**Name: AJAL RC**

**Student ID: 005039456**

**Algorithms and Data Structures (MSCS-532-B01)**

**Final Project Part 1: Optimization Technique and Implementation Project Report**

**Introduction**

High-Performance Computing (HPC) is critical in solving complex, large-scale problems in science, engineering, and data analytics. In these environments, even small inefficiencies can significantly increase execution time and resource usage when scaled across thousands of cores.

One of those massive inefficiencies in HPC is inefficient and slow algorithms. Azad et al. (2023) also identified inefficient algorithm implementations as one of the most common causes of performance degradation in HPC applications. Building on these findings, this project examined the impact of eliminating inefficient algorithms as an optimization strategy. The work included analyzing the empirical study and understanding the core issues, implementing a prototype in Python, and measuring its performance improvements.

**Overview of the Empirical Study**

The study, *An Empirical Study of High-Performance Computing (HPC) Performance Bugs* by Azad et al. (2023), studied the most used HPC applications and the bug reports for them. The authors classified performance bugs such as memory mismanagement, redundant computations, inefficient algorithms, and poor parallelization strategies.

The analysis revealed that inefficient algorithms caused 39.3% of performance bugs. This is a very high volume and a serious drag on the performance. Fixes typically involved replacing high-complexity routines with more efficient ones. Additionally, the study demonstrated that algorithmic improvements with better selection of the core data structures and processes often yielded greater benefits. The results exceeded low-level tuning techniques like loop unrolling or cache adjustments.

**Chosen Optimization Technique – Eliminating Inefficient Algorithm Implementation**

This project focused on replacing inefficient algorithms with better ones of lower time complexity. For example, changing an O(n2) approach to O(nlogn) or O(n) can significantly reduce execution time. This is especially the case when the data size increases drastically.

This technique was chosen because it:

1. Had the highest impact in the empirical study.
2. It could be easily applied across various programming languages and platforms.
3. Scaled effectively with large datasets.
4. It could be demonstrated clearly through a simple Python prototype.

**Strengths and Weaknesses in HPC Context**

Let us go through some of the strengths and weaknesses when working with this optimization technique of using a better algorithm.

**Strengths**

1. Delivered significant runtime reductions with straightforward changes in logic.
2. Required no specialized hardware knowledge.
3. Improved scalability for large datasets.

**Weaknesses**

1. An understanding of data structures and algorithm complexity is required, mainly the time complexity and memory usage.
2. Occasionally, code readability is reduced in cases where some APIs are faster but easier to read.
3. In Python, the interpreter overhead limits absolute performance compared to compiled languages.

**Prototype Implementation**

To demonstrate this technique, two versions of a frequency-counting function were developed. In both approaches, a dataset of 100,000 integers ranging from 0 to 1000 is generated, and then a simpler and optimized frequency counter function is implemented and analyzed. You can see the implementation for both below.

1. **Naïve approach:** In the first approach, the .count() function is used inside the loop, resulting in O(n2) complexity. This might not seem like a lot, but the time complexity increases drastically when the dataset increases. In this prototype, you can see from the results below that for the dataset of 100000, it took around 109 seconds to complete the counting. This dataset number could be in millions and billions in real-life scenarios, so a faster approach was guaranteed.
2. **Optimized approach:** In this optimized approach, a hash table (dictionary) is implemented to count the integer occurrences in a single pass, resulting in O(n) complexity. This is substantially faster than the previous approach. If you look at the result below, for the same dataset of 100000, this counting approach took sub-second, even in milliseconds, which is incredible faster than 109 seconds from before.

**Code Screenshot**

**A screen shot of a computer program

AI-generated content may be incorrect.  
A black background with white numbers

AI-generated content may be incorrect.**

**Performance Analysis**

The functions were benchmarked with datasets containing 10,000, 50,000, and 100,000 integers. Down below are the results from the analysis. Here n is the dataset size, while naïve\_min\_s, naïve\_mean\_s and naïve\_std\_s indicates the minimum, average and standard deviation of runtimes in seconds for the naïve implementation. Likewise, fast\_min\_s, fast\_mean\_s and fast\_std\_s indicates the minimum, average and standard deviation of runtimes in seconds for the optimized implementation. Finally, speedup\_mean\_x indicates how many times faster the optimized method is compared to the naïve one.

