PROBLEM STATEMENT

- K-mer counting is one of the most widely used algorithm in genome sequencing.
- The goal is to find frequency of k-mers in a given sequence, such as DNA.
- A k-mer is a substring of the sequence of length k.
- For example, AGTTACCGTA
  - 3-mers: AGT, GTT, TTA, TAC, ACC, CCG, CGT, GTA
  - 4-mers: AGTT, GTTA, TTAC, TACC, ACCG, CCGT, CGTA

PROBLEM STATEMENT

- To compute such k-mers for large genome sequences (in the order of GBs), the computation becomes difficult and requires greater computational resources.
- Our aim was to conduct such computation using limited resources by using appropriate algorithms and optimization techniques, like multi-threading and exploiting cache locality.
- We have used techniques like one-one mapping, hashing, bloom filters, and sorting to solve this problem efficiently.

COMPUTE CONFIGURATION

- The tests have been run on a MacBook Air M1 (Late 2020) model, consisting of an 8-core CPU, and 7-core GPU.
- Observation: Running ICC compiler on this computer did not help with the performance. We believe that this is because ICC is not natively supported for the M1 chip, and is emulated through Rosetta 2.
- All further tests were run using the GNU GCC compiler.

COMPUTE CONFIGURATION

- The tests were mostly conducted on 200 MB genome sequences.
- The test for Bloom Filters was conducted on a 2 MB genome sequence to save time. However, the output from the bloom filter which was used for the sorting algorithm were received from running the bloom filter on a 200 MB genome sequence.

PERFORMANCE EVALUATION FRAMEWORK

- The computation time (in seconds) is used as the performance evaluation framework.
- We have compared results against unoptimized (sequential, no compiler optimization) baseline programs.

DATASETS USED

- We have used the FASTA dataset provided.
- We have also made a random sequence generator that provides us with a random sequence of the required size.

APPROACH

- We have 2 different approaches based on the length of the k-mer.
  - For shorter k-mers $(2 \leq k \leq 10)$, we have used direct mapping to map each sequence to a unique address, and store its frequency. This could not be done for $k > 10$ as this method requires significant amount of memory space.
  - For large k-mers $k > 10$, we have implemented a bloom filter which filters the sequences of higher frequency, followed by sorting to find the frequency of each filtered k-mer.

APPROACH: DIRECT MAPPING ($2 \leq k \leq 10$)

- Just like the binary and ternary number systems, we devised a number system with base 4 and associated a value with each of the 4 bases, to get the value of a permutation of these 4 bases.

- Example:

  - If $A = 0,\ C = 1,\ G = 2,\ T = 3$, then the value represented by GATC would be $4^3 * 2 + 4^2 * 0 + 4^1 * 3 + 4^0 * 1 = 141$

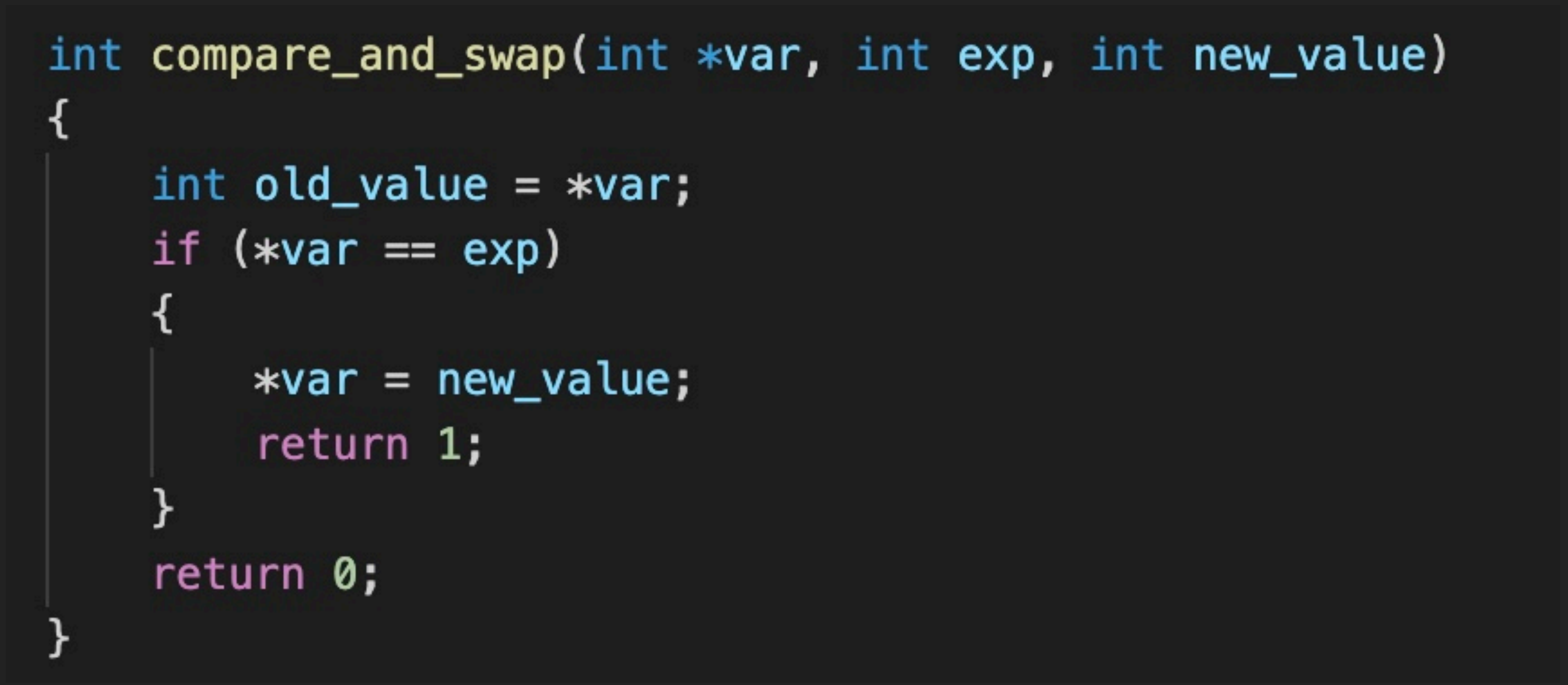
APPROACH: DIRECT MAPPING $(2 \leq k \leq 10)$

- We made a mapping array (hash) of size 4^k such that it could contain every k-length k-mer.
- From the previous example, the index corresponding to GATC would be 141. The index corresponding to AAAA would be 0, and the index corresponding to TTTT would be $4^k - 1$.
- Initially, every value in the hashmap is zero, and every time that we encounter a k-mer, we increment the value corresponding to that index by 1.

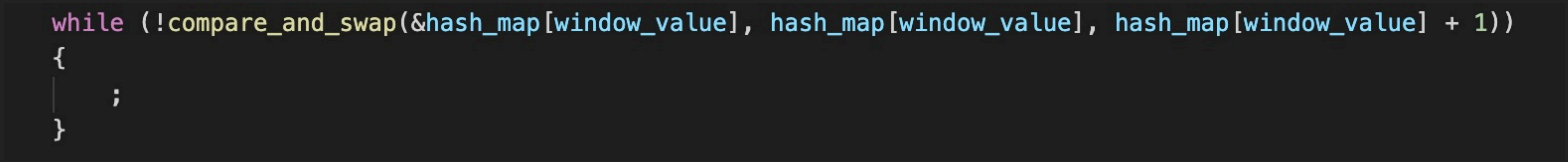
APPROACH: DIRECT MAPPING $(2 \leq k \leq 10)$

- To optimize this method, we have used the sliding-window technique, parallelization, and compiler optimization.
- Concurrency could have been dealt with mutex locks, but that would have significantly reduced the performance.
- To overcome this, we used compare-and-swap method over the hash table.

