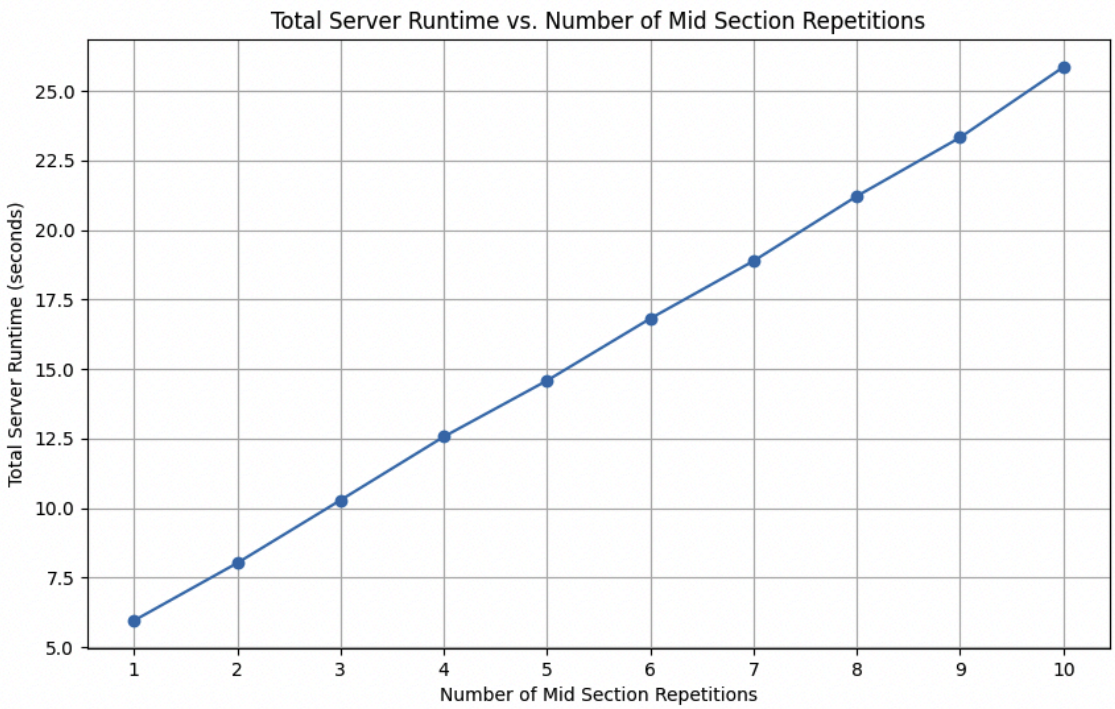
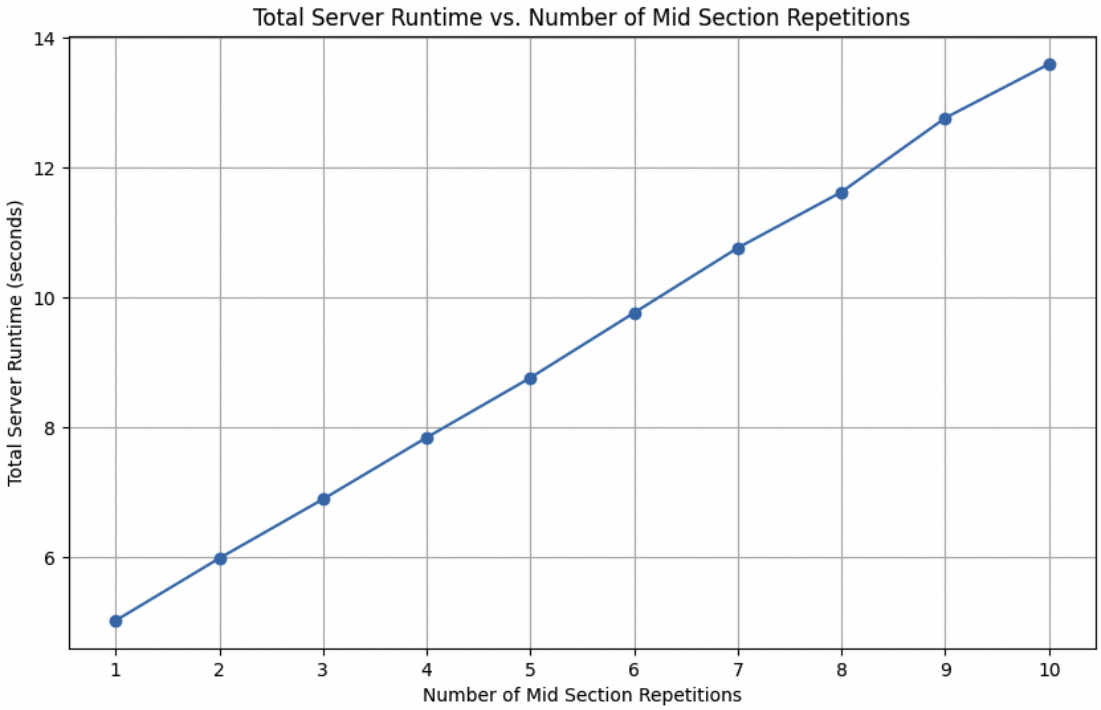
A)

```
Run 1: 5.96 seconds
Run 2: 8.03 seconds
Run 3: 10.29 seconds
Run 4: 12.57 seconds
Run 5: 14.59 seconds
Run 6: 16.82 seconds
Run 7: 18.89 seconds
Run 8: 21.22 seconds
Run 9: 23.33 seconds
Run 10: 25.86 seconds
```



