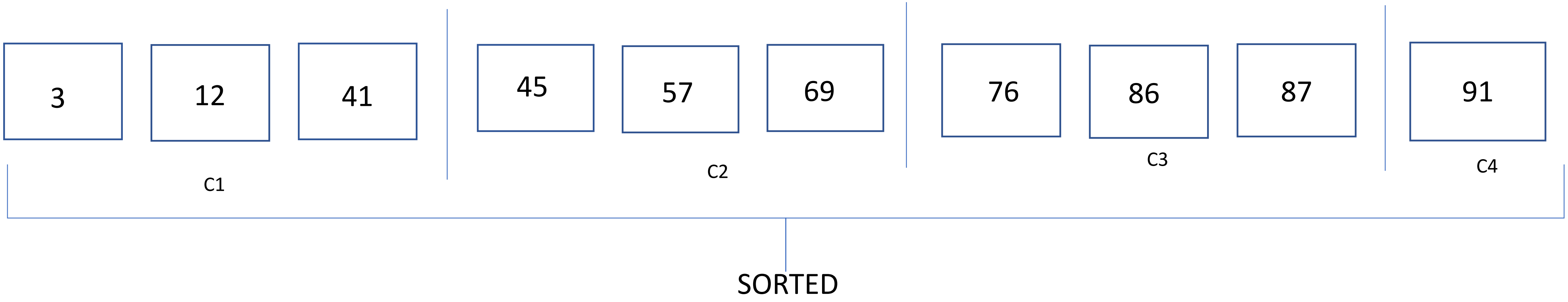


12
57
41
69
87
45
86
76
3
91
C1
C2
C3
C4
IF max(C1) < min (C2)
IF max(C3) < min (C4)
57 < 45 – False – so Swap 57 and 45
86 < 91 – True – No Swap
12
45
41
69
87
57
86
76
3
91
C1
C2
C3
C4
IF max(C1) < min (C2)
IF max(C3) < min (C4)
45 < 57 – True – No Swap
86 < 91 – True – No Swap
IF max(C2) < min (C3)
87 < 3 – False – Swap 87 and 3
69
3
57
86
76
87
C2
C3


12
45
41
C1
69
3
57
C2
86
76
87
C3
91
C4
IF max(C2) < min (C3)
69 < 76 – True – No Swap
12
45
41
C1
69
3
57
C2
86
76
87
C3
91
C4
IF max(C1) < min (C2)
IF max(C3) < min (C4)
45 < 3 – False – Swap 45 and 3
87 < 91 – True – No Swap
12
3
41
C1
69
45
57
C2
86
76
87
C3
91
C4