APPROACH: DIRECT MAPPING $(2 \leq k \leq 10)$



```c
int compare_and_swap(int *var, int exp, int new_value)
{
    int old_value = *var;
    if (*var == exp)
    {
        *var = new_value;
        return 1;
    }
    return 0;
}
```

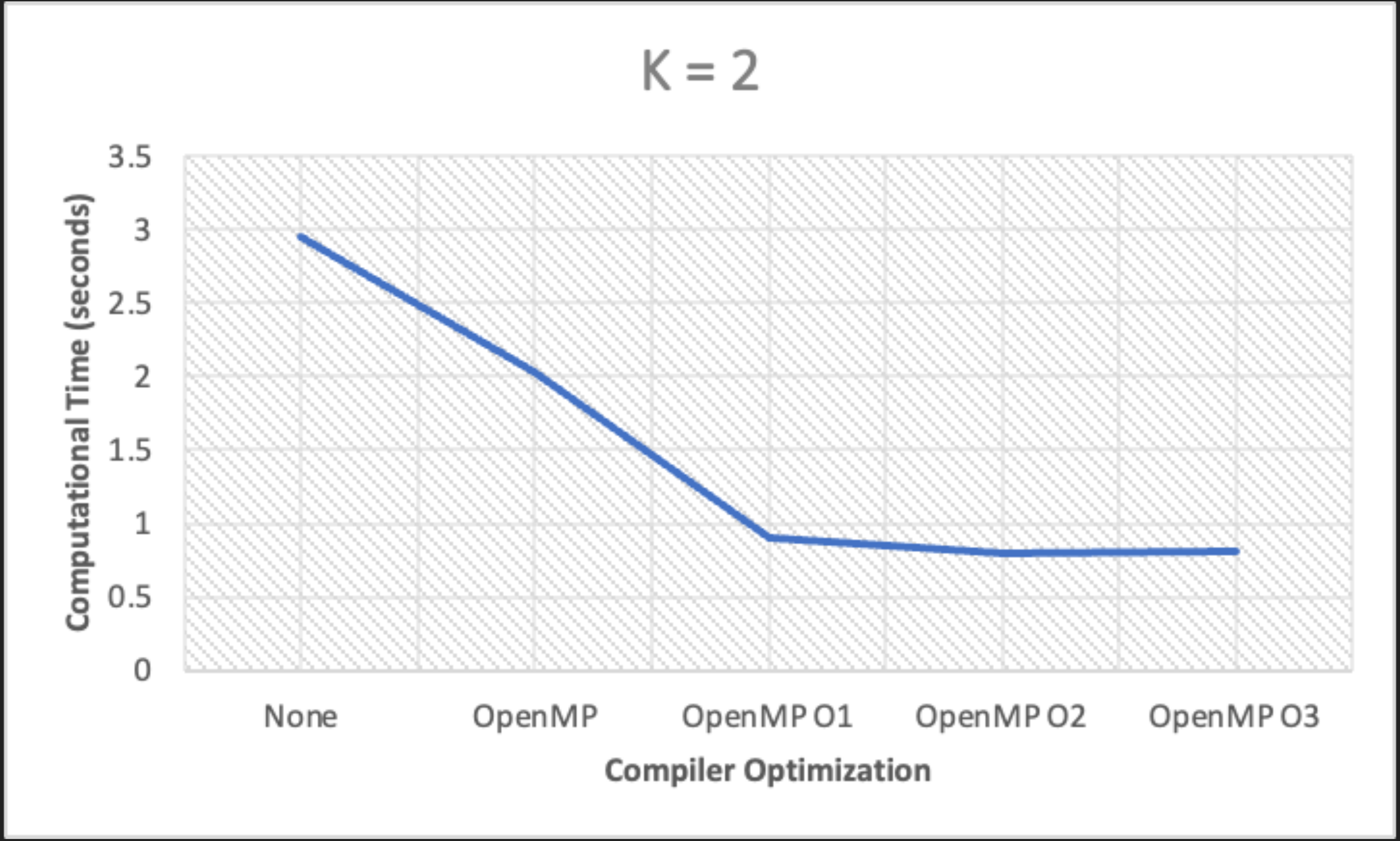


```c
while (!compare_and_swap(&hash_map[window_value], hash_map[window_value], hash_map[window_value] + 1))
{
    ;
}
```

