UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING AND ELECTRICAL AND APPLIED MATHEMATICS

Master's Degree in Computer Engineering

Project report

RB Tree Search

Lecturer: Francesco Moscato fmoscato@unisa.it

Student and Creator: Agostino Cardamone 0622702276 a.cardamone7@studenti.unisa.it

ACADEMIC YEAR 2023/2024

Index

1. Introduction

1.1 Overview of the Project

This project focuses on the parallelization and performance evaluation of a Red-Black AVL Tree Search algorithm, a complex hybrid data structure known for its efficiency in search operations. The primary goal is to develop and analyze parallel versions of this algorithm using two distinct computational approaches: "OpenMP + MPI" and "OpenMP + CUDA." A critical part of this study involves comparing these parallel implementations with a traditional sequential version of the same algorithm. This comparison aims to highlight the performance enhancements achieved through parallelization. The project emphasizes the importance of varying the types and sizes of input data to thoroughly assess the scalability and adaptability of each implementation. Notably, the parallel algorithm designed for the "OpenMP + MPI" framework will be different from the one developed for "OpenMP + CUDA," thereby providing a comprehensive insight into the strengths and limitations of each parallel computing method. This analysis will not only demonstrate the potential benefits of parallel computing in optimizing complex algorithms but also offer valuable comparisons between parallel and sequential processing paradigms in data-intensive tasks with the personal hardware.

1.2 Objectives and Scope

The primary aim of this project is to practically apply the knowledge acquired during the course by implementing the Red-Black AVL tree search algorithm in two parallel computing environments: OpenMP combined with MPI, and OpenMP in conjunction with CUDA. The focus is not on creating sophisticated or intricate implementations but on applying fundamental parallel computing concepts to enhance the algorithm's performance. This endeavor seeks to compare the effectiveness of these parallelization strategies and to evaluate how they perform against the more traditional, single-node sequential approach.

The scope of the project is rooted in real-world applicability and accessibility, with testing and development being constrained by the capabilities of available personal hardware. It involves a series of performance tests, employing various data types and sizes to examine how different inputs impact the algorithm's efficiency in each computing environment. Through this process, the project aims to deliver a grounded and practical analysis of the theorical performance improvements that parallel computing can offer. This exploration will provide valuable insights into the scalability of the chosen methods and their potential applications in typical computing scenarios, reflecting the realities of hardware limitations and the practicalities of implementing parallel computing techniques.

2. Background and Theory

2.1 Red-Black AVL Tree Search Algorithm

The Red-Black AVL Tree Search Algorithm is a hybrid data structure that combines elements from two types of self-balancing binary search trees: the Red-Black Tree and the AVL (Adelson-Velsky and Landis) Tree. Red-Black Trees maintain balance through rules centered around node colors (red or black), ensuring that the path from the root to the farthest leaf is no more than twice as long as the path to the nearest leaf. This feature helps keep the tree balanced with every insert and delete operation, maintaining an *O(log n)* time complexity for these operations.

On the other hand, AVL Trees focus on balancing the height of the tree. They maintain a balance factor for each node (the difference in heights between left and right subtrees), ensuring it remains within -1, 0, or +1. This tight balancing act makes AVL Trees more rigidly balanced compared to Red-Black Trees.

The Red-Black AVL Tree Search Algorithm, therefore, seeks to leverage the strengths of both these structures. It aims to maintain an efficient balance that facilitates quick search operations while also ensuring that insertions and deletions do not significantly disrupt the tree's structure. The algorithm is particularly effective in scenarios where the tree undergoes frequent modifications, as it can maintain balance with minimal restructuring.

2.2 Parallel Computing: OpenMP, MPI, and CUDA

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It allows for efficient use of multi-core processors in a single system. OpenMP simplifies the development of parallel applications by enabling parallel execution of code with simple directives. It's effective for shared-memory architectures where threads can be dynamically allocated to parallelize tasks such as loops.

OpenMP serves as a valuable solution for parallelizing existing code initially developed for single-core execution. However, there are instances where adopting OpenMP may not yield significant benefits. One reason is when the code is already optimized for single-core performance, rendering parallelization less impactful. Additionally, OpenMP is primarily tailored for parallelizing loop-intensive code, commonly found in scientific computing problems. Codes with minimal loop dependencies may not experience substantial gains from parallelization. Another factor limiting the suitability of OpenMP is the need for extensive inter-core communication. In cases demanding high communication levels between cores, alternative parallel libraries like MPI, designed for efficient inter-core communication, may be more appropriate.

MPI (Message Passing Interface) is a protocol for parallel programming across multiple nodes in a distributed computing environment. MPI is designed for systems where each processor has its own memory (distributed memory systems). It enables communication between processes running on different nodes through message passing, making it suitable for large-scale parallel processing tasks that require coordination between multiple processors.

MPI stands as a valuable solution for parallelizing pre-existing code originally designed for single-core execution. However, there are scenarios where embracing MPI might not result in significant advantages. One such situation arises when the code is already fine-tuned for optimal single-core

performance, diminishing the impact of parallelization. Moreover, MPI excels in handling parallelization across distributed memory systems and is well-suited for applications that require communication between processes running on separate nodes.

MPI is particularly adept at addressing the challenges of extensive inter-process communication. In contrast to OpenMP, which is more focused on shared-memory architectures, MPI shines in scenarios demanding high levels of communication between processes, especially in large-scale distributed computing environments. Its capability to facilitate efficient message passing between independent processes makes MPI a preferable choice when intricate coordination and communication among processors are crucial.

CUDA (Compute Unified Device Architecture), created by NVIDIA, is a parallel computing platform using NVIDIA GPUs. It allows developers to leverage the power of GPUs for general-purpose processing. CUDA is effective for computations that can be highly parallelized, utilizing the thousands of cores in a GPU to perform simultaneous calculations, making it ideal for data-intensive tasks that require massive computational power.

One of CUDA's notable strengths is its proficiency in parallelizing tasks across the numerous cores within a GPU, allowing for efficient and high-throughput processing. This parallel approach significantly accelerates computations, especially for applications where parallelization can be effectively exploited. Additionally, CUDA's architecture is tailored to streamline parallel processing, making it an excellent choice for tasks that can be divided into parallelizable subproblems.

Each technology offers distinct advantages for parallelizing the Red-Black AVL tree search algorithm. OpenMP enables thread-level parallelism on a single machine, MPI facilitates scaling across multiple machines in a network, and CUDA is ideal for high-performance computations on GPUs. Understanding their unique capabilities is crucial for optimizing the algorithm in different parallel computing environments.

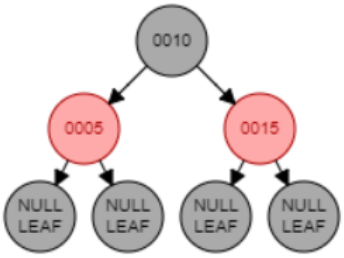
3. Project Description

3.1 Implementation of the Red-Black AVL Tree Structure

In the context of this project, a Red-Black AVL Tree structure has been implemented using RBMatrix, a sparse matrix representation. The decision to employ this approach is geared towards optimizing memory usage, especially in the handling of larger datasets, achieved by storing nodes in an array format rather than relying extensively on pointers.

The choice of the sparse matrix format in *RBMatrix* is motivated by the need to address potential inefficiencies in memory allocation and access speed that may arise with conventional tree structures. Representing the tree in an array results in a more compact and memory-efficient storage method.

Index	0	1	2	3	4	5	6
Value	10	Null	5	Null	15	Null	Null



The array-based storage in RBMatrix brings about a modification in how nodes are accessed compared to traditional pointer-based trees. Direct access to any node through its index eliminates the necessity of traversing the tree, thereby enhancing efficiency in operations such as searching and updating nodes. The linear nature of the array also simplifies the iteration over tree nodes.

To accommodate unpredictable tree growth, RBMatrix incorporates a *resizeMatrix* function, enabling the tree to dynamically adjust its size. This functionality ensures that the array expands appropriately as nodes are added, mitigating issues like memory overflow. Dynamic resizing proves particularly useful for applications dealing with varying dataset sizes.

The implementation encompasses standard rebalancing functions specific to Red-Black Trees, such as *insertValue* and *rebalanceInsert*. These functions are not only implemented but are also adapted to function within the sparse matrix context. For instance, rebalanceInsert plays a crucial role in maintaining the tree's balance after each insertion, ensuring adherence to Red-Black Tree properties.

Fundamental tree rotations, such as *rotateLeft* and *rotateRight*, are implemented to modify the tree's structure while preserving its properties. These rotations are essential, especially after insertions or deletions, to uphold the tree's balance. Balancing is critical for efficient search and traversal times, ensuring the tree's height remains logarithmic relative to the number of nodes.

The design of RBMatrix proves advantageous for applications handling large datasets. Its ability to dynamically grow and the memory efficiency of the sparse matrix format make it well-suited for scenarios prioritizing data size and efficiency.

A notable feature of RBMatrix lies in its memory efficiency. Through the adoption of an array-based approach and the storage of only essential node information, the structure utilizes memory more judiciously compared to traditional pointer-based trees. This efficiency becomes increasingly significant as the tree size grows, rendering RBMatrix a practical choice for large-scale applications.