B)

```
Run 1: 5.03 seconds
Run 2: 5.99 seconds
Run 3: 6.9 seconds
Run 4: 7.85 seconds
Run 5: 8.77 seconds
Run 6: 9.77 seconds
Run 7: 10.77 seconds
Run 8: 11.63 seconds
Run 9: 12.77 seconds
Run 10: 13.6 seconds
```

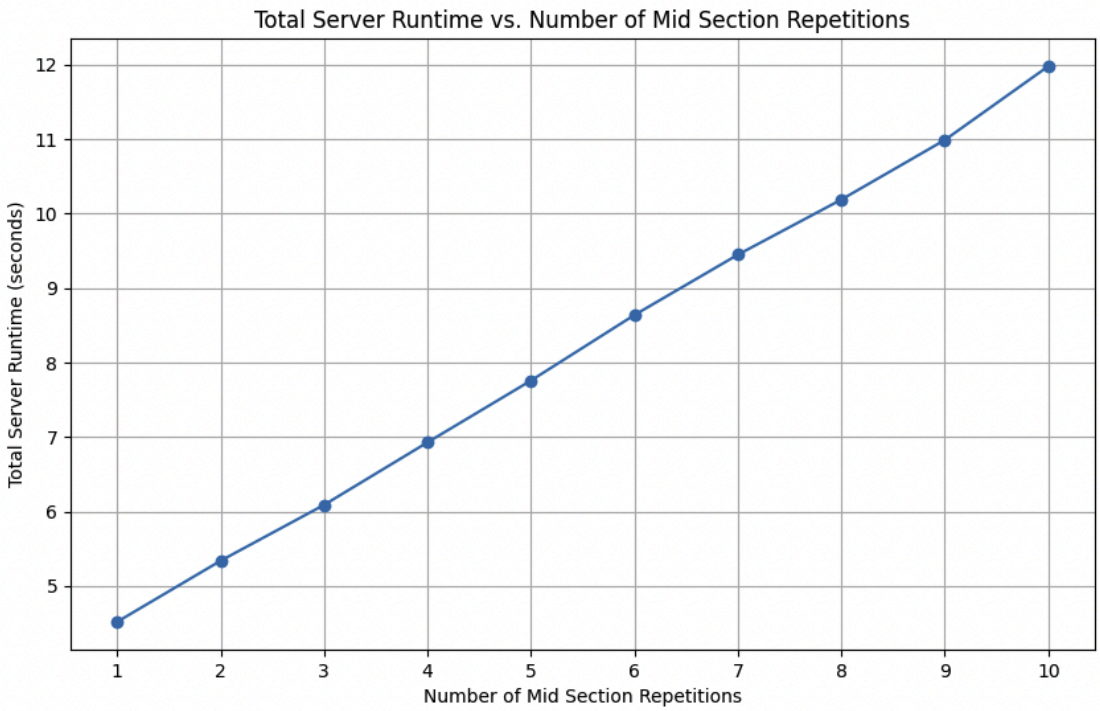


It is pretty much what I had expected. Multithreaded environments perform better than single threaded environments but the difference is most noticeable when the number of requests is large. When the number of requests is lower it performs better than a single threaded approach but not by a large margin, you really see system improvements when the number of requests increase. For example on the 10th run the multithreaded approach performs significantly better with a 12 seconds reduction in time taken to complete.