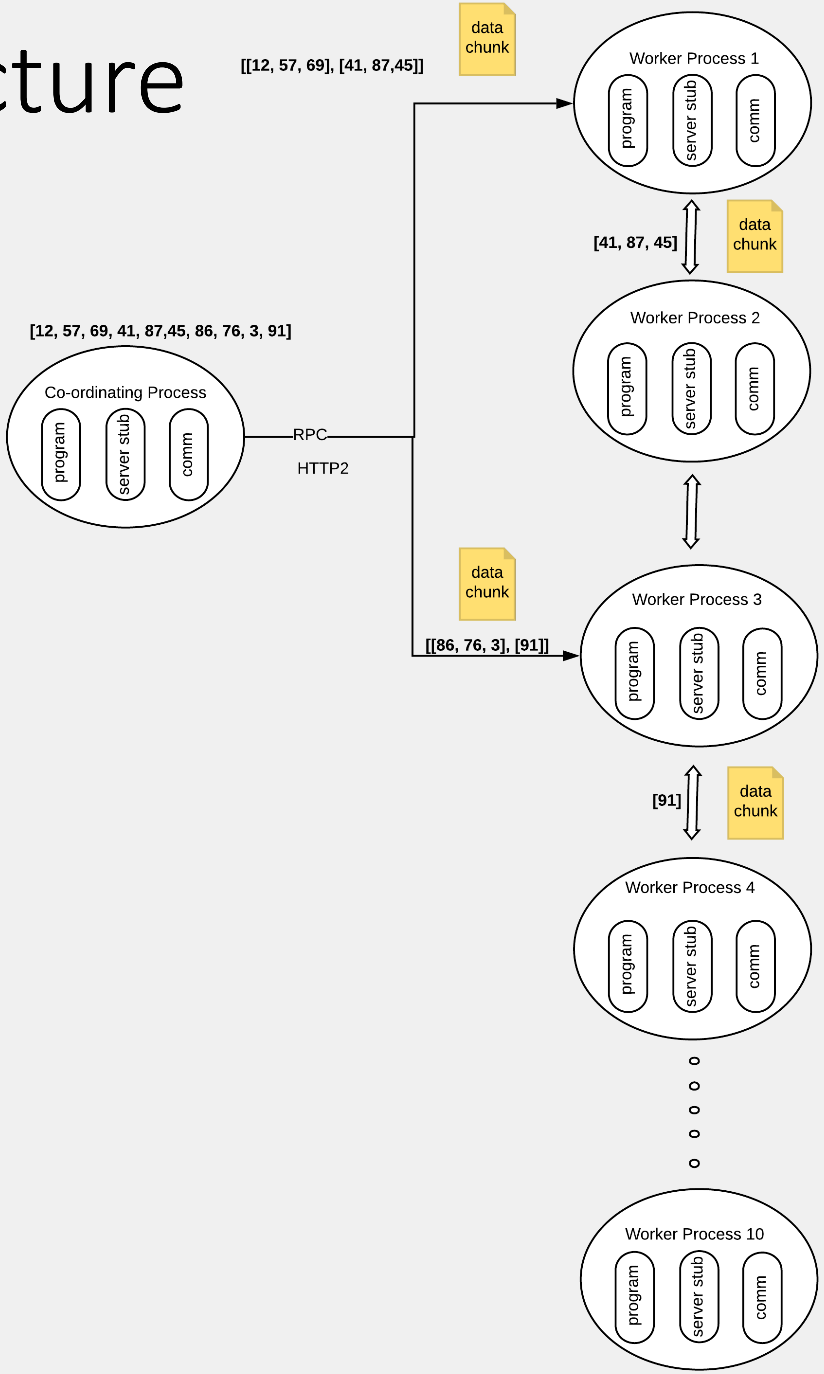
Now Sort individual chunks using min-max sort mentioned in the slides



Implementation Specifics

- There are many inter-process communication protocols like
    - Remote Method Invocation,
    - Remote Procedure call,
    - REST,
    - Message passing interface etc.,
- I’ve chosen to use **Google’s gRPC** due to its **HTTP2 support** and optimized memory foot print.
- I have used **Python Programming Language** to build this Parallel execution environment.
- Pros
    - Works across platforms and languages
- Cons
    - Bit of a learning curve. Lack of proper documentation.