3.2 Sequential Implementation of Red-Black Tree Search

n this project, the sequential implementation of a Red-Black Tree search algorithm is encapsulated in the RBSequential.c file. This implementation is a fundamental part of the larger exploration into the comparative efficiencies of parallel computing paradigms. It provides a baseline against which the performance of more complex parallel implementations using OpenMP + MPI and OpenMP + CUDA can be assessed.

At the core of this sequential approach is the *searchRBTree* function. It operates by traversing the tree, represented internally as a matrix (RBMatrix), in a linear fashion. Starting from the root node, the function methodically navigates through the tree structure, comparing the search value with the value at the current node. The process leverages the binary search tree property, which significantly narrows down the search path at each step. If a match is found, the function returns the index of the node containing the search value. Otherwise, if the search reaches a leaf node without finding the value, it concludes that the value isn't present in the tree and returns -1. This straightforward yet effective search mechanism forms the groundwork for understanding and evaluating the efficiencies brought in by parallel processing techniques.

The main function in *RBSequential.c* sets the stage for the search operation. It starts by parsing command-line arguments to determine the number of nodes, the seed for random value generation, and the file paths for output. Following this, it initializes the RBMatrix and fills it with randomly generated values, ensuring the tree is populated and ready for the search process. A random value is then chosen as the target for the search, and the *searchRBTree* function is invoked to locate this value within the tree. This function's execution is timed to quantify the search efficiency, offering a clear insight into the performance of the sequential search algorithm.

3.3 Parallel Implementation of Red-Black Tree Search Using MPI and OpenMP

The task splitting strategy and search parallelization in the implementation of the Red-Black tree search algorithm using MPI and OpenMP were designed to optimize efficiency on distributed computing systems.

3.3.1 Division of Tasks with MPI

In dividing tasks, the approach taken considers the total number of nodes in the RBMatrix and the number of MPI processes available. Each MPI process receives a segment of the tree, with the goal of distributing the work evenly. This choice is driven by the desire to balance the load among processes to avoid some processes sitting idle while others are overloaded, a phenomenon known as "load imbalance."

The importance of this strategy lies in the fact that, in a distributed environment, execution time is not determined by the fastest process, but by the slowest process. Therefore, ensuring that each process has a comparable amount of work is critical to reducing the total execution time.

Task splitting is implemented so that each process has a contiguous portion of the tree. This approach is preferred over a more fragmented split because it reduces complexity in data management and process synchronization. Having a contiguous block of nodes allows each process to search without having to frequently interact with other processes, thus reducing overlap in MPI communication.

Example of Task Division: Let's consider an RBMatrix with 100 nodes and 4 MPI processes. If each process handles a quarter of the tree, the division can be visualized as follows:

MPI Process	Start Node Index	End Node Index
Process 1	0	24
Process 2	25	49
Process 3	50	74
Process 4	75	99

This division ensures that each process has a nearly equal portion of the tree to analyze, thus allowing for more efficient utilization of computational resources.

3.3.2 Researcher Parallelization with OpenMP

Once each MPI process receives its segment of the tree, the search in that segment is parallelized with OpenMP. This level of internal parallelization allows effective use of the multicore capabilities of each node in the cluster. OpenMP is chosen for its ability to dynamically manage the workload between threads, an essential feature for efficiently handling variations in the size of tree segments.

In each MPI process, the *parallelSearchInSubtree* function is responsible for the parallel search. Here, OpenMP threads perform an independent search on different nodes in the assigned segment. The choice to use OpenMP within each MPI process takes full advantage of the hardware architecture, allowing each processor core to actively contribute to the search.

The advantage of this internal parallelization is that it allows a much faster search of each segment. Instead of examining nodes one by one, threads can examine several nodes at once. The *reduction* clause in OpenMP plays a key role here, aggregating the results of all threads to find the minimum index where the searched value was found in the segment.

Example of Parallel Search in a Segment: Suppose Process 2 is assigned the segment from node 25 to node 49 and uses 4 threads. The workload division among the threads can be represented as follows:

OpenMP Thread	Start Node Index	End Node Index
Thread 1	25	31
Thread 2	32	38
Thread 3	39	45
Thread 4	46	49

Each thread independently examines its sub-portion of the segment. For instance, if Thread 3 finds the sought value at node 42, this information is shared among the threads to identify node 42 as the minimum index where the value was found in that segment.

3.3.3 Memory Management for MPI Processes

In the MPI framework, each process operates in its own memory space. When the RBMatrix, representing the Red-Black Tree, is divided among the MPI processes, it's essential to ensure that each process has access to its respective segment of the tree. This is achieved through a combination of memory allocation and data distribution techniques.

Initially, the root process creates the entire RBMatrix and populates it with data. This matrix is then shared across all MPI processes using MPI communication functions. The *MPI_Bcast* function is particularly useful in broadcasting the metadata of the RBMatrix (like the total number of nodes) to all processes. Subsequently, each process allocates memory for its segment of the RBMatrix. This

approach ensures that every process works with its data copy, thereby preventing data access conflicts and the need for complex synchronization mechanisms.

3.3.4 Memory Management within OpenMP Threads

In contrast to MPI processes, OpenMP threads within a single process share the same memory space. This shared memory model simplifies data access but requires careful consideration to avoid *race conditions* and ensure *thread-safe operations*.

When the search task is parallelized using OpenMP within an MPI process, each thread works on a specific subset of the assigned RBMatrix segment. Since threads can access the same memory space, there's no need for separate data allocation for each thread. However, care is taken to partition the search task so that each thread works on a different part of the segment to avoid overlapping searches and potential conflicts.

The shared memory model of OpenMP is advantageous for searching operations, as it allows threads to access and search through the RBMatrix without the *overhead* of data replication. The use of OpenMP directives, particularly the *parallel for* construct, facilitates the distribution of the search task among the threads while ensuring that the threads do not interfere with each other's operations.

3.4 Parallel Implementation of Red-Black Tree Search Using CUDA and OpenMP

In the parallel implementation of the Red-Black Tree search algorithm using CUDA and OpenMP, the design choices reflect a focus on maximizing the computational power of the GPU while attempting to leverage CPU parallelism through OpenMP. The CUDA and OpenMP version of the Red-Black AVL tree search algorithm uses a specific strategy to take full advantage of the parallel processing capabilities of GPUs. This approach differs significantly from the MPI and OpenMP version, mainly because of the inherent architectural differences between GPUs and CPUs.

3.4.1 Workload Subdivision Strategy

In the CUDA implementation, the search workload in the tree, represented by RBMatrix, is uniquely divided. The matrix is divided into subtrees, assigned to different CUDA blocks. This division is based on optimizing the parallel processing capabilities of the GPU. By dividing the tree into subtrees, the algorithm ensures that each CUDA block has a distinct section of the tree to process, allowing simultaneous searches in different parts of the tree.

The choice of dividing the tree into subtrees for different CUDA blocks stems from the need to balance the workload among the available GPU cores. This method is particularly effective for GPUs because it aligns well with their architecture, which is designed to efficiently handle multiple parallel threads. Unlike MPI processes, which require more intensive communication, CUDA blocks can operate more independently, reducing the overhead of inter-process communication.

3.4.2 Implementation of Parallel Search in CUDA

Within each CUDA block, individual threads undertake the task of searching within their assigned subtree. This fine-grained parallelism is one of the key strengths in using CUDA for tree search algorithms. Each thread traverses its section of the tree independently, comparing node values with the search target.

The implementation takes full advantage of shared memory within CUDA blocks to store intermediate results and synchronize the search operation between threads in the same block. This approach minimizes global memory access, which is a slower operation, thus improving the overall performance of the search process.

In the implementation of the parallel Red-Black Tree search using CUDA, two distinct versions of the CUDA kernel have been developed to optimize the search process within subtrees of a Red-Black Matrix. These versions are *searchSubtreeKernel* and *balancedSearchSubtreeKernel*, each offering a unique approach to parallelization and reduction strategies within the GPU environment.

- The ***searchSubtreeKernel*** focuses on efficiently distributing the workload across multiple CUDA threads. Each thread in a block is tasked with searching a subset of rows within the matrix, aiming to optimize the search process for a specific target value. This kernel utilizes shared memory to store local minimum indices found by individual threads. After each thread computes its local minimum, a reduction operation is performed within the block to determine the global minimum index. The kernel employs atomic operations, specifically *atomicMin*, to ensure consistent updating of the global minimum index.

Example for searchSubtreeKernel

Imagine a red-black tree represented as a sparse array of 16 nodes (for simplicity). Suppose we have 4 CUDA blocks, each with 4 threads. The value to be searched is "42".

Each block deals with a portion of the matrix (e.g., blocks 1-4 deal with nodes 0-3, 4-7, 8-11 and 12-15, respectively).

Block 1	**Block 2**	**Block 3**	**Block 4**
Node 0	Node 4	Node 8	Node 12
Node 1	Node 5	Node 9	Node 13
Node 2	Node 6	Node 10	Node 14
Node 3	Node 7	Node 11	Node 15

1. Each thread in a block checks the value in a node.
2. If a thread finds the value "42," it writes it into a shared array in the block.
3. Finally, a *reduction* is performed within the block to find the minimum index where "42" appears.

Each block has a shared array in memory to keep track of the indexes found.