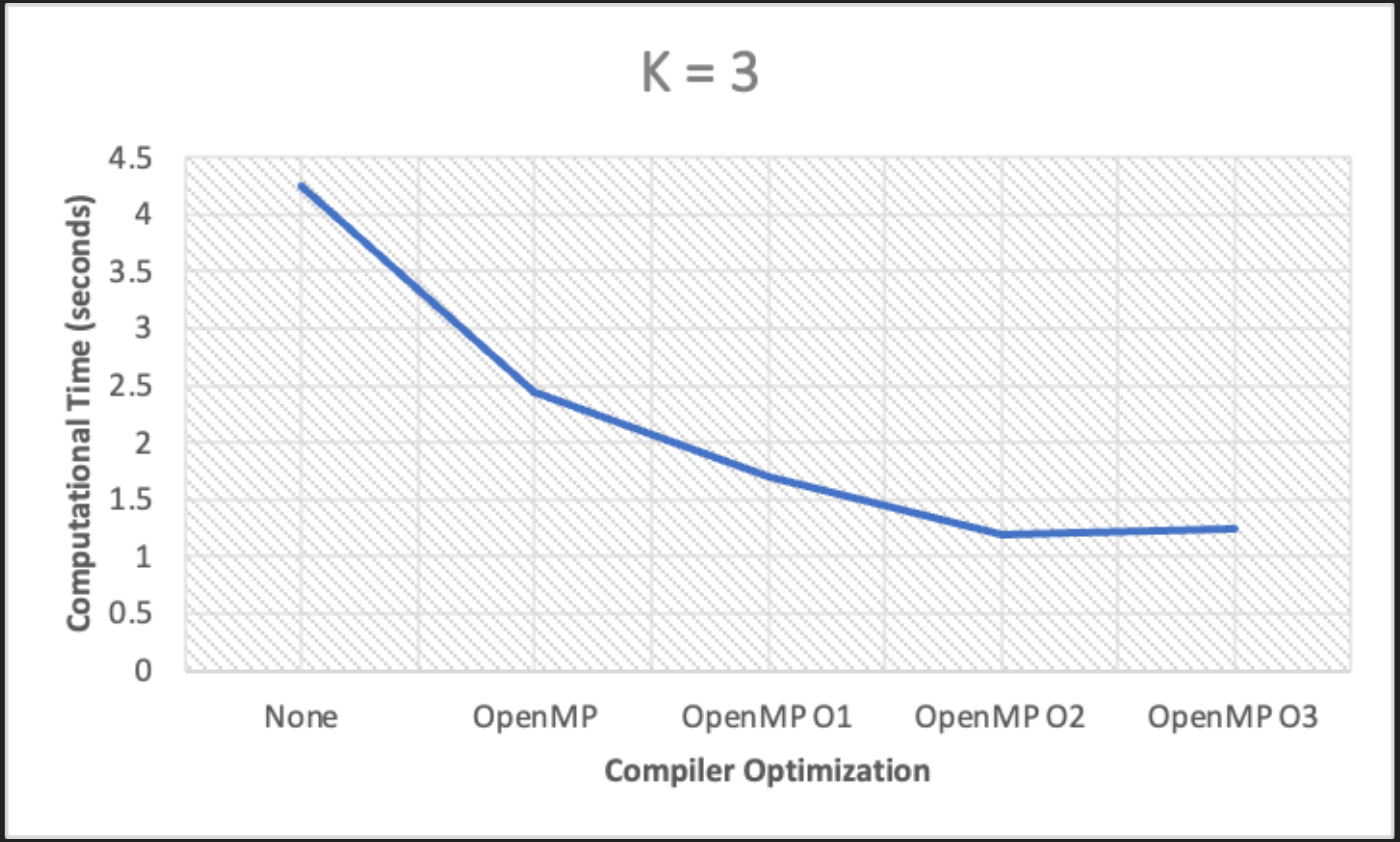
DIRECT MAPPING: OPTIMIZATION

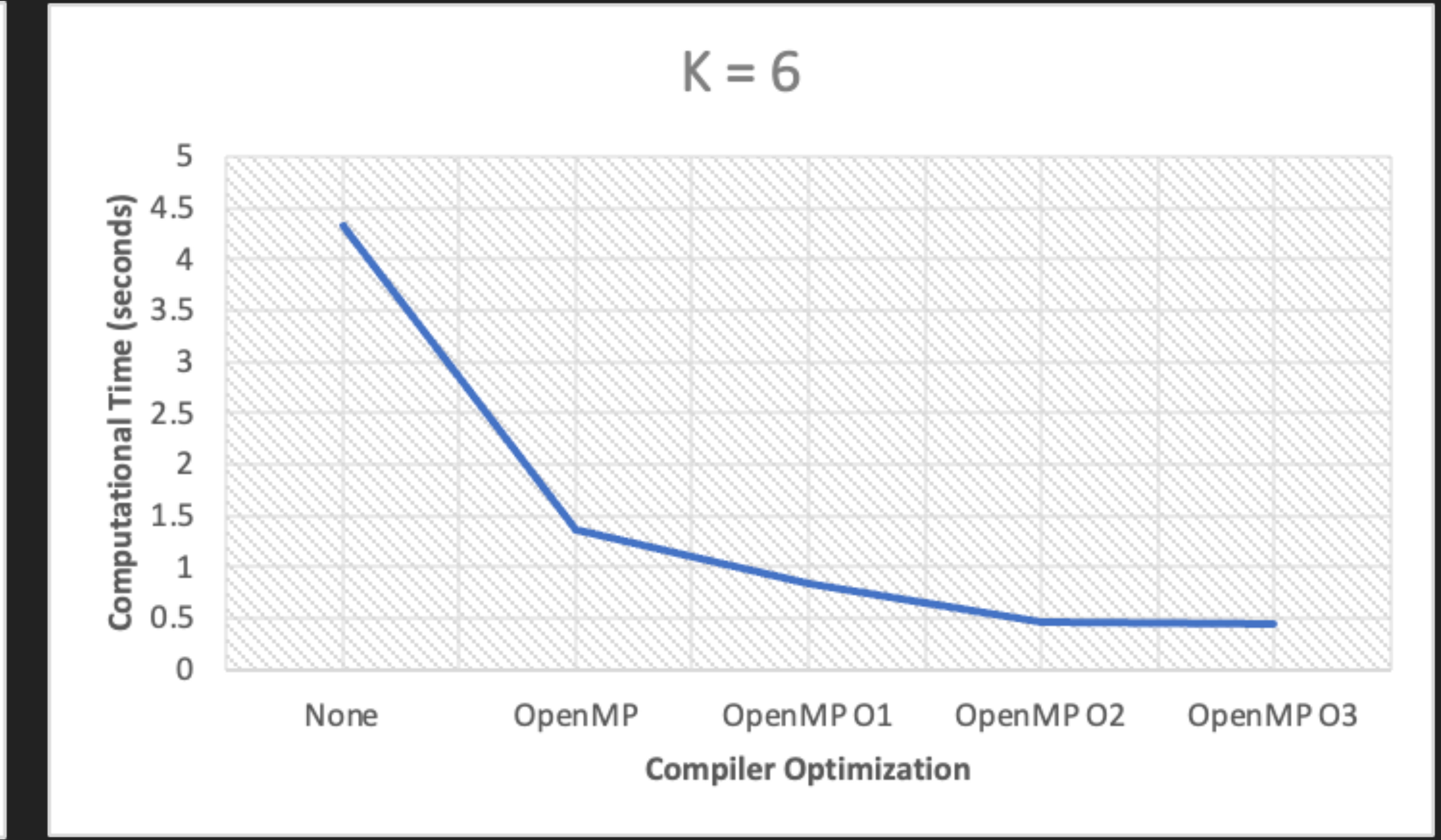
- While processing the input DNA sequence, we use multiple threads to start reading and processing the sequence from multiple start points; such that none of the threads overlap and the sequence is covered in entirety.
- This helps in reducing the time required to read and process each place individually. We now read and process multiple k-mers at the same time.
- Sliding-window

PERFORMANCE ANALYSIS



PERFORMANCE ANALYSIS





PERFORMANCE ANALYSIS (G++)

K	None	OpenMP	OpenMP O1	OpenMP O2	OpenMP O3
1	3.0136144	1.6031286	0.7619248	0.3217892	0.3089368
2	2.9552178	2.0314664	0.9050326	0.7987276	0.811868
3	4.254655	2.4373428	1.7035296	1.19369	1.2521258
4	4.3260098	1.5571162	1.2837936	1.013441	0.9925132
5	4.3297462	1.2915544	0.959752	0.6070784	0.6018942
6	4.3348744	1.355129	0.8357354	0.4548026	0.4444926
7	4.3521634	1.347194	0.7986878	0.4143256	0.4155908
8	4.4443794	1.4621214	0.8901894	0.5091254	0.5079214
9	4.7935638	1.7763682	1.2142638	0.843637	0.8406
10	6.2440276	3.1868876	2.6176774	2.208734	2.246716

PERFORMANCE ANALYSIS (ICC)

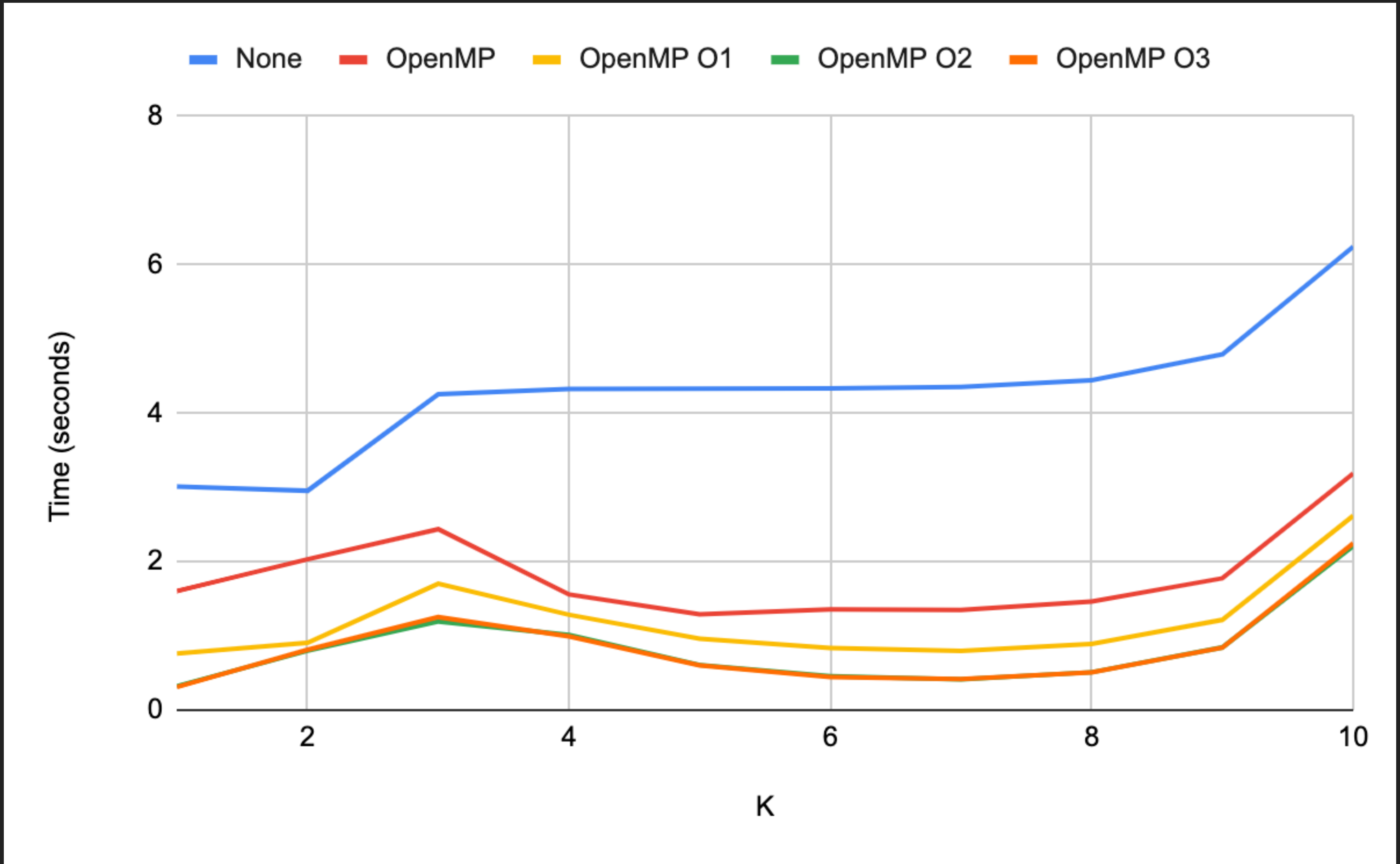
K	None	OpenMP	OpenMP O1	OpenMP O2	OpenMP O3
1	2.2869374	1.8175736	4.9909132	1.8002624	1.7845466
2	2.2664718	2.8708668	4.5373788	2.9187022	2.8597712
3	3.125434	3.4872878	3.6987776	3.5074438	3.4917146
4	3.3506528	2.1196114	2.2406018	2.2252836	2.267931
5	3.3401748	1.3666512	1.700832	1.4265398	1.449304
6	3.3377198	1.1453556	1.3477028	1.1702606	1.2286354
7	3.3635756	1.1769414	1.3692594	1.171799	1.3073752
8	3.4927536	1.3572262	1.5524804	1.3489832	1.4868972
9	4.0630384	1.8388154	2.0839738	1.856455	2.037074
10	6.2550564	3.9300448	4.187388	3.9790542	4.4873388