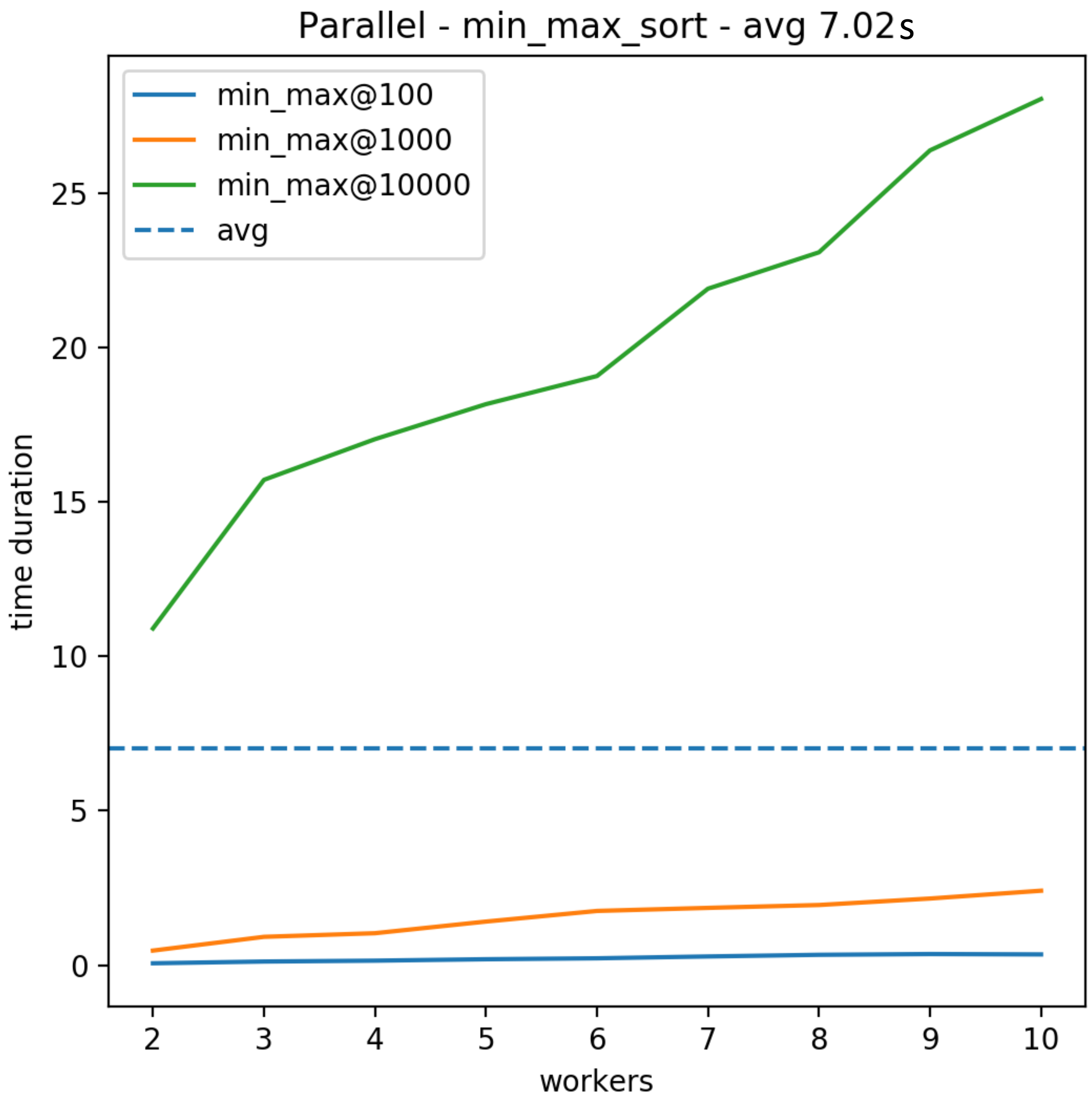
Architecture



List of RPC methods

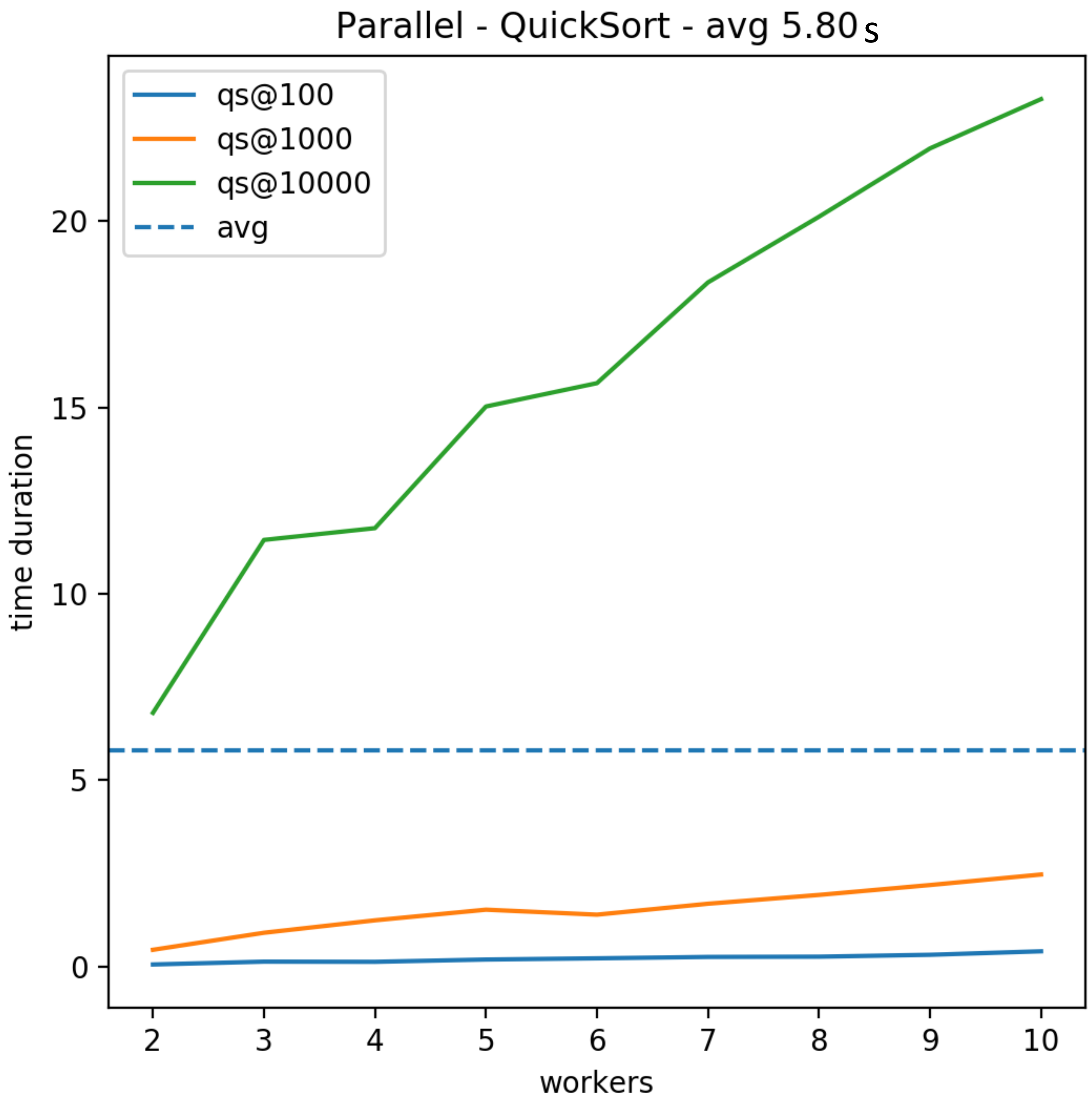
- get_max
- get_min
- Swap
- remote_sort
- process_data
- get_partial_sorted_data
- start_connection
- end_connection
- fetch_result_from_peer