- In contrast, the ***balancedSearchSubtreeKernel*** introduces a hybrid reduction methodology that combines *warp-level* and *block-level* reductions, offering a more advanced approach to parallel subtree search. This kernel leverages *__shfl_down_sync* for reduction within warps. This function allows threads in a warp to communicate and compute the minimum index, minimizing the use of shared memory and potentially reducing contention. The kernel employs a combination of warp and block-level strategies. The minimum index from each warp, calculated using warp-level reduction, is written to shared memory. The first thread in each block then performs the final aggregation of these warp-level results to find the global minimum index.

Example for balancedSearchSubtreeKernel

This kernel uses both warp-level and block-level reduction. Consider the same scenario as above, but with more advanced reduction.

1. As before, each thread in a block looks for the value "42" in its portion of the tree.
2. Each thread in a block examines a subset of the nodes.
3. In block 2, the thread examining node 7 finds the value "42".
4. Within each warp, reduction is performed with *__shfl_down_sync*.
5. Suppose the warp that found the value "42" has a minimum index of 7.

Global memory was used to store the tree structure (nodes) and the result (index found) while the shared memory was used to store the intermediate results of the warp-level reduction and for the final block-level reduction. This advanced reduction enables efficient use of the GPU, reducing the amount of shared memory required and minimizing synchronization overhead between threads. This makes the balancedSearchSubtreeKernel particularly effective for searching large parallel data structures, such as a red-black tree represented as a sparse matrix.

Both these kernels are tailored to leverage the parallel processing capabilities of GPUs, but they differ in their approach to handling and reducing data.

3.4.3 Memory Management in CUDA

Memory management in CUDA is handled separately from the MPI and OpenMP implementation. CUDA kernels are designed to effectively use both global and shared memory. Global memory stores the tree structure, while shared memory is used for intermediate data during the search process. This separation in memory use is critical for performance since access to global memory is slower than to shared memory.

Global memory is the main memory of the GPU and is used to store long-term or shared data between different blocks and threads. In the search implementation, the Red-Black tree structure (RBMatrix) is stored in this memory. Since global memory is accessible by all threads on all blocks, it is the ideal place to store the complete tree structure. This approach, however, can slow performance by being relatively slow to access this memory. The tree data is initially created in the host (CPU) context and then transferred to the device's global memory (GPU) through *cudaMemcpy* calls.

Shared memory in CUDA is the fastest but smallest memory accessible to all threads within a block. It is used to store data that needs fast access or for data that is used by multiple threads within the same block. In the CUDA implementation, during the search, each thread calculates a minimum local index and then uses reduction techniques within the shared memory to determine the minimum block-level index. This operation significantly reduces the need to write to global memory.

Atomic functions such as *atomicMin* are used to update variables shared among multiple threads, such as the minimum global index found. These functions ensure that the update of values is consistent and free of race conditions, which are crucial in a parallel execution environment.

3.4.4 Role of OpenMP in CUDA Implementation

In this hybrid approach, OpenMP plays a supporting role, primarily used for initializing data structures and handling pre-processing tasks before handing off the intensive search operation to the GPU. The use of OpenMP in this context is limited because the parallel processing power of the GPU (through CUDA) is more significant for the search operation. Therefore, OpenMP's contribution to the actual search process in the CUDA implementation is minimal compared to its role in the MPI and OpenMP approach.

The role of OpenMP is distinct from and complementary to that of CUDA, focused primarily on data preparation and initialization, rather than on the intensive computation of actual search. This choice is due to the diversity of parallel processing capabilities offered by OpenMP (CPU-centric) and CUDA (GPU-centric). OpenMP is used to parallelize operations that are most efficiently handled by the CPU. This includes initializing data structures such as RBMatrix and preparing the data for transfer to the GPU. Doing so reduces the time spent in the preparation phase before passing the data to the GPU, contributing to a better overall performance balance.

The GPU, through CUDA, handles the most computationally intensive part of the search operation. The parallel processing capability of the GPU far exceeds that of the CPU for this type of task, making CUDA the ideal choice for the tree search operation. CUDA is designed to take advantage of the parallel processing capabilities of thousands of GPU threads. This results in much faster and more efficient search than what could be accomplished with the CPU alone.

4. Experimental Setup

4.1 Hardware Configuration

The parallel implementations of the Red-Black tree search algorithm were tested on the following hardware configuration:

- **CPU**: AMD Ryzen 5 7640HS
- **Cores**: 6 physical cores, each with 2 threads, totaling 12 threads.
- **Technology**: 4 nm.
- **Core Speed**: 4211.0 MHz.
- **L1 Cache**: 6 x 32 KB for both data and instructions.
- **L2 Cache**: 6 x 1024 KB.
- **L3 Cache**: 16 MB.
- **Instruction Sets**: Supports advanced instructions like AES, AVX, AVX2, AVX512, and FMA3.
- **TDP**: 45.0 Watts.

This CPU, with its multi-core architecture and high clock speeds, effectively handled parallel workloads distributed across various threads, particularly in implementations using OpenMP.

- **GPU**: NVIDIA GeForce RTX 4060 Laptop GPU
- **CUDA Cores**: 24.
- **Base Frequency**: 2010.00 GHz.
- **Global Memory**: 8.00 GB.
- **Shared Memory per Block**: 48.00 KB.
- **Maximum Number of Threads per Block**: 1024.
- **Maximum SM (Streaming Multiprocessors) Count:** 24
- **Maximum Thread Grid Size:** 2147483647 x 65535 x 65535
- **Maximum Thread Block Size:** 1024 x 1024 x 64
- **Kernel Execution Capability:** 8.9

In practical applications, this GPU configuration excels in accelerating tree search operations through parallel implementations using CUDA. The ability to concurrently execute a significant number of threads across multiple CUDA cores results in a substantial boost in performance, particularly when compared to CPU processing. This makes the GPU well-suited for tasks that demand efficient parallelization and benefit from the simultaneous execution of a multitude of threads.

- **Memory Type**: DDR5.
- **Memory Size**: 16 GB.
- **Memory Speed**: 5600 MHz.

4.2 Software Environment

Operating Systems:

- **Primary**: Windows 10 Pro, Version 22H2 (Build 19045.3803)
- **Secondary**: Ubuntu 22.04.3 LTS via Windows Subsystem for Linux (WSL)

Development Environment:

- Visual Studio Code, Version 1.85.1

Compilers:

- **GCC**: gcc.exe (Rev7, Built by MSYS2 project) 13.1.0
- **NVCC (CUDA Compiler)**: NVIDIA Cuda compiler driver, release 12.3, V12.3.103
- **MPI Compiler**: mpicc (Rev7, Built by MSYS2 project) 13.1.0

Other Software:

- **Python**: Version 3.11.7
- **CMake**: Version 3.28.1

4.3 Compilation and Execution Details

To compile and run the programs for the tests, the compilers mentioned in the previous section were used while for execution the tests were carried out on both the primary operating system i.e. Windows 10 Pro 22H2 and the secondary operating system Ubuntu 22.04.3 LTS via Windows Subsystem for Linux (WSL).

The compilation of the programs was done the four levels of compiler optimization (-O0, -O1, -O2, -O3) in a progressive manner to evaluate the impact on performance with the use of these optimizations.

For program execution in the various tests, the values were chosen so that the trend and performance could be evaluated as a function of various types of input: for data values to work on, number of OpenMP threads, number of MPI processes, and to randomize the creation of node values, two seeds were chosen to ensure randomness and eliminate bias in the input data sets.

The various tests conducted can be modified simply by going to define new values such as:

- Number of values to be entered in the red black tree
- Number of Max OMP Threads in parallel execution (depends on the CPU in use)
- Number of OMP Threads
- Number of MPI Processes
- Values for seed

for the input parameters to the tests within the makefile.

The first set of tests performed was on the first version of the operating system which is Windows 10.

The second group of tests, on the other hand, was performed using the second version of the operating system namely Ubuntu 22.04.3 LTS via Windows Subsystem for Linux (WSL).

5. Performance Analysis

5.1 Methodology for Performance Testing

The methodology for performance testing of the Red-Black Tree implementations was designed to be thorough and systematic, ensuring that the tests could measure the effectiveness and efficiency of the code under a variety of conditions.

The tests were conducted across multiple dataset sizes and configurations to assess the scalability and performance impact of the Red-Black Tree operations. Two sets of numerical values (*VALS1* and *VALS2*) were chosen to represent different scales of the problem size, allowing for the observation of the algorithms' behaviors on both medium and large datasets.

The first data set (*VALS1*) consists of:

- 200,000 values to be entered.
- 2,000,000 values to be entered.
- 20,000,000 values to be entered.

This initial set is used in the first test conducted on the Windows operating system, with the objective of evaluating the performance of the implementations implemented across a wide range of values to be generated and processed and then proceeding to search for these values within the matrix representation of the red-black tree.

The second data set (*VALS2*) consists of:

- 20,000 values to be entered.
- 200,000 values to be entered.
- 2,000,000 values to be entered.

This set is employed in the second test run on the Ubuntu operating system, with the intention of evaluating implementations across a smaller range of values than in the first test. The goal is to analyze the performance of implementations with smaller values and compare them with the results obtained in the first test.