A black screen with white numbers

AI-generated content may be incorrect. The benchmark confirmed a dramatic separation between the naïve O(n2) and optimized O(n) implementations. For the naïve method, mean runtime grew from ~1.04 s at n=10,000 to ~109.36 s at n=100,000, reflecting the expected quadratic blow‑up. In contrast, the optimized dictionary‑based counter scaled near‑linearly, increasing from ~0.00063 s to ~0.00553 s over the same range. As a result, average speedup rose almost linearly with input size—from **~1,633×** at 10k to **~19,762×** at 100k. Standard deviations were minor across runs, indicating stable measurements. These results are consistent with algorithmic complexity theory and reinforce the empirical finding that algorithmic fixes (Azad et al., 2023) deliver the most significant performance gains in HPC-style workloads. These results were consistent with the findings of Azad et al. (2023), confirming that algorithmic optimization offers the most significant performance improvements in HPC workloads, especially as problem size grows.

Similarly, we can see from the graph (runtime vs input size) below that the naïve approach, which has the time complexity of O(n2), grows steeply with n. In contrast, the optimized solution with time complexity O(n) remains near linear and millisecond scale even at 100k. A graph with a line and a blue line

AI-generated content may be incorrect.

This second chart down below shows how many times faster the optimized method is compared to the naïve one. Speedup grows almost linearly with dataset size because the naïve algorithm is **O(n²)** while the optimized version is **O(n)**. At 10,000 elements, the optimized code is ~1,633× faster; by 100,000 elements, it’s ~19,762× faster — finishing in just a tiny fraction of the time required by the naïve approach.  
A graph with a blue line

AI-generated content may be incorrect.

**Challenges and Lessons Learned**

Here are some challenges and lessons learned while working on the optimization process.

**Challenges:**

1. Achieving consistent benchmark results in Python required averaging multiple runs to offset runtime variability.
2. The occurrence of Python's interpreter overhead. While relative gains were clear, absolute performance still lagged compiled implementations.

**Lessons Learned:**

1. Algorithmic inefficiency can be the primary bottleneck in HPC applications.
2. Profiling before optimizing ensures that effort is focused on the true performance hotspots.
3. Even minor algorithmic improvements can yield dramatic speedups at scale.

**Conclusion**

This project applied the optimization technique of eliminating inefficient algorithms to a frequency-counting problem in Python. The optimized implementation achieved a very high-performance boost (speedups of over 100×) for large datasets. These findings supported the empirical study's conclusion that algorithmic improvements often yield the most substantial performance benefits in HPC systems.

Future work could involve adapting the optimization to parallel or distributed counting, integrating with compiled extensions for further speed gains, and testing on real HPC workloads such as graph analytics or numerical simulations.

**References**

Azad, M. A. K., Iqbal, N., Hassan, F., & Roy, P. (2023). An empirical study of high performance computing (HPC) performance bugs. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), 194–206. <https://doi.org/10.1109/MSR59073.2023.00037>

Zhao, Y., Xiao, L., Wang, X., Sun, L., Chen, B., Liu, Y., & Bondi, A. B. (2020). How are performance issues caused and resolved? — An empirical study from a design perspective. Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering, 181–192. <https://doi.org/10.1145/3358960.3379130>

Gravelle, B. (2019). Understanding the performance of HPC applications. University of Oregon. <https://www.cs.uoregon.edu/Reports/AREA-201903-Gravelle.pdf>

Li, Y., He, J., & Tang, Y. (2021). Improving high performance computing efficiency through algorithmic optimization. Journal of Supercomputing, 77(7), 6936–6955. <https://doi.org/10.1007/s11227-020-03457-3>

Ekanayake, J., & Fox, G. (2010). High performance parallel computing with clouds and cloud technologies. Cloud Computing, 20(1), 20–27. <https://doi.org/10.1007/978-1-4419-6524-0_4>

Shan, H., & Singh, J. P. (2001). A comparison of MPI, SHMEM, and cache-coherent shared address space programming models on the SGI Origin2000. Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 1–12. <https://doi.org/10.1145/582034.582037>