Analysis

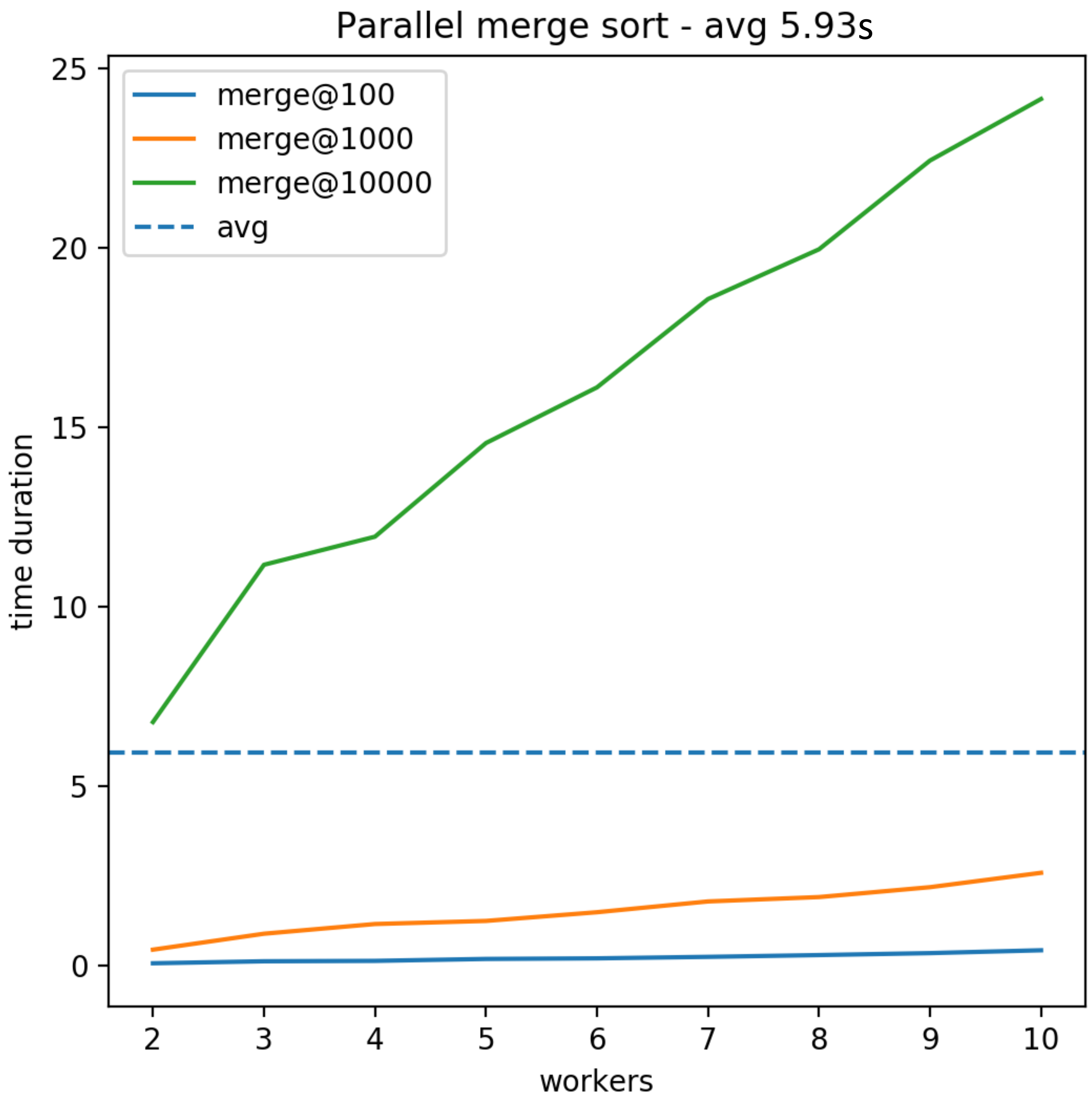


Parallel - min_max_sort - avg 7.02s
min_max@100
min_max@1000
min_max@10000
avg
time duration
workers
0
5
10
15
20
25
2
3
4
5
6
7
8
9
10