Seed values (*SEED1* and *SEED2*) were used to maintain the consistency of the data across different runs, ensuring that the tests were fair and that performance variations were due to the execution environment and the algorithm itself, rather than differences in the data.

The tests were executed with a varying number of OpenMP threads (THREADS) and MPI processes (MPI_PROCESSES) to determine the impact of this parameter on the parallel execution on the performance. This varied number of OpenMP threads from single-threaded execution to using the maximum number of threads supported by the hardware (MAX_THREADS), which in this case was 12.

The limitation to a maximum of 12 concurrent threads is imposed by the hardware configuration, specifically the AMD Ryzen 5 7640HS processor used in the system. This processor features 6 physical cores, each with 2 threads, totaling 12 threads. Therefore, the maximum number of concurrent threads is set to match the available hardware resources. Exceeding this limit could result in suboptimal resource utilization and potential performance degradation. The restriction is in line with optimizing the parallel execution for the given hardware configuration to ensure efficient use of available processing capabilities.

All possible combinations tested (using at most 12 total threads running simultaneously) for every test set of values are as follows:

Version	OMP Threads	MPI Processes
OpenMP+MPI	1	1
OpenMP+MPI	1	2
OpenMP+MPI	1	4
OpenMP+MPI	1	12
OpenMP+MPI	2	1
OpenMP+MPI	2	2
OpenMP+MPI	2	4
OpenMP+MPI	4	1
OpenMP+MPI	4	2
OpenMP+CUDA	1	0
OpenMP+CUDA	2	0
OpenMP+CUDA	4	0
OpenMP+CUDA	12	0

All of these were tested for each set of values, although the specific details of the values were omitted for simplicity. In addition, they were evaluated for each compiler optimization configuration.

5.2 Analysis of Execution Times and Speed-Up

The performance analysis for the Red-Black Tree search involves evaluating the efficiency and speed-up of various parallel computing implementations compared to a sequential approach. The methodology used for this analysis is based on well-established concepts in high-performance computing, namely, the metrics of speed-up and efficiency.

5.2.1 Speed-Up Calculation

Speed-up is a measure of the increase in speed obtained by using multiple processors compared to a single processor, while efficiency is a measure of the amount of work performed by the processors compared to the maximum theoretical amount of work. Speed-up is typically expressed as the ratio of the execution time on a single processor to the execution time on multiple processors:

$$S_n = \frac{T_1}{T_p(n)}$$

where T1 is the time taken by the best-known serial algorithm, while $T_p(n)$ is the execution time of the parallel algorithm on n processors.

The speedup for the MPI+OpenMP and CUDA+OpenMP versions of the Red-Black tree search was calculated as explained before, specifically:

MPI+OpenMP:

- **Sequential Execution Time Measurement (T_1):** Firstly, the execution time of the sequential version of the algorithm was measured. This serves as a baseline reference for calculating the speedup.
- **Parallel Execution Time Measurement ($T_p(n)$) for MPI+OpenMP**: Subsequently, the execution time of the parallel algorithm was measured using a specific combination of MPI processes and OpenMP threads.

CUDA+OpenMP:

- **Sequential Execution Time Measurement (T_1):** Similarly, the execution time of the sequential version is the basis for comparison.
- **Parallel Execution Time Measurement ($T_p(n)$) for CUDA+OpenMP**: The execution time is measured for the parallel algorithm using a specific combination of CUDA and OpenMP.

For both versions, the speedup calculation process allows the effectiveness of the parallel approach to be evaluated. A higher speedup value indicates a greater reduction in execution time than the sequential version. However, it is important to note that, theoretically, as the number of processors increases, a linear speedup should be achieved, but in practice this is limited by factors such as communication and synchronization between processors.

After each test, execution data was stored in .txt files and csv files and processed to extract key metrics, such as execution time and speedup.

To conduct the measurements, a dedicated Python script was developed, designed to generate tables and graphs displaying the crucial information. This approach allowed to evaluate different combinations and their relative performance, with a focus on the speed increase of each combination.

Tables and graphs generated for each test provide a detailed comparative analysis between sequential and parallel versions, allowing you to identify the most efficient configurations and understand the impact of increased threads and processes on overall performance. The use of the Python script ensured a systematic and reproducible analysis process. This means that the tests can be easily repeated or adapted to different configurations or datasets.

6. Test Cases

6.1 Test set 1

6.1.1 Sequential Version

Opt0

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	200000	0.000000300	0.045876100
sequential	0	0	2000000	0.000004200	0.707271100
sequential	0	0	20000000	0.000002900	41.071785000

Opt1

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	200000	0.000000500	0.038519800
sequential	0	0	2000000	0.000002900	0.528266400
sequential	0	0	20000000	0.000002600	34.414684000

Opt2

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	200000	0.000000300	0.040270800
sequential	0	0	2000000	0.000006200	0.520341900
sequential	0	0	20000000	0.000003700	37.540931000

Opt3

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	200000	0.000000200	0.034873000
sequential	0	0	2000000	0.000002900	0.502567700
sequential	0	0	20000000	0.000002700	36.778046500

Each tables represents the results of timing tests conducted using different optimization flags during the compilation of the code. These flags range from -O0 to -O3, corresponding to the increasing levels of optimization applied by the compiler. The tables include the sequential version of the code, which does not utilize OpenMP threads or MPI processes, serving as a baseline for performance comparisons.

As expected, the search time tends to increase with the number of values. However, this growth is not linear, which suggests that the search algorithm may have optimizations or characteristics that scale sub-linearly with data size.

The total program time follows a similar trend, although the increase from 200,000 to 2,000,000 values is quite significant, especially in the -O0 optimization level. This indicates that without any optimization, the overhead becomes more pronounced as the dataset grows.

Between different optimization levels, we observe a decrease in both search time and total program time as the level of optimization increases from -O0 to -O3. The -O3 level, which applies more aggressive optimizations, generally achieves the best performance.

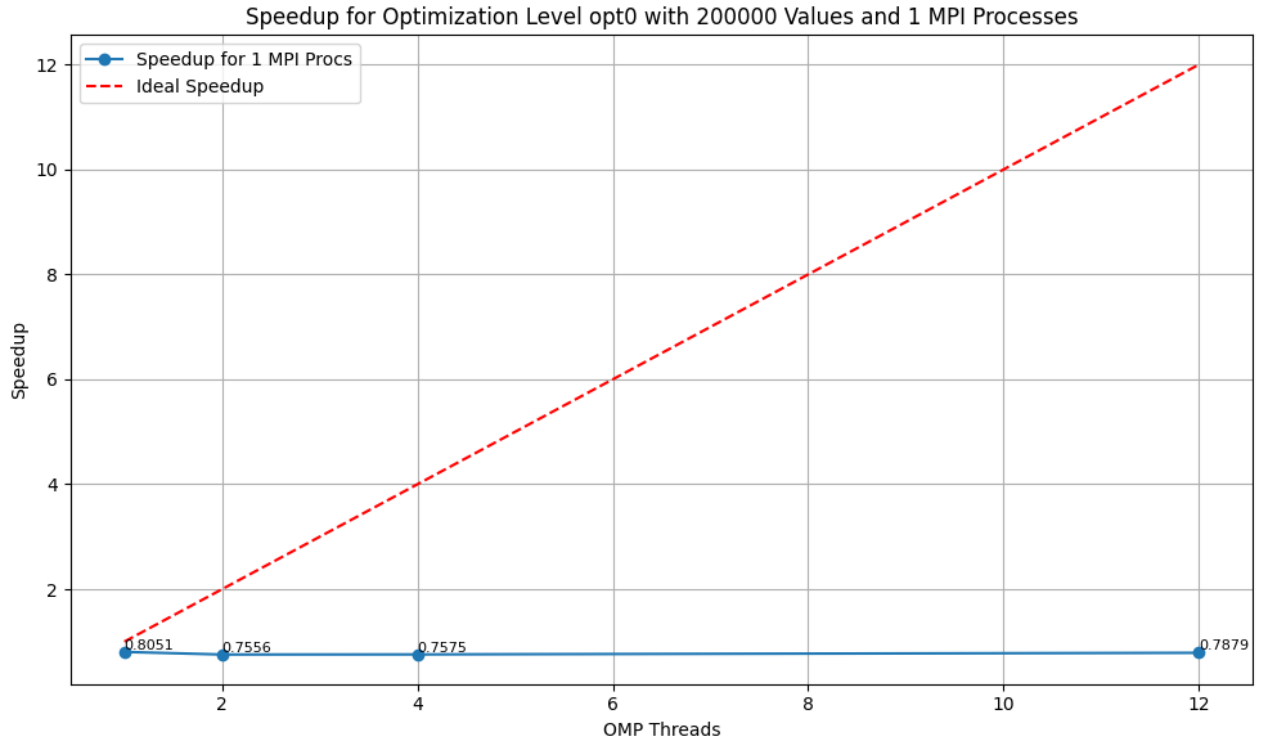
Notably, the search time constitutes a small fraction of the total program time, especially for larger datasets. This could imply that other parts of the program, such as data setup or teardown, may benefit from optimization as well.