PERFORMANCE ANALYSIS (G++ VS ICC)

- Shown as G++ values minus ICC values. Negative values denote G++ was faster.

K	None	OpenMP	OpenMP O1	OpenMP O2	OpenMP O3
1	0.726677	-0.214445	-4.2289884	-1.4784732	-1.4756098
2	0.688746	-0.8394004	-3.6323462	-2.1199746	-2.0479032
3	1.129221	-1.049945	-1.995248	-2.3137538	-2.2395888
4	0.975357	-0.5624952	-0.9568082	-1.2118426	-1.2754178
5	0.9895714	-0.0750968	-0.74108	-0.8194614	-0.8474098
6	0.9971546	0.2097734	-0.5119674	-0.715458	-0.7841428
7	0.9885878	0.1702526	-0.5705716	-0.7574734	-0.8917844
8	0.9516258	0.1048952	-0.662291	-0.8398578	-0.9789758
9	0.7305254	-0.0624472	-0.86971	-1.012818	-1.196474
10	-0.0110288	-0.7431572	-1.5697106	-1.7703202	-2.2406228

PERFORMANCE ANALYSIS



APPROACH: BLOOM FILTER ($k > 10$)

- For larger ks, approach 1 is impractical as the need for main memory increases exponentially. Thus, for larger cases, we require another method.
- We have used sorting to find frequently occurring k-mers. However, this method involves approximation which is a tradeoff for computational resources.
- We also take the help of bloom filters in our approach.

APPROACH: BLOOM FILTER ($k > 10$)

- We have divided this approach into 2 steps:
  - Filtering the k-mers through the bloom filter.
  - Sorting the k-mers based on the frequency.

BLOOM FILTER

- A bloom filter is a space efficient probabilistic data structure which requires multiple hash functions.
- The hash function used in bloom filters should be independent and uniformly distributed, and should be computationally fast.
- For our approach, we have used 4 different hash functions that make the use of various prime numbers to map every k-mer to an identifier index.
- We have used a bloom array of size 1299709 which initially consists of zeroes. Whenever we encounter a k-mer, we mark the value at the identifier to be True.

BLOOM FILTER

- If the values corresponding to all hash functions are already true, then we output the k-mer in a file.
- This approach gives true positives as well as false positives. Hence, the probabilistic nature.
- We might encounter the same k-mer again. In such a case, we output the k-mer again, ignoring the order.
- We now sort these outputted k-mers. The sorting method is discussed later.

BLOOM FILTER: OPTIMIZATION

- We use the same technique employed in approach 1 to read and process the string from multiple starting points.
- Sliding-window technique can be used to calculate the hash value for k-mers. Instead of calculating the hash value individually for each k-mer, we can use the hash value obtained for the previous k-mer to obtain the hash value of the current k-mer.
- The powers of prime numbers can be calculated separately and stored in an array to avoid repetitive calculations.

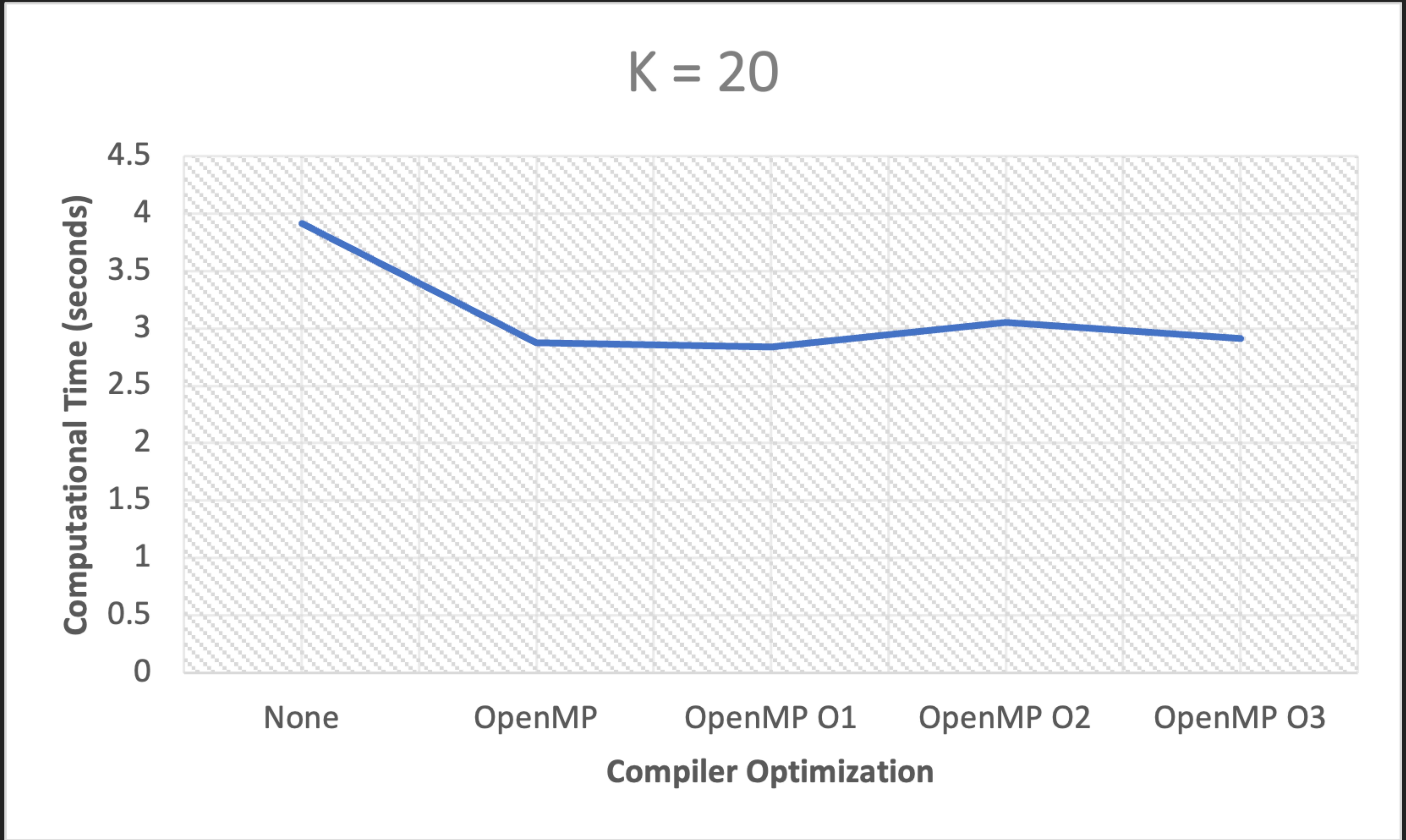
BLOOM FILTER: OPTIMIZATION

- Observation
  - The parallelization techniques employed did not create any significant difference. We suspected that this was because we were outputting the k-mers into a single output file.
  - Thus, each thread waited for other threads to finish their modification.
- Solution?
  - Instead of a single output file, we employed multiple output files; one corresponding to each thread. This improved the parallelization performance significantly.