Analysis

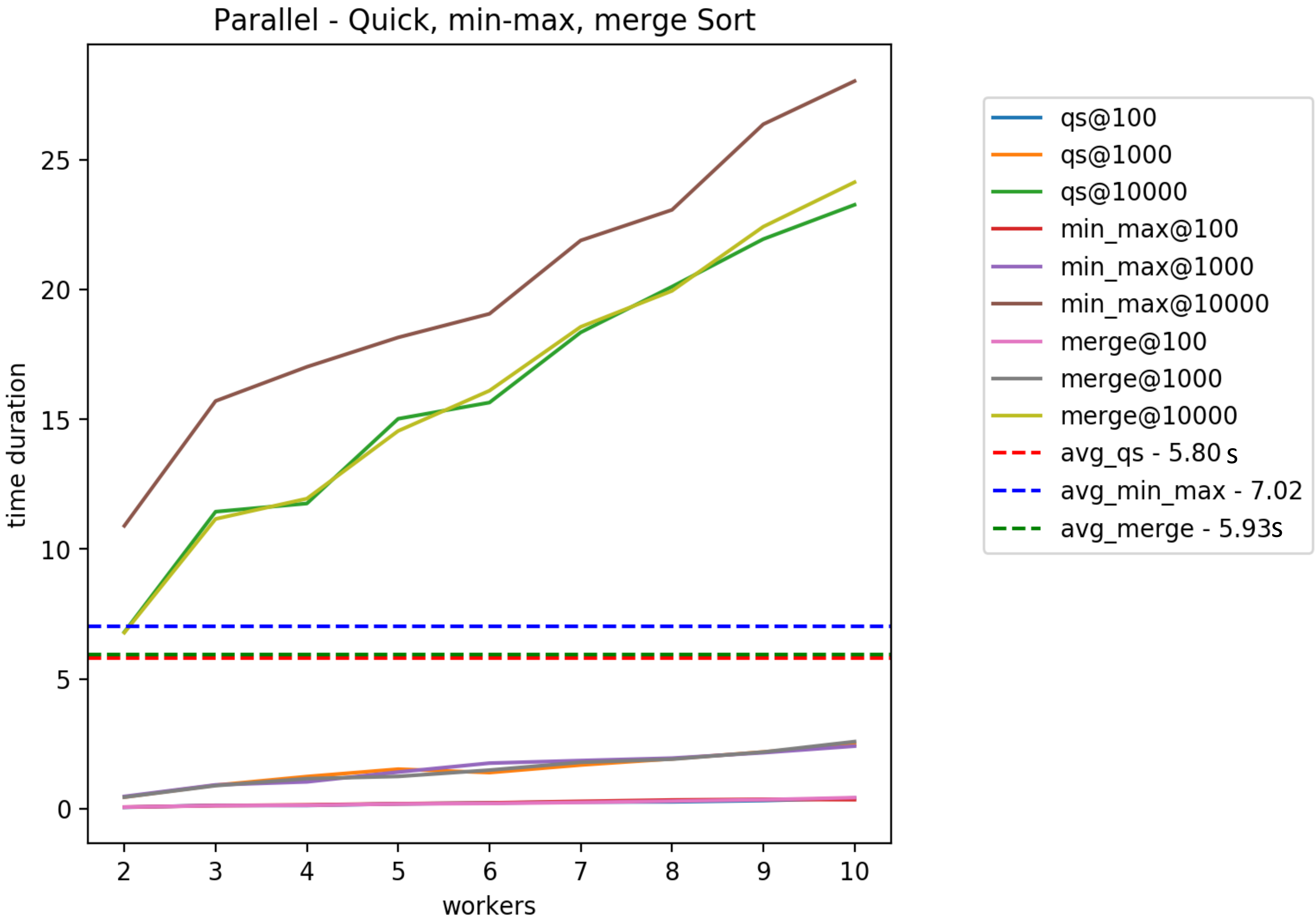

Parallel - QuickSort - avg 5.80s
qs@100
qs@1000
qs@10000
avg
time duration
workers
0
5
10
15
20
2
3
4
5
6
7
8
9
10

Analysis



Parallel merge sort - avg 5.93s
merge@100
merge@1000
merge@10000
avg
time duration
workers
0
5
10
15
20
25
2
3
4
5
6
7
8
9
10