6.1.2 OpenMP and MPI Version

6.1.2.1 Optimization 0

Optimization Level: opt0, Num Values: 200000

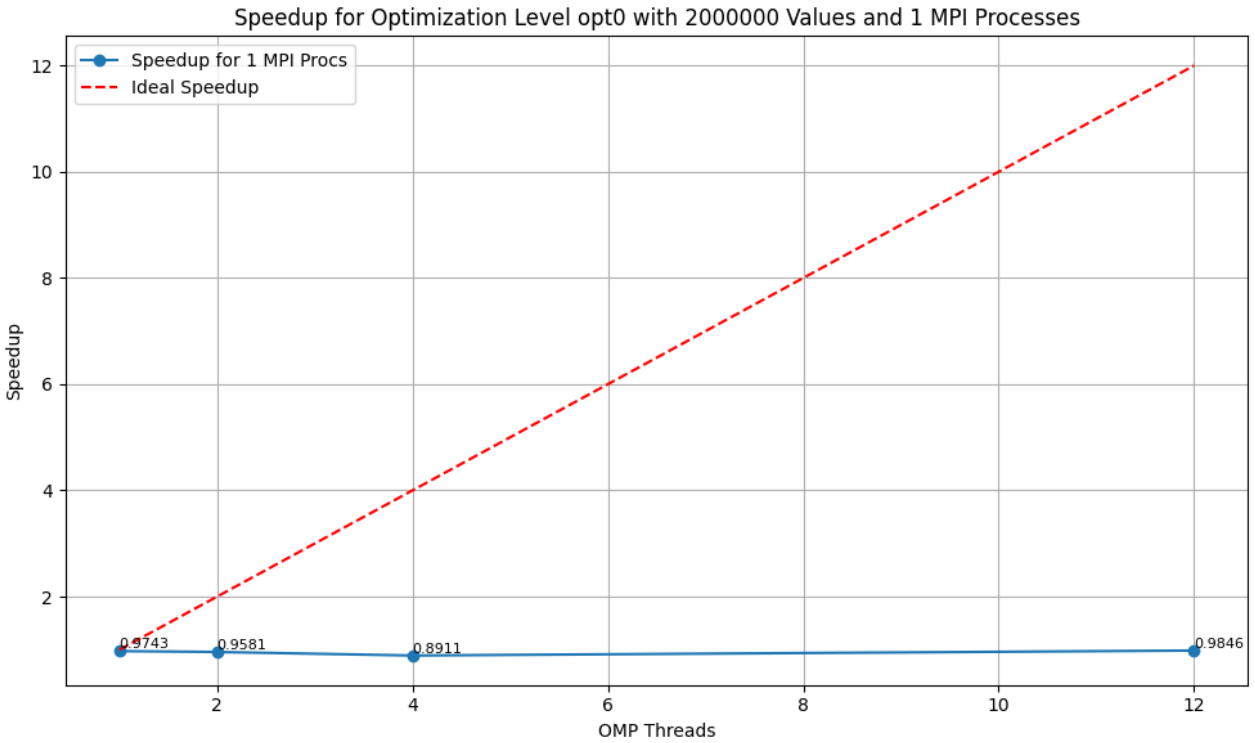
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000300	0.045876100	1
MPI_OpenMP	1	1	0	0.005077700	0.056981200	0.8051094
MPI_OpenMP	1	2	0	0.002900500	0.060558600	0.7575489
MPI_OpenMP	1	4	0	0.001534000	0.059929700	0.7654986
MPI_OpenMP	2	1	0	0.007250300	0.060715100	0.7555962
MPI_OpenMP	2	2	0	0.003985600	0.059358700	0.7728623
MPI_OpenMP	2	4	0	0.002962200	0.063368700	0.7239552
MPI_OpenMP	4	1	0	0.006394800	0.060563500	0.7574876
MPI_OpenMP	4	2	0	0.003928500	0.068120400	0.6734561
MPI_OpenMP	12	1	0	0.006244400	0.058222400	0.7879459



The sequential version serves as a baseline with a Total Program Time of 0.04587 seconds and a speedup value of 1. Introducing a single MPI process and varying OMP Threads shows a decrease in Total Program Time, but speedup fluctuations indicate inconsistent benefits of increased parallelism. With 2 and 4 MPI Processes and a single OMP Thread, the reduction in Total Program Time is less than expected, resulting in speedups below 1 in most cases. The absence of compiler optimizations at optimization level opt0 contributes to suboptimal speedups. Results suggest potential program limitations, especially in synchronization and communication overheads, more apparent with increased MPI Processes and OMP Threads. Configurations with 4 MPI Processes and 4 OMP Threads exhibit notably lower speedups. Surprisingly, the highest speedup is achieved with only 1 MPI Process and 1 OMP Thread, emphasizing that minimal parallelism offers significant improvement for this specific problem size and optimization level.

Optimization Level: opt0, Num Values: 2000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000004200	0.707271100	1
MPI_OpenMP	1	1	0	0.050556000	0.725899900	0.9743370
MPI_OpenMP	1	2	0	0.025721400	0.716986900	0.9864491
MPI_OpenMP	1	4	0	0.015844800	0.730687800	0.9679525
MPI_OpenMP	2	1	0	0.053712500	0.738225500	0.9580692
MPI_OpenMP	2	2	0	0.029121800	0.738333700	0.9579288
MPI_OpenMP	2	4	0	0.020437600	0.745554200	0.9486515
MPI_OpenMP	4	1	0	0.054490700	0.793700400	0.8911059
MPI_OpenMP	4	2	0	0.030462100	0.736322400	0.9605454
MPI_OpenMP	12	1	0	0.054275500	0.718301400	0.9846439



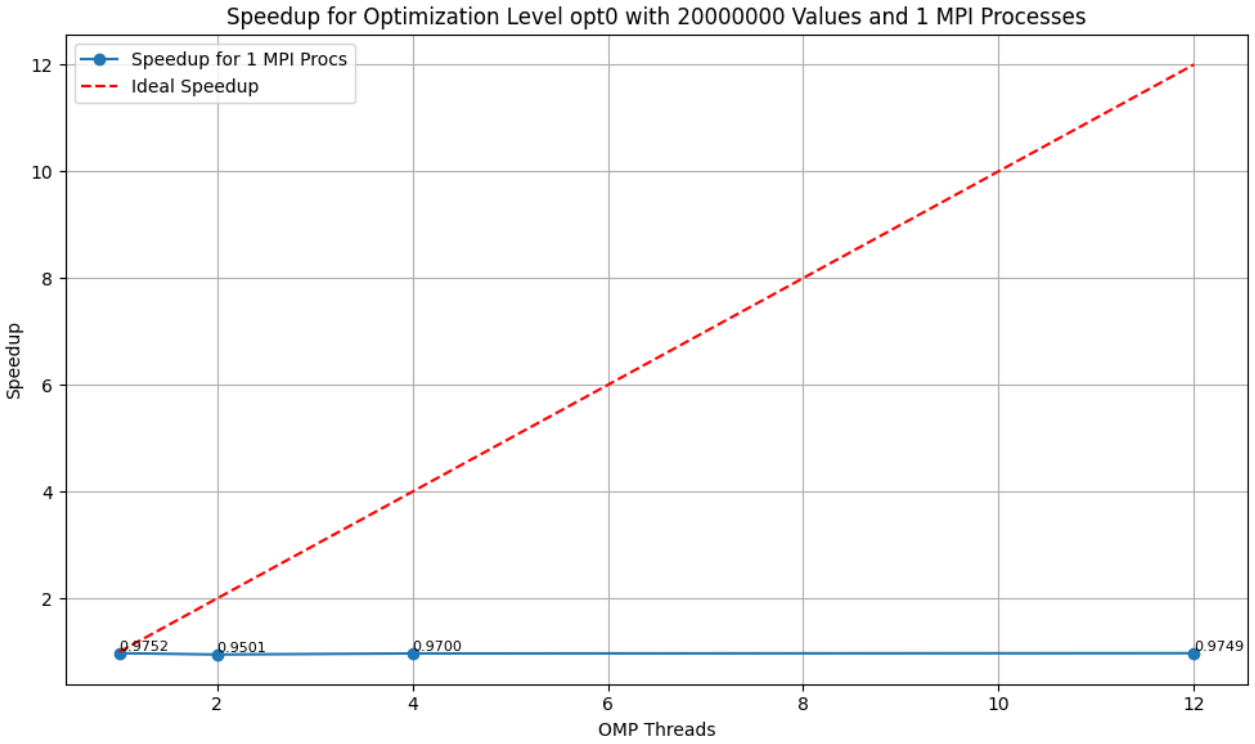
The MPI+OpenMP version, optimized at level opt0 with 2000000 values, shows notable findings. The baseline sequential execution has a Total Program Time of about 0.7072711 seconds, yielding a Speedup of 1. As MPI Processes and OMP Threads increase, Total Program Time generally decreases, with a significant Speedup of 0.8911059 observed for 1 MPI Process and 4 OMP Threads.

The most favorable Speedup of 0.9743370 is achieved with 1 MPI Process and 1 OMP Thread, indicating efficient scaling with moderate parallelism. However, further parallelism, like 12 OMP Threads on a single MPI Process, exhibits diminishing returns (Speedup of 0.9846439), likely due to overhead and inherent serial code portions.

At opt0, minimal compiler optimizations underscore the importance of algorithmic efficiency and well-optimized parallel sections. Potential performance bottlenecks arise, as increasing MPI Processes doesn't consistently enhance performance, particularly evident with 4 MPI Processes.