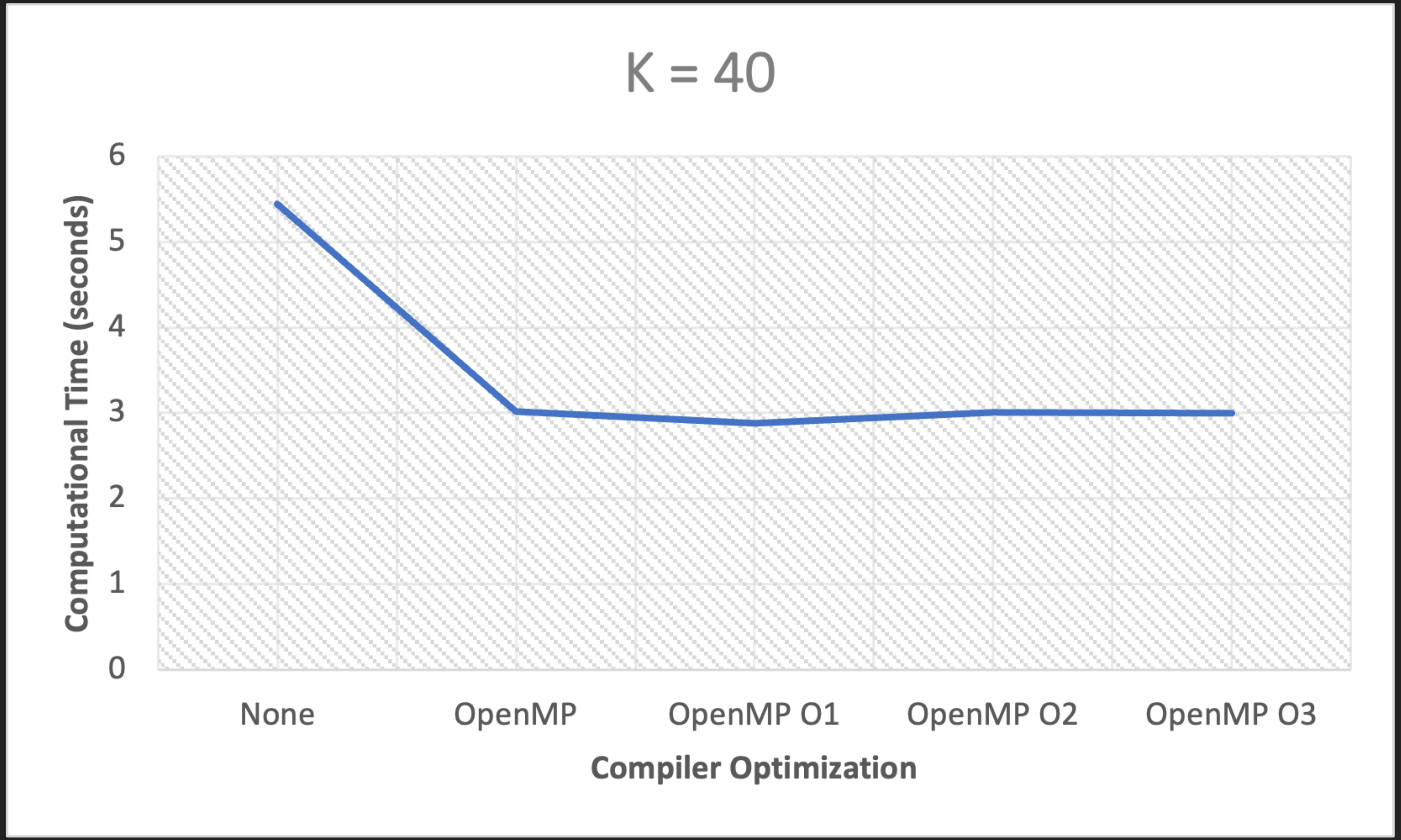
BLOOM FILTER: OPTIMIZATION

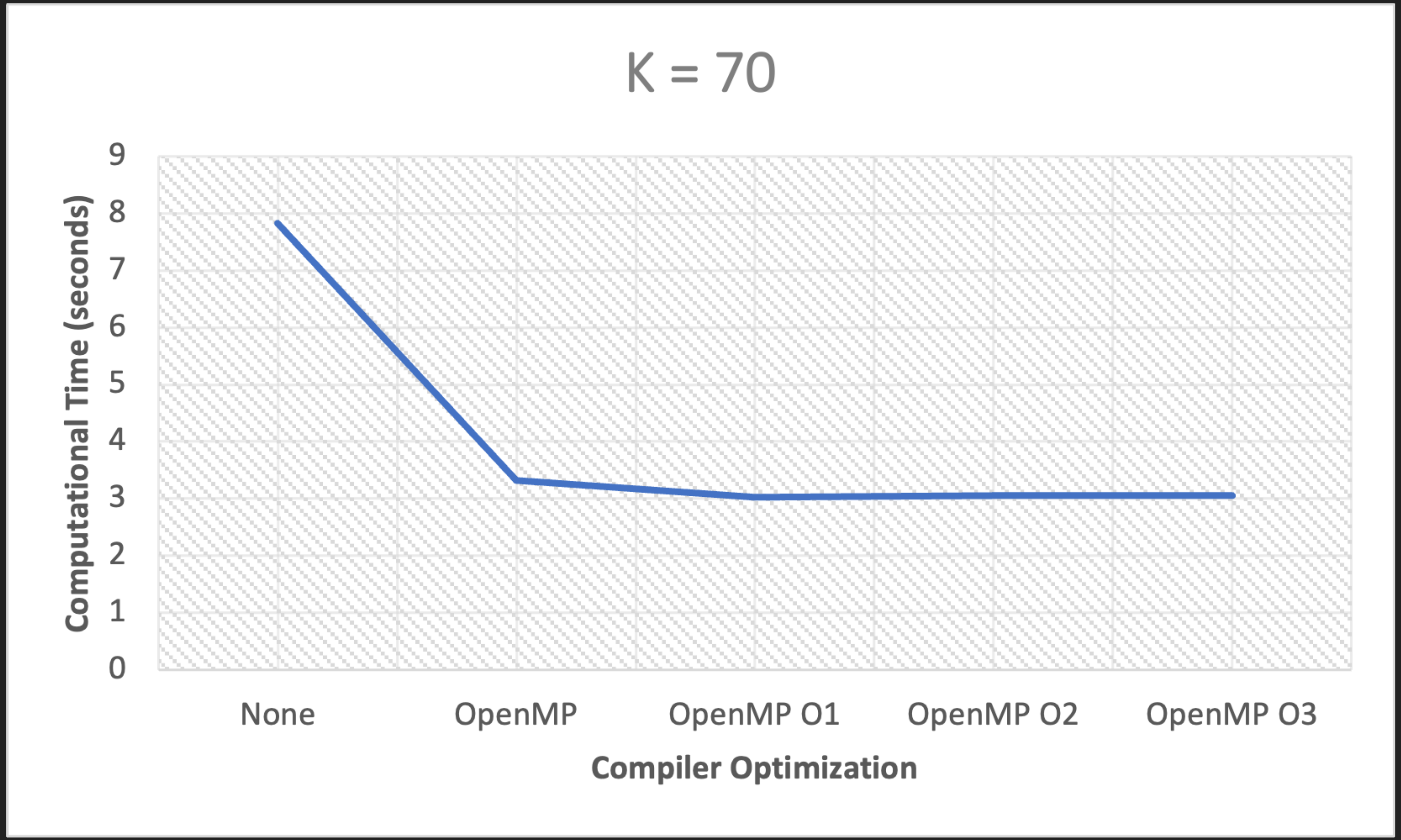
- Since we were outputting individual k-mers into the output files, we believe that these multiple write instructions were the bottleneck to our program. Ergo, we tried to reduce the number of write operations.
- This was done by storing the required k-mers into a single string, and then writing the entire string in a single operation (until the string reaches a particular length).
- Observation
  - Instead of improving the performance, this method decreased the performance by about 10 times.

