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

Shrisan Kapali

Student Id: 005032249

MSCS 532 – Algorithms and Data Structures

Satish Penmatsa

February 23, 2025

            High-performance computing (HPC) solves complex real-world computing problems by aggregating clusters of powerful processors working parallelly and processing enormous multidimensional data sets (Smalley & Susnjara, 2024). Compared to traditional desktops and laptops, they have an average processing speed of one million times faster (Smalley & Susnjara, 2024). An HPC consists of multiple high-performance CPUs or GPUs, and a single HPC cluster may include 100,000 or more of these computers, which are called nodes.

In the empirical study of HPC conducted by Azad, Iqbal, Hassan, and Roy, they found the challenges of HPC in achieving high efficiency and scalability because of the complex hardware architecture, inefficient algorithms and memory management, data structure design, and many more (Azad et al., n.d.). Upon analyzing 1729 HPC performance commits collected from 23 real-world projects, Azad and his team members identified significant challenges HPC faced and provided solutions on how these challenges can be optimized.

The empirical study conducted by Azad et al. (n.d.) identified several challenges, including inefficiencies in data structure algorithms, issues related to memory and data locality, computationally expensive and redundant operations, inefficiencies in compiler code, and a lack of parallelism. These challenges were revealed through a careful analysis of the project commits.

Several performance optimization techniques were discussed in the empirical study to mitigate these inefficiencies and challenges. Micro-architecture-specific optimization techniques were discussed, including operator strength reduction, the use of data types to reduce computation and memory overhead, data locality, and structure optimization. Domain-specific optimization techniques were also discussed, such as eliminating unnecessary operations and reducing loop iteration. Compiler optimization using function inlining or loop unrolling, introducing parallelism, memory management, and domain and architecture agnostic algorithm and data structure optimization were some of the other optimization techniques that Azad and his team members discussed in their empirical research.

Among all the optimization techniques discussed, optimizing domain and architecture-agnostic algorithms and data structure by reducing the asymptotic complexity of the search algorithm stood out the most to me. As HPC processes massive amounts of data and performs complex calculations, search operations are fundamental as they are used in data indexing, Machine Learning and AI, financial analytics, and scientific computing in several aspects (“HPC Applications & Real-World Examples,” 2023). Processing big data sets of terabytes or petabytes and filtering the applicable information in real-time are vital operations in HPC. As a result, the usage of efficient search algorithms that can tailor the required information with good time and memory complexity is significant.

Today, with the availability of hardware configurations of HPC, especially GPUs, Deep Learning and Neural network advances have become possible (Kaveh & Mesgari, 2022). In the research by Kaven and Mesgari (2022), they identified the limitations of gradient-based algorithms and the need to optimize them to reduce expensive executive time. Machine Learning, Deep Learning, and Neural Networks, in some cases where HPC is often used, rely significantly on processing the data and searching through a large data set.

Modification and hybridization of meta-heuristic algorithms are used to improve search algorithm performance (Kaveh & Mesgari, 2022). Their time complexity usually determines the efficiency of these search algorithms. In the empirical study, it has been proven that usage of efficient search algorithms can reduce execution time from O (n) to O (log n) or even constant time O (1), which in terms of big data can be hours of computational work reduced to minutes or seconds (Azad et al., n.d.).

Choosing an efficient search algorithm begins with selecting a fast data structure interface. Many applicable data structures like array, list, queues, heap, and stacks can be used to store the collection of the same elements; however, analyzing their time, space complexity, and use case can greatly influence the system’s performance. For example, dictionaries have an average search time complexity of O (1), while arrays and lists have O (n) (Hunner, 2025).

However, dictionaries do have a higher memory overhead, and in applications where the size of the data set is known, using an array of fixed length can be memory efficient, particularly in systems where memory is highly constrained. If the data set is always inserted in ordered list, the usage of binary tree may be preferred as in binary tree the data values are usually sorted (“Binary tree vs. linked lists vs. hash table,” 2023).