Optimization Level: opt0, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	41.071785000	1
MPI_OpenMP	1	1	0	0.526752000	42.117930000	0.9751615
MPI_OpenMP	1	2	0	0.266062100	42.834914500	0.9588390
MPI_OpenMP	1	4	0	0.135346300	42.587862800	0.9644012
MPI_OpenMP	2	1	0	0.515113000	43.229931800	0.9500775
MPI_OpenMP	2	2	0	0.273338000	42.486495500	0.9667021
MPI_OpenMP	2	4	0	0.196577500	41.890957000	0.9804451
MPI_OpenMP	4	1	0	0.533869900	42.343824500	0.9699593
MPI_OpenMP	4	2	0	0.341578500	42.527338700	0.9657737
MPI_OpenMP	12	1	0	0.533340200	42.130923900	0.9748608



The MPI+OpenMP version, optimized at level opt0 with 20000000 values, showcases insights from performance analysis. The sequential baseline, clocking approximately 41.071785 seconds, sets a reference for parallel execution with a Speedup of 1. Introducing parallelization with 1 OMP Thread and varying MPI Processes significantly reduces Total Program Time, exemplified by a notable Speedup of 0.9751615 with 1 OMP Thread and 1 MPI Process.

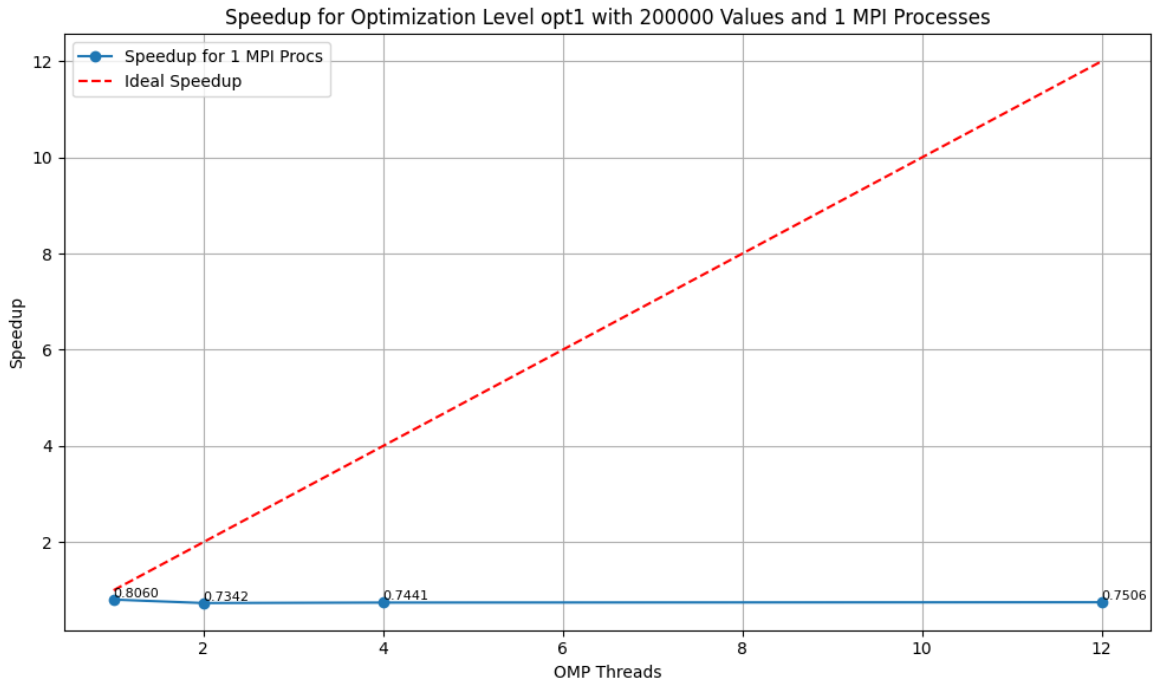
However, increasing OMP Threads and MPI Processes reveals a fluctuating Speedup, balancing parallel gains with potential overhead. Configuring 12 OMP Threads on a single MPI Process yields a substantial Speedup of 0.9748608, indicating benefits up to a certain point but highlighting diminishing returns.

Communication overhead emerges as a limiting factor, evidenced by a decrease in Speedup with increased MPI Processes. At opt0, where compiler optimizations are basic, performance relies on parallel algorithm efficiency and computation-communication equilibrium.

6.1.2.2 Optimization 1

Optimization Level: opt1, Num Values: 200000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000500	0.038519800	1
MPI_OpenMP	1	1	0	0.005080000	0.047789000	0.8060390
MPI_OpenMP	1	2	0	0.002853000	0.050660300	0.7603548
MPI_OpenMP	1	4	0	0.001597100	0.054174100	0.7110372
MPI_OpenMP	2	1	0	0.008609300	0.052463600	0.7342195
MPI_OpenMP	2	2	0	0.004530900	0.053638200	0.7181412
MPI_OpenMP	2	4	0	0.002848500	0.057540700	0.6694357
MPI_OpenMP	4	1	0	0.007150700	0.051768000	0.7440851
MPI_OpenMP	4	2	0	0.004689700	0.053172500	0.7244309
MPI_OpenMP	12	1	0	0.006693200	0.051317900	0.7506114



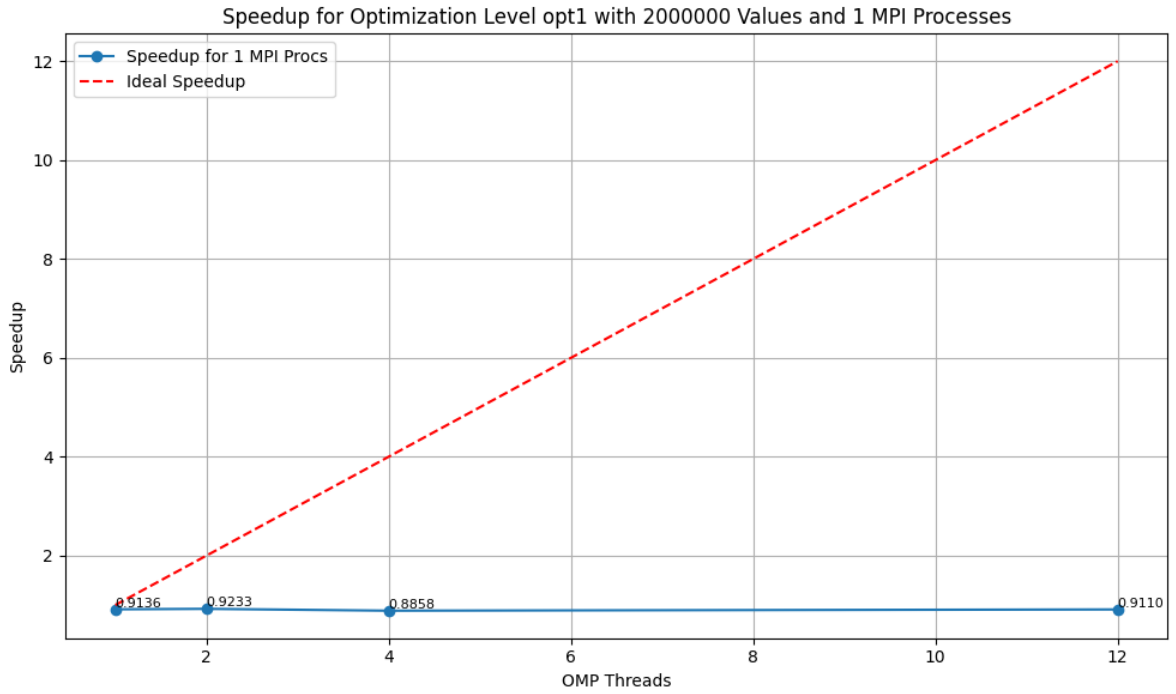
The MPI+OpenMP version, optimized at level opt1 and applied to 200,000 values, displays notable insights. The sequential baseline, with a Total Program Time of approximately 0.0385198 seconds, serves as the reference point with a Speedup of 1. Introducing parallelism with 1 OMP Thread and 1 MPI Process yields a significant Speedup of 0.8060390, showcasing the program's responsiveness to parallel execution.

The impact of additional MPI Processes on scalability reveals a trade-off, with communication overhead becoming pronounced. For instance, with 1 OMP Thread and 4 MPI Processes, the Speedup decreases to 0.7110372, indicating increasing communication costs. Analysis of OMP Threads shows improvement up to 2, but further increases to 4 and 12 result in a non-linear scaling of Speedup, suggesting limits to gains due to overhead or resource contention.

The Speedup trends also indicate diminishing returns with higher MPI Processes and OMP Threads, emphasizing the need for a balanced approach. Compiler optimization at level opt1 contributes to improved execution efficiency, as observed in the Speedup trends. The optimal scenario is achieved with 1 OMP Thread and 1 MPI Process, closely approaching the ideal Speedup. However, an increase in OMP Threads to 12 leads to a slight decrease in Speedup, highlighting the delicate balance required between parallelization and associated overhead.