PERFORMANCE ANALYSIS



PERFORMANCE ANALYSIS

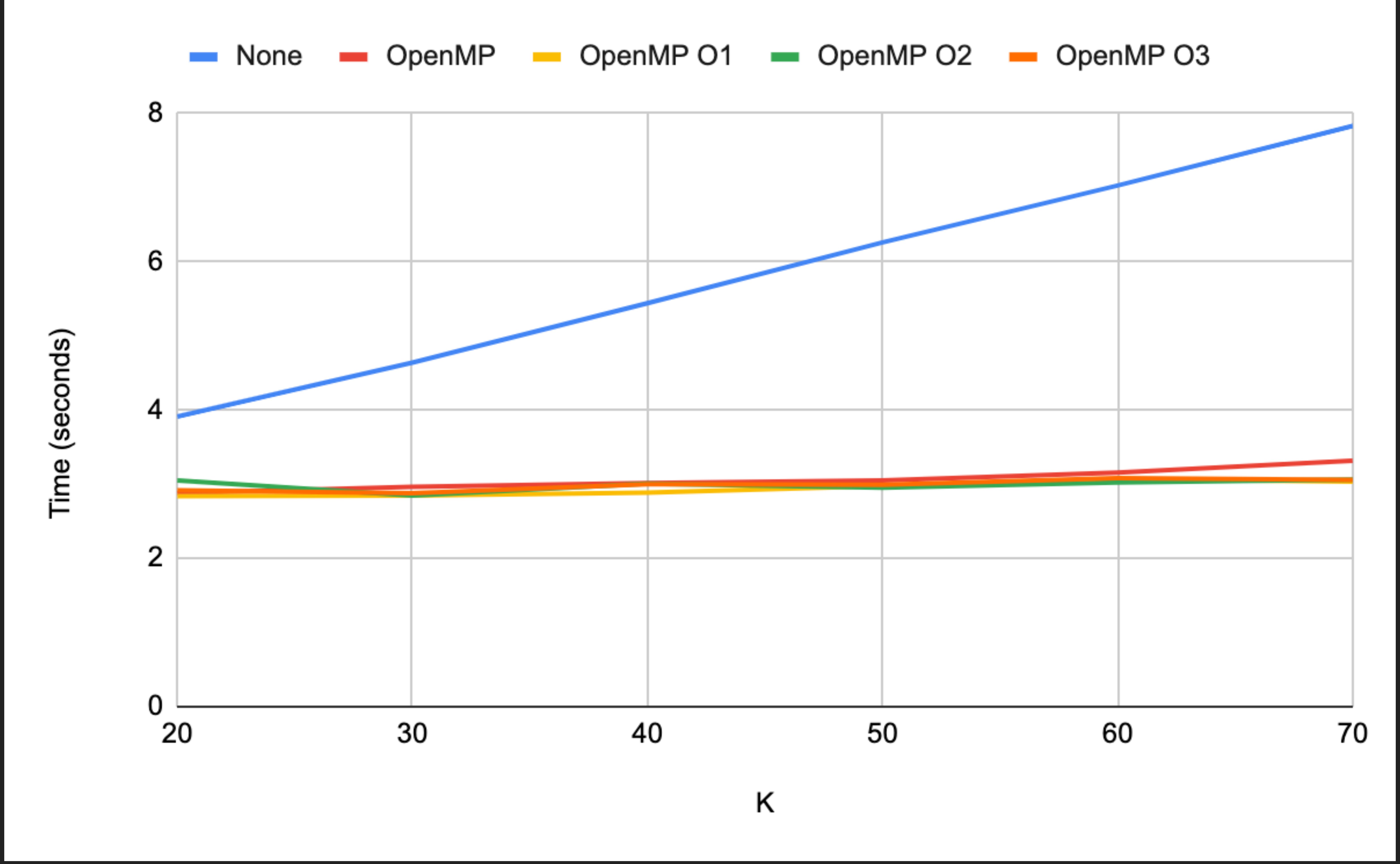




PERFORMANCE ANALYSIS

K	None	OpenMP	OpenMP O1	OpenMP O2	OpenMP O3
20	3.9128646	2.8785384	2.840756	3.0518072	2.9163722
30	4.638607	2.9648424	2.8456088	2.8482374	2.8802214
40	5.442579	3.0163326	2.8862044	3.009182	3.0015894
50	6.2609636	3.0508238	2.9754982	2.9533024	2.9905368
60	7.0283918	3.1577876	3.072164	3.0218262	3.0840088
70	7.8324288	3.3155462	3.0332978	3.060919	3.0601522

PERFORMANCE ANALYSIS



SORTING

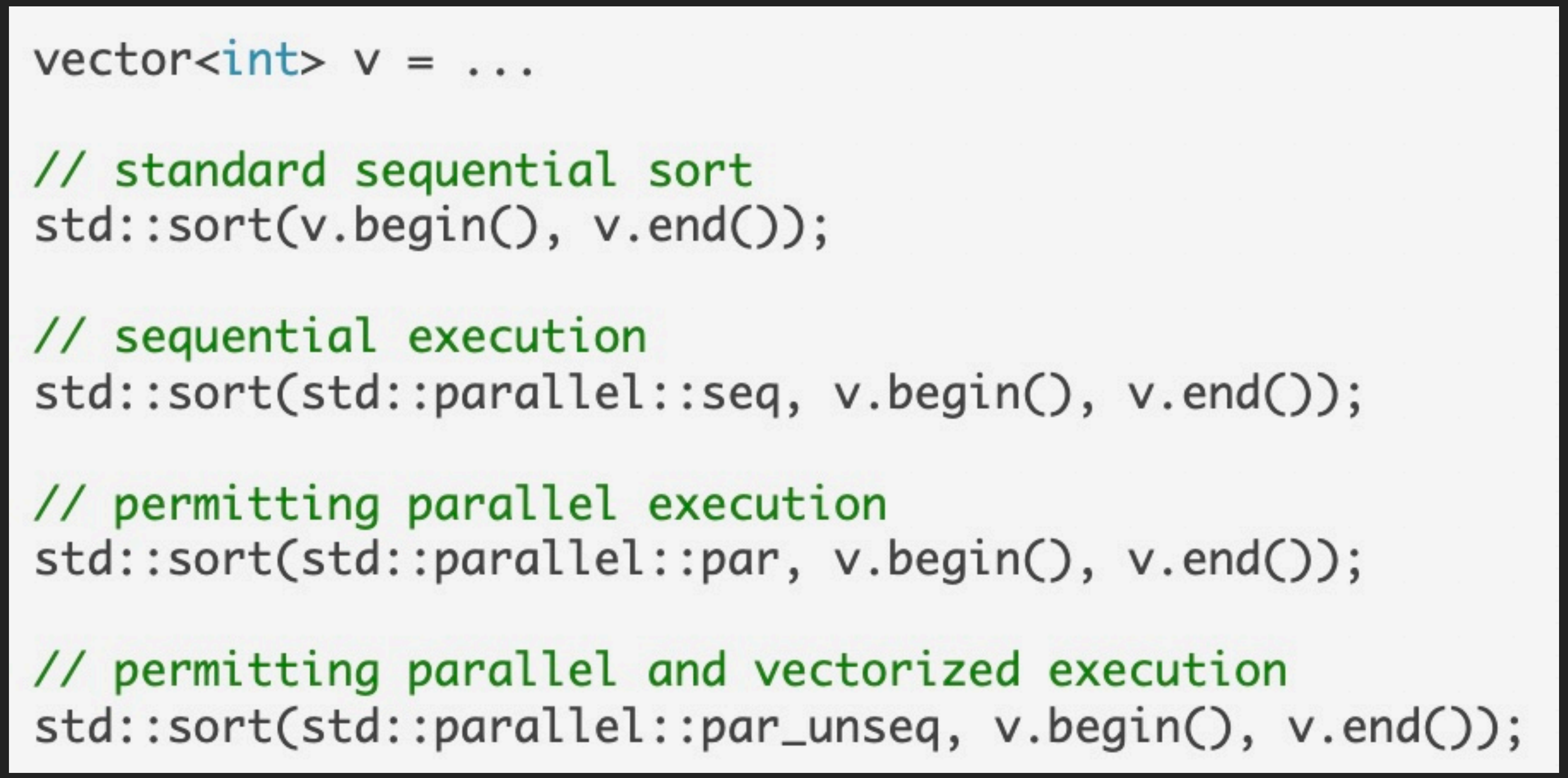
- Once we get the filtered k-mers from the bloom filter in the raw form, we need to cluster the same entries and obtain their frequency. Based on this frequency, we form our results.
- The first step to sorting includes inputting the k-mers from the output files of the bloom filter.
- This can be parallelized using multiple threads. For our approach, due to the limited memory space, we have used a single input file.