In addition, using complex search algorithms such as divide-and-conquer algorithms, graph search, or greedy algorithms can significantly optimize search performance. For example, in applications where bubble sort, which has an average time complexity of O (n2), is used, it can be optimized using divide and conquer algorithms such as quicksort or merge sort, which have an average time complexity of O (n logn) (“Divide and conquer algorithm,” 2025). Even in the empirical study, usage of priority queue helped achieve O (1) time for peek operations, and using binary search reduced the atom lookup time complexity to O (logn) (Azad et al., n.d.).

Greedy algorithms such as Dijkstra's, Kruskal's minimum spanning tree, and Huffman's coding implementation are widely used in HPC. In real-life applications, these algorithms play an important role in networking, e-commerce, Machine Learning, and gaming ("Hands-on guide: Real-world greedy algorithm applications," 2024). One of the most used applications, GPS navigation, uses Dijkstra's algorithm to find the shortest and quickest route.

In HPC applications where linear search is implemented, the performance can be improved by using binary search or even complex or hybrid search algorithms combining multiple search algorithms. In the search algorithm performance research conducted by Mastripolito et al. (2021), they compared the performance of different algorithms such as linear and binary search, hunt & locate search, log hash search, exponent hash search, and skiplist search against different data sizes and distribution and under different compilers. It was determined that depending on how data was distributed and the data's size, the search algorithms' performance varied (Mastripolito et al., 2021). Thus, depending on the nature and size of the data, HPC can significantly improve its performance by implementing applicable efficient search algorithms.

In the context of HPC, depending on the application that is being optimized, the search data may consist of millions of variables and constraints (Liu et al., 2022). Efficient search operations in such datasets require database management systems to implement techniques to minimize data retrieval and search time. Database indexing strategies can enhance search or query operations (Jyewfatt, 2025). Large-scale and complex databases can improve their data lookups by using indexes to quickly find the relevant data and drastically improve the search time complexity. By using key indexing techniques such as B-trees, hash indexing, bitmap indexing, and R-tree, the time complexity for search and query can greatly be optimized in HPC (Jyewfatt, 2025). However, database indexing can slow down the write operations. This is because, for each write, the database needs to update all the relevant indexes (Jyewfatt, 2025). In addition, over-indexing can greatly slow down the write, update, and delete operations in systems where the data is frequently manipulated.

HPC consists of clusters of powerful CPUs and GPUs. The search operations can be optimized for performance by utilizing hardware resources with parallel computing, multi-threading, caching, and memoization techniques. The empirical study found that caching and memoization eliminated the need for recomputing and vastly improved search algorithm performance (Azad et al., n.d.). Search algorithms using parallel computing can significantly improve performance by dividing the task load across multiple processors. In addition, as caching and memoization store the search results, these techniques can significantly improve the search performance in HPC.

Despite the advantages of the complex search algorithms in HPC, as discussed above, these algorithms pose significant challenges. Depending on the type of data structures used, trade-offs in time and space complexity must be carefully balanced to achieve optimal performance. The search performance may be higher in a data structure that uses hash tables or binary trees, but they are memory intensive, and in the context of large data sets, they need high extra memory space. Caching and parallelization in HPC boost performance, but their memory trade-off can be critical in datasets on limited memory hardware.

In conclusion, reducing the asymptotic complexity of the search algorithm can significantly improve the search efficiency in HPC. However, they introduce significant challenges related to time and space trade-offs. The choice of data structures used determines the search time complexity, and solutions such as implementing hybrid search algorithms, distributed indexing, caching, and parallelization strategies can help optimize these HPC search algorithms trade-offs. In addition, depending on the nature and size of the data, the usage of applicable search algorithms can significantly improve the overall performance of HPC. Along with the search algorithms optimization, other optimization techniques mentioned in the empirical study, such as the elimination of unnecessary operations and efficient cache access, can be combined to optimize the HPC performance

**Python search optimization prototype**