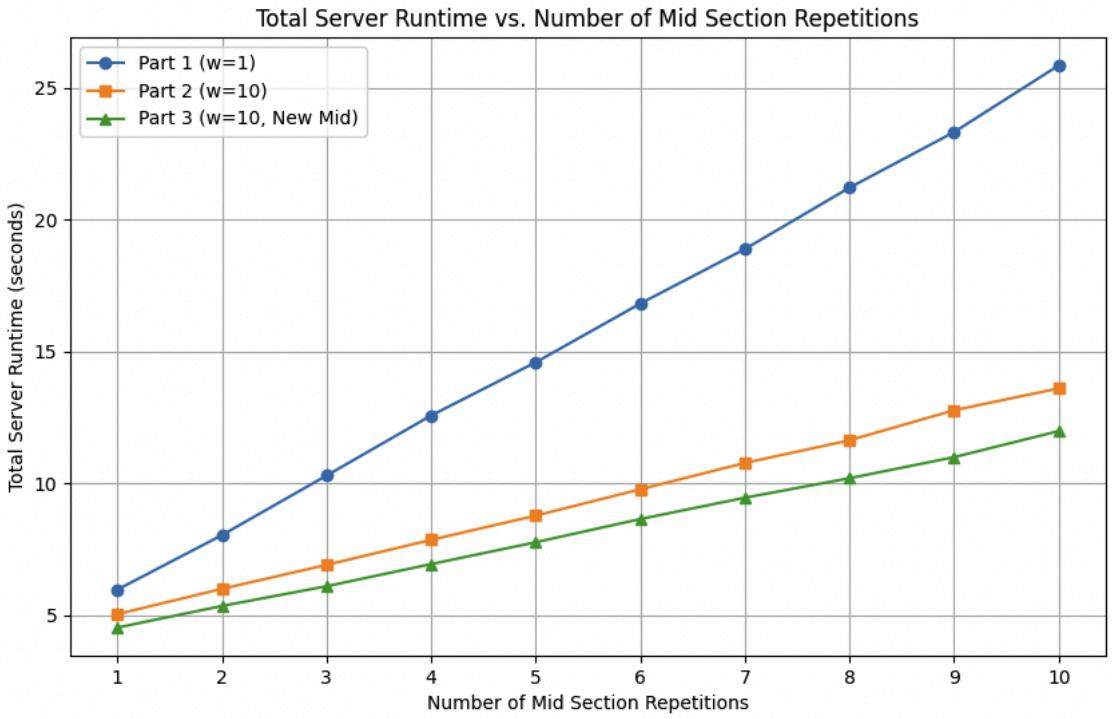
C)

This is just the standalone results for part C.

```
Run 1: 4.52 seconds
Run 2: 5.34 seconds
Run 3: 6.09 seconds
Run 4: 6.93 seconds
Run 5: 7.76 seconds
Run 6: 8.64 seconds
Run 7: 9.45 seconds
Run 8: 10.19 seconds
Run 9: 10.99 seconds
Run 10: 11.98 seconds
```


Total Server Runtime vs. Number of Mid Section Repetitions
Total Server Runtime (seconds)
Number of Mid Section Repetitions
12
11
10
9
8
7
6
5
1
2
3
4
5
6
7
8
9
10

Combined:


Total Server Runtime vs. Number of Mid Section Repetitions
Part 1 (w=1)
Part 2 (w=10)
Part 3 (w=10, New Mid)
Total Server Runtime (seconds)
Number of Mid Section Repetitions
25
20
15
10
5
1
2
3
4
5
6
7
8
9
10

The additional speedup observed in Part 3 compared to Part 2 seems to be due to the different ordering of operations in the Mid section of the client script. Even though the same exact operations are being performed on the same set of images, the way these operations are scheduled and dispatched to the worker threads significantly impacts the overall runtime of the system.

D)

We could change the scheduling policy from **FIFO** to a **Round Robin** approach applied per image.

Scheduler Policy: Round Robin Scheduling per Image

This should ideally improve performance by allowing multiple threads to work on the same operation.

- **Maximize Parallelism:** Allow multiple worker threads to process operations concurrently by interleaving requests on different images.
- **Maintain Correctness:** Ensure that operations on the same image are executed in the order they were received.

Implementation Steps:

1. **Maintain Separate Queues for Each Image:**
    - Create a separate queue for each image ID to hold the requests targeting that image.
    - This can be implemented as an array or a map of queues, indexed by image ID.
2. **Implement Round Robin Scheduling Across Image Queues:**
    - Maintain a list or circular queue of image IDs with pending requests.
    - The scheduler iterates over the list of image queues in a cyclic manner, dequeuing one request from each image queue in turn.
    - If an image queue is empty, skip it and move to the next.
3. **Worker Threads Pull from a Shared Execution Queue:**
    - The requests selected by the scheduler are placed into a shared execution queue.
    - Worker threads pull requests from this execution queue to process them.
4. **Ensure Per-Image Operation Order:**
    - Within each image queue, requests are enqueued and dequeued in FIFO order.
    - This maintains the correct sequence of operations on each image.