SORTING

- Since we are dealing with a huge amount of k-mers, assume we input n k-mers in a single time. We sort these n k-mers such that the same k-mers are now stored together.
- We then traverse over these n entries and count the frequency of each k-mer. Out of these n k-mers, we then store the m most frequently occurring k-mers in our final result.
- In our approach n was 10^4 and m was 1.

SORTING: OPTIMIZATION

- We can employ multiple threads to read the input from multiple input files.
- Even if we input a single file, we can use multiple threads to select multiple collections of n k-mers, and process them.
- For sorting, we can use standard techniques like parallel sorting.

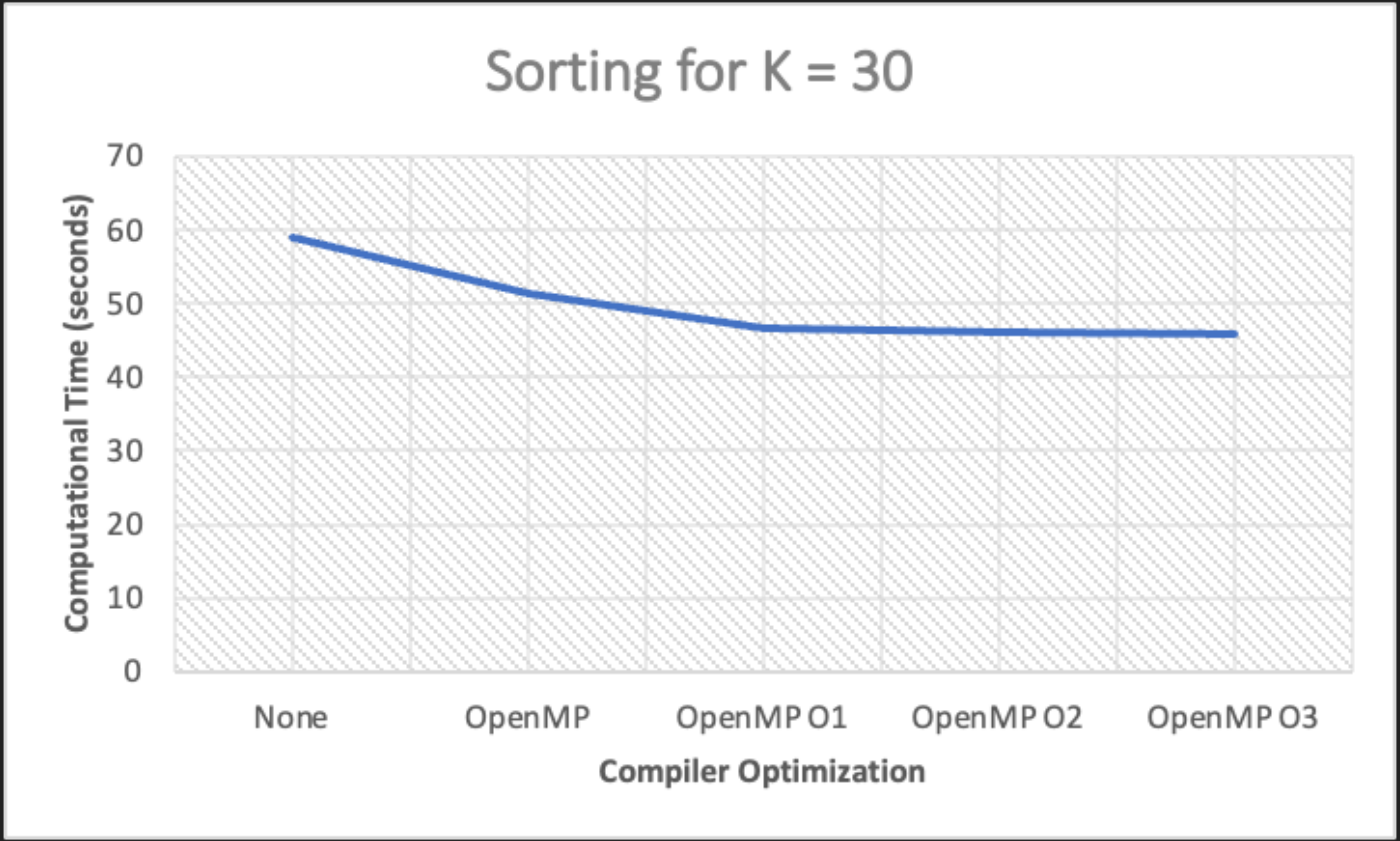


```
vector<int> v = ...

// standard sequential sort
std::sort(v.begin(), v.end());

// sequential execution
std::sort(std::parallel::seq, v.begin(), v.end());

// permitting parallel execution
std::sort(std::parallel::par, v.begin(), v.end());

// permitting parallel and vectorized execution
std::sort(std::parallel::par_unseq, v.begin(), v.end());
```

PERFORMANCE ANALYSIS

K	None	OpenMP	OpenMP O1	OpenMP O2	OpenMP O3
30	59.0877	51.4870	46.6232	46.0814	45.7560