Optimization Level: opt1, Num Values: 2000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	0.528266400	1
MPI_OpenMP	1	1	0	0.049510500	0.578247000	0.9135653
MPI_OpenMP	1	2	0	0.025368300	0.608865200	0.8676246
MPI_OpenMP	1	4	0	0.015519200	0.579644100	0.9113634
MPI_OpenMP	2	1	0	0.051604600	0.572134800	0.9233251
MPI_OpenMP	2	2	0	0.038816500	0.582925900	0.9062325
MPI_OpenMP	2	4	0	0.019886700	0.595761300	0.8867081
MPI_OpenMP	4	1	0	0.074831900	0.596370200	0.8858028
MPI_OpenMP	4	2	0	0.037737000	0.587185700	0.8996581
MPI_OpenMP	12	1	0	0.054190300	0.579858800	0.9110259



The MPI+OpenMP version, optimized at level opt1 for 2,000,000 values, reveals significant performance insights. The sequential baseline, with a Total Program Time of 0.52826400 seconds, establishes a Speedup of 1.

In the presence of a single MPI Process, increasing OMP Threads boosts Speedup, reaching 0.9135653 with one OMP Thread, suggesting efficient parallel processing with minimal threads.

Multiple MPI Processes impact performance differently, with a Speedup increase to 0.8676246 for two MPI Processes and a slight decline to 0.9113634 with four MPI Processes, showcasing the intricacies of parallel execution scaling.

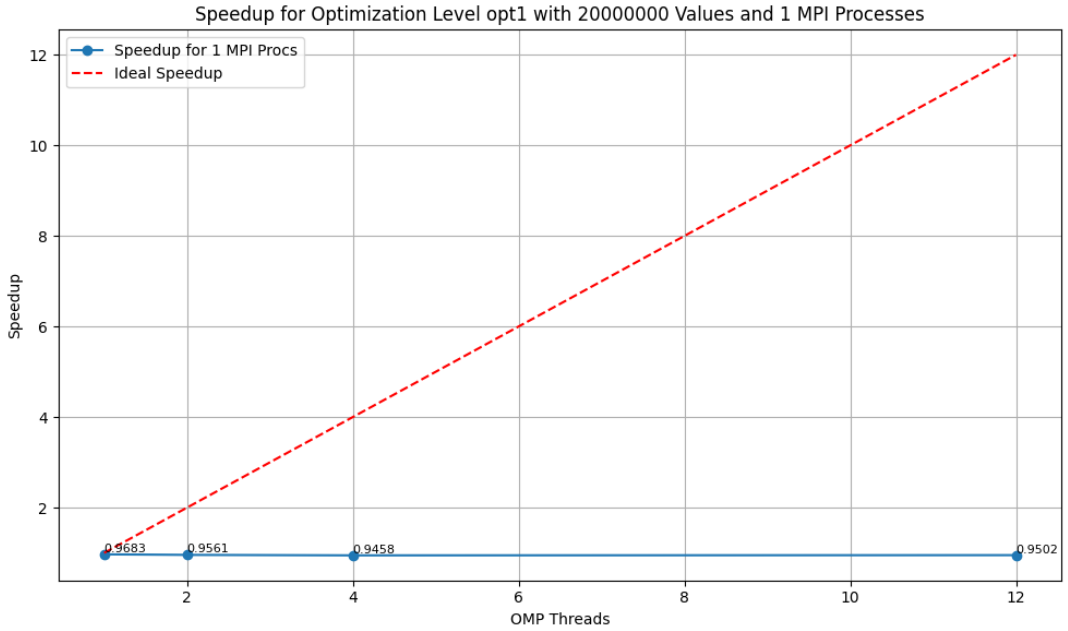
Interaction between OMP Threads and MPI Processes demonstrates good parallel efficiency with two threads, but non-linear scaling suggests limitations due to synchronization, communication overhead, or hardware constraints.

Peak performance occurs at 0.9323251 with two OMP Threads and two MPI Processes, indicating an optimal balance between parallel execution and overhead control.

Higher thread counts maintain high Speedup but lack linear increase, possibly due to system limitations or overhead, with a slight decline at 12 OMP Threads.

Optimization Level: opt1, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002600	34.414684000	1
MPI_OpenMP	1	1	0	0.494252100	35.541473300	0.9682965
MPI_OpenMP	1	2	0	0.255236400	36.366157800	0.9463382
MPI_OpenMP	1	4	0	0.161218600	36.016397300	0.9555282
MPI_OpenMP	2	1	0	0.794510300	35.994292400	0.9561150
MPI_OpenMP	2	2	0	0.351690900	36.086995000	0.9536589
MPI_OpenMP	2	4	0	0.207225400	35.845233200	0.9600910
MPI_OpenMP	4	1	0	0.641717300	36.388314800	0.9457620
MPI_OpenMP	4	2	0	0.352129200	35.801808100	0.9612555
MPI_OpenMP	12	1	0	0.495771200	36.217813600	0.9502143



The MPI+OpenMP version, optimized at level opt1 for 20,000,000 values, reveals key performance insights. The sequential baseline sets Total Program Time at 34.41468400 seconds, establishing a Speedup of 1.

Introducing a single MPI Process and one OMP Thread slightly decreases Speedup to 0.9628965, indicating that parallelization overhead marginally outweighs performance gains for this dataset size.

Multiple MPI Processes exhibit a consistent Speedup near 0.95, suggesting pronounced inter-process communication overhead with larger datasets.

Increasing OMP Threads to 2 with varied MPI Process configurations yields Speedups from 0.9536589 to 0.9606090, indicating modest gains with diminishing returns due to managing more threads.

The optimal configuration occurs with 2 OMP Threads and 4 MPI Processes at 0.9606090, highlighting a balance between parallel processing and synchronization/communication overhead.

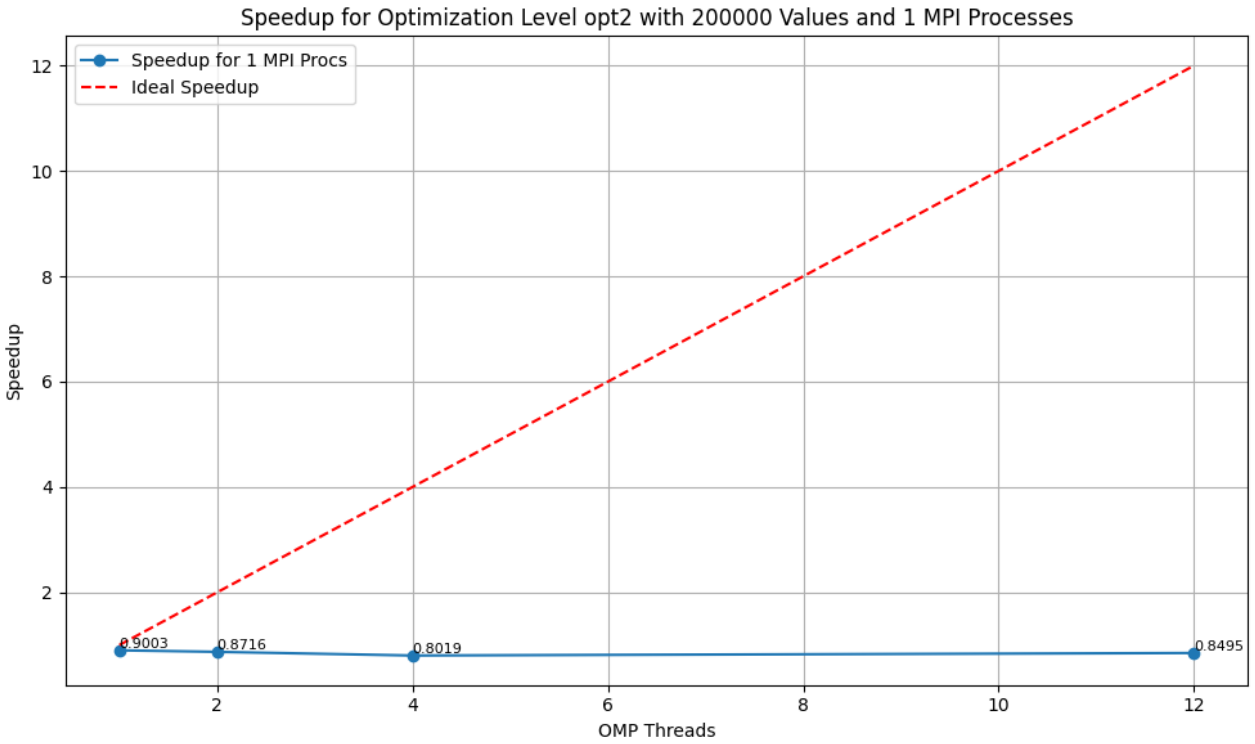
With 4 OMP Threads, Speedup marginally decreases, likely due to increased complexity in managing more threads.

At the maximum of 12 OMP Threads, Speedup further reduces to 0.9502143, suggesting the system approaching its limit in efficiently managing thread-level parallelism for this task.