Analysis

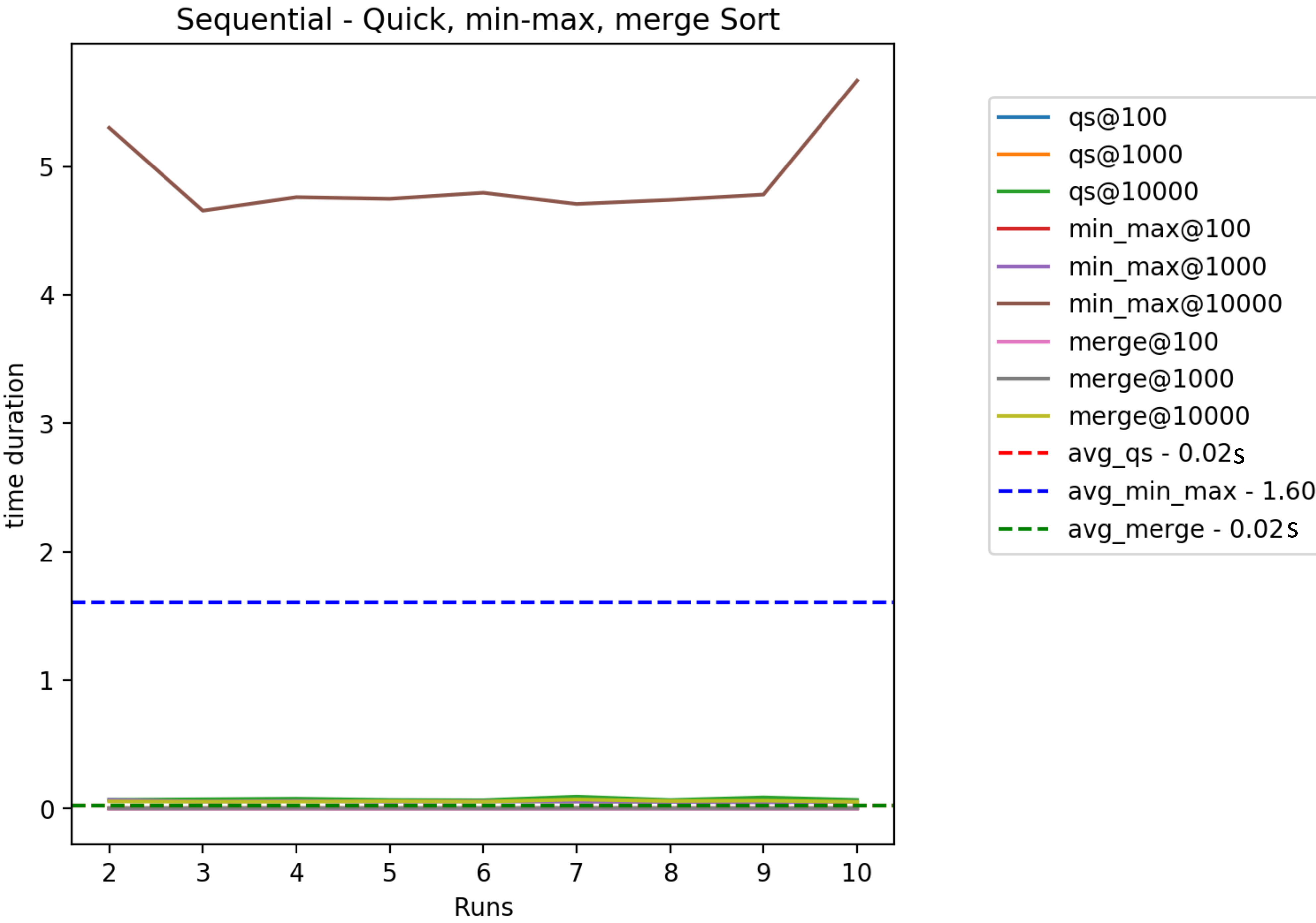

Parallel - Quick, min-max, merge Sort
time duration
workers
0
5
10
15
20
25
2
3
4
5
6
7
8
9
10
qs@100
qs@1000
qs@10000
min_max@100
min_max@1000
min_max@10000
merge@100
merge@1000
merge@10000
avg_qs - 5.80 s
avg_min_max - 7.02
avg_merge - 5.93s

Analysis

Min-max – $O(n^2)$

Merge – O(n * log n)

Quicksort – O(n * log n)



Future Scope

IMPROVE THE TIME COMPLEXITY OF THE SHUFFLE .

LOAD BALANCING

RANDOM SELECTION OF WORKERS RATHER THAN RESOURCE HIERARCHY.

REDUNDANCY OF DATA

CHECK POINTING (LOSS OF SORTED CHUNK, REBUILD THE SORTED CHUNK FROM CHECK POINTED DATA)

SHARED MEMORY QUEUE/ BUFFER FOR TRANSFERRING THE DATA.

Thank You