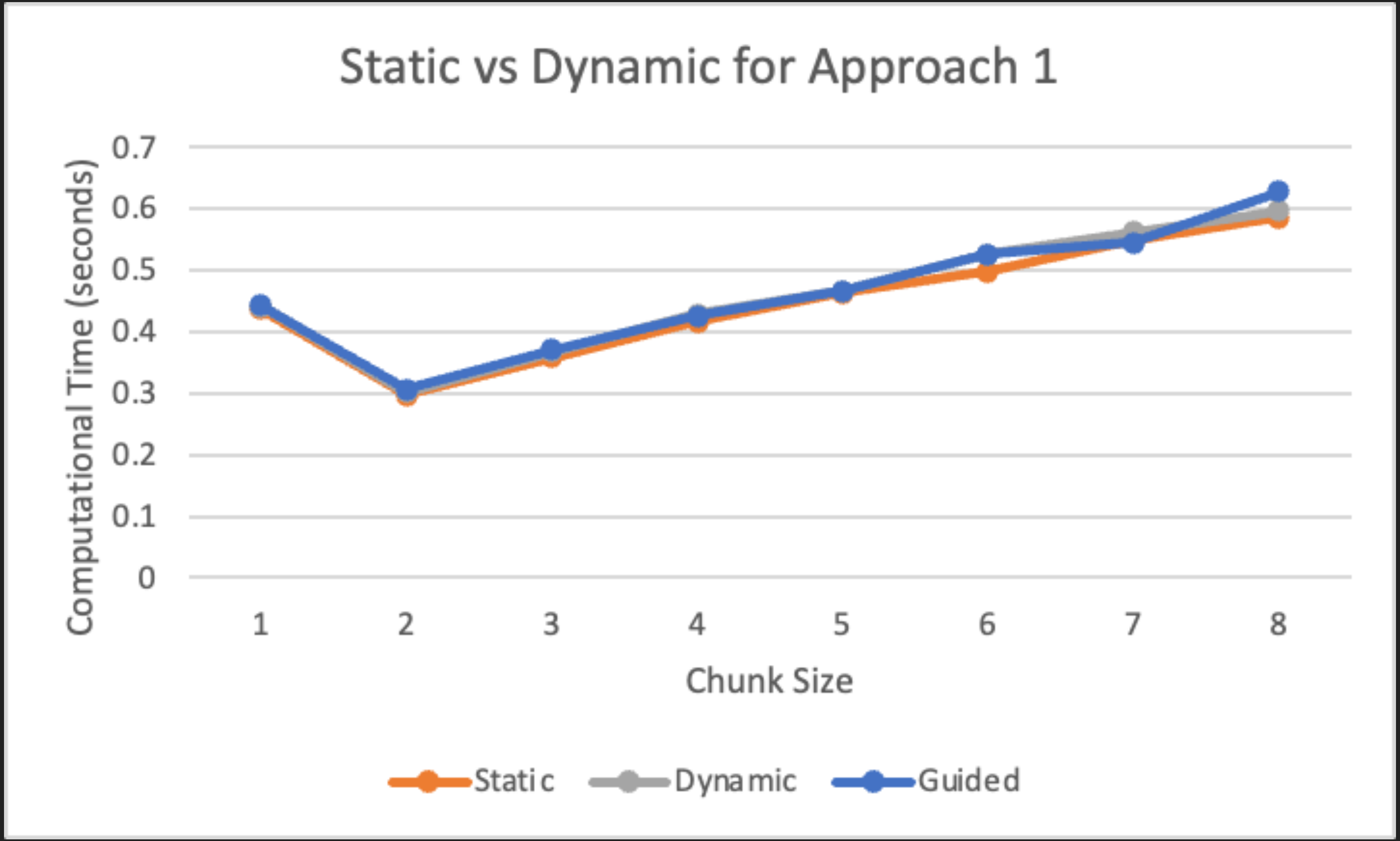
PERFORMANCE ANALYSIS



OPTIMIZATION BASED ON SCHEDULING: APPROACH 1

Chunk Size	Static	Dynamic	Guided
1	0.435813	0.440897	0.443125
2	0.296768	0.301142	0.306895
3	0.35775	0.366726	0.369914
4	0.415806	0.427995	0.42479
5	0.463553	0.466389	0.466866
6	0.497406	0.526266	0.526438
7	0.547731	0.561857	0.544143
8	0.584807	0.596234	0.628296

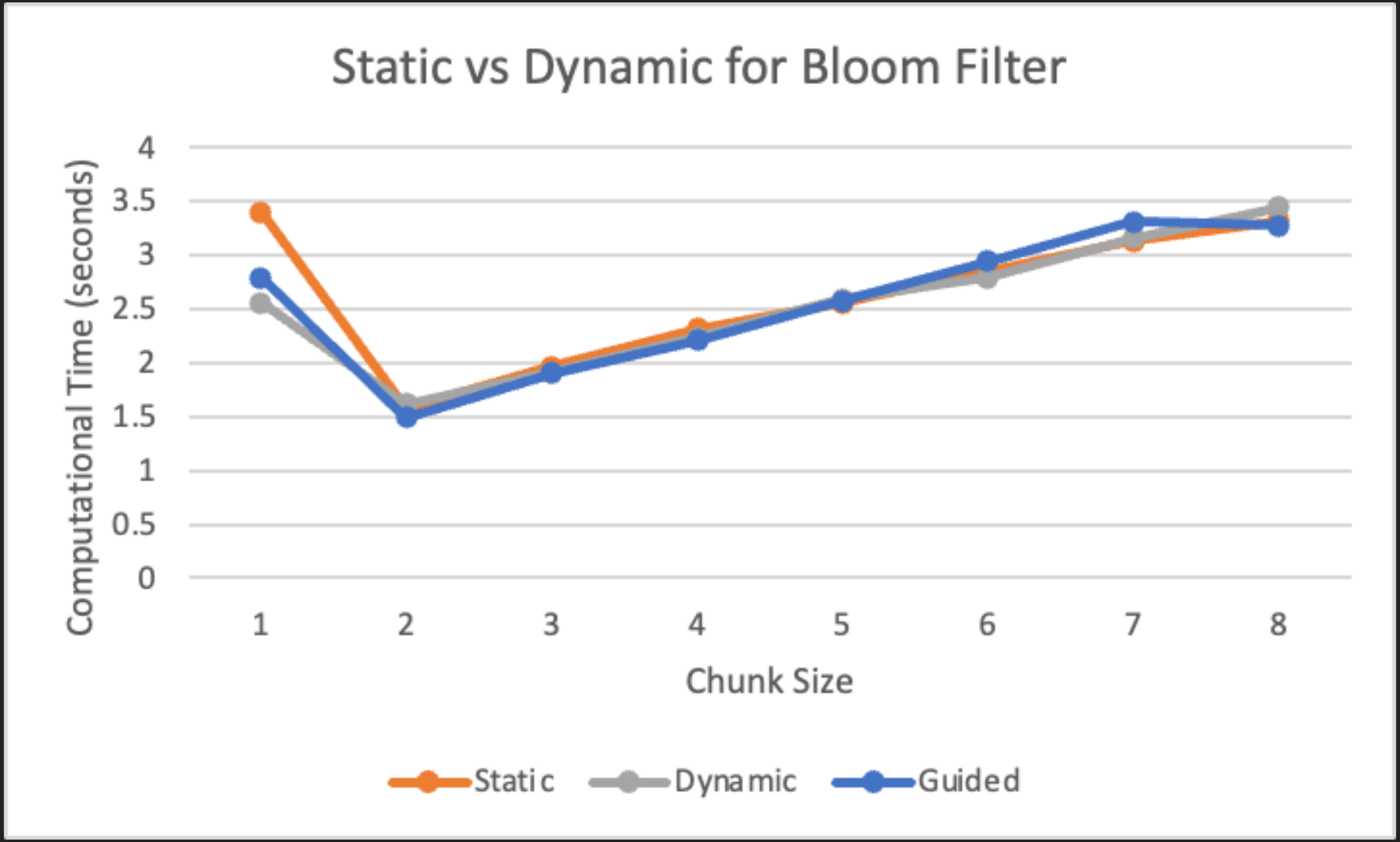
OPTIMIZATION BASED ON SCHEDULING: APPROACH 1



OPTIMIZATION BASED ON SCHEDULING: BLOOM FILTER

Chunk Size	Static	Dynamic	Guided
1	3.398665	2.555918	2.7927
2	1.583289	1.61374	1.488257
3	1.966618	1.91394	1.902521
4	2.314211	2.233896	2.203833
5	2.555563	2.591191	2.571372
6	2.845544	2.789991	2.942086
7	3.130327	3.159472	3.304811
8	3.318271	3.438365	3.270544

OPTIMIZATION BASED ON SCHEDULING: BLOOM FILTER



OBTAINED SPEED-UPS

Approach	Worst time	Best time	Speedup
One-one mapping	4.33	0.3	14.4333333
Bloom filter	4.64	1.49	3.11409396
Sorting	59.09	45.76	1.29130245

THANK YOU!