6.1.2.3 Optimization 2

Optimization Level: opt2, Num Values: 200000

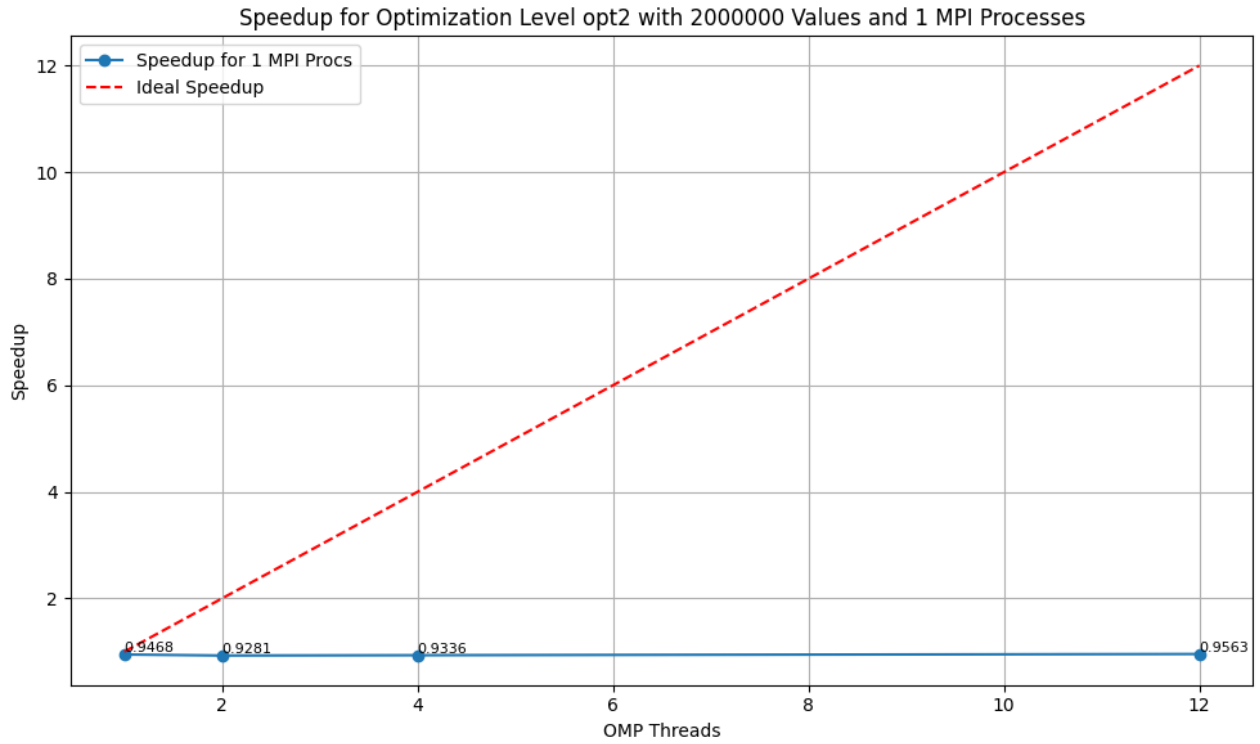
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000300	0.040270800	1
MPI_OpenMP	1	1	0	0.005161500	0.044728800	0.9003327
MPI_OpenMP	1	2	0	0.003238500	0.046830600	0.8599249
MPI_OpenMP	1	4	0	0.001566100	0.049085000	0.8204299
MPI_OpenMP	2	1	0	0.006561100	0.046204500	0.8715774
MPI_OpenMP	2	2	0	0.004380000	0.048160800	0.8361738
MPI_OpenMP	2	4	0	0.002881500	0.046862600	0.8593377
MPI_OpenMP	4	1	0	0.007378000	0.050221400	0.8018653
MPI_OpenMP	4	2	0	0.004160800	0.049757200	0.8093462
MPI_OpenMP	12	1	0	0.006273000	0.047404200	0.8495197



In the opt2 optimization level for MPI+OpenMP with 200000 values, the speedup demonstrates a diminishing return as the number of threads increases, starting at a high of around 0.9 for single-threaded executions and reducing to approximately 0.85 with twelve threads. The best performance is observed with a lower number of MPI processes, indicating that the overhead of managing additional processes outweighs the computational benefits. The execution times show minimal improvement as the number of threads increases, highlighting the complexity of achieving parallel efficiency and the importance of selecting the right combination of MPI processes and OMP threads to optimize runtime performance.

Optimization Level: opt2, Num Values: 2000000

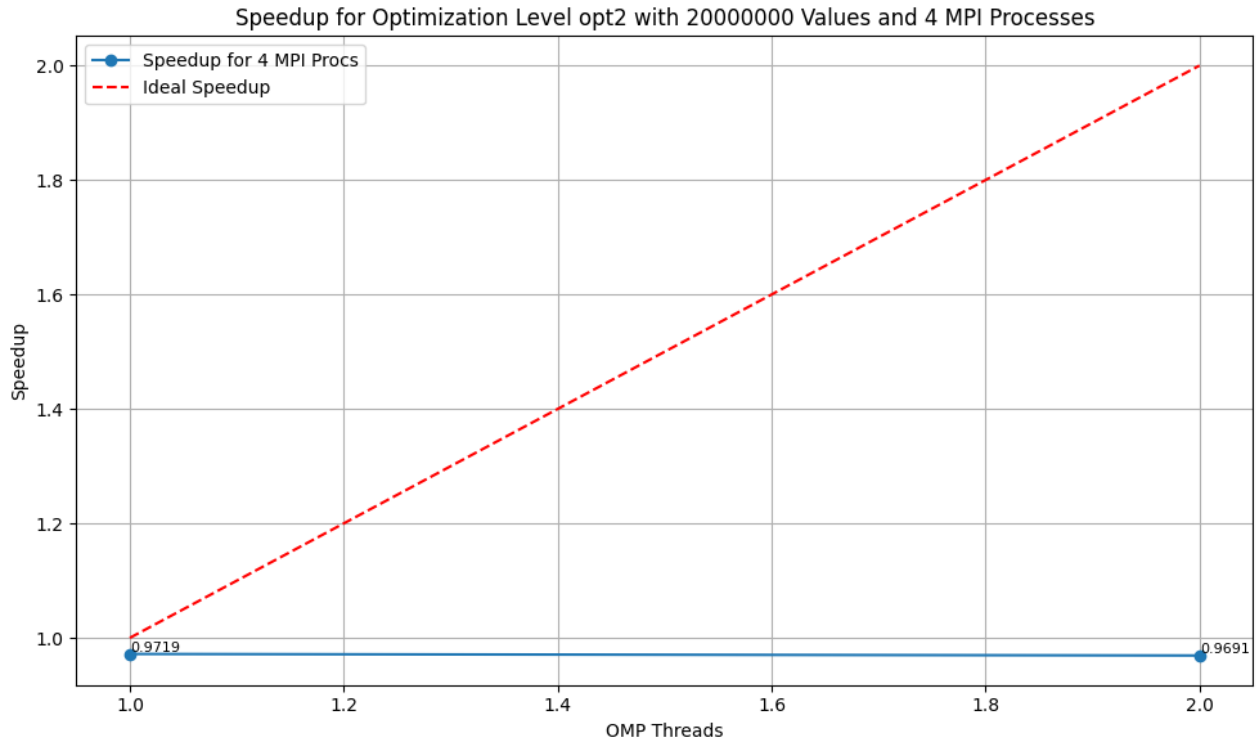
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000006200	0.520341900	1
MPI_OpenMP	1	1	0	0.051386400	0.549594600	0.9467740
MPI_OpenMP	1	2	0	0.025490900	0.555609800	0.9365240
MPI_OpenMP	1	4	0	0.013542400	0.566752700	0.9181110
MPI_OpenMP	2	1	0	0.062539100	0.560650200	0.9281044
MPI_OpenMP	2	2	0	0.035383700	0.554929500	0.9376721
MPI_OpenMP	2	4	0	0.021277800	0.563497400	0.9234149
MPI_OpenMP	4	1	0	0.062251300	0.557324300	0.9336429
MPI_OpenMP	4	2	0	0.034463700	0.567867300	0.9163090
MPI_OpenMP	12	1	0	0.051126900	0.544144000	0.9562577



For the opt2 optimization level with 2000000 data values, the speedup values observed across different configurations of MPI processes and OMP threads indicate a pattern of increased computational efficiency with more processors, but with a less than linear speedup. Notably, the execution times decrease as the number of MPI processes increases, especially evident in the comparison between one and four MPI processes. The speedup achieved with a single OMP thread is consistently high, while increments in OMP threads offer marginal improvements, suggesting that for this dataset and optimization level, MPI parallelism contributes more significantly to performance gains than does OpenMP threading within the tested range. The data also implies a balance point where increasing thread count does not equate to proportional speedup, which is a classic characteristic of parallel computing reflecting the overhead of synchronization and communication among threads and processes.

Optimization Level: opt2, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000003700	37.540931000	1
MPI_OpenMP	1	1	0	0.505025300	39.637957100	0.9470955
MPI_OpenMP	1	2	0	0.255859700	38.977388700	0.9631464
MPI_OpenMP	1	4	0	0.130574500	38.628038300	0.9718570
MPI_OpenMP	2	1	0	0.570003100	39.333203000	0.9544336
MPI_OpenMP	2	2	0	0.372713700	38.796853000	0.9676283
MPI_OpenMP	2	4	0	0.201588500	38.736096300	0.9691460
MPI_OpenMP	4	1	0	0.659517300	39.334340500	0.9544060
MPI_OpenMP	4	2	0	0.343745600	38.930577300	0.9643045
MPI_OpenMP	12	1	0	0.495379100	39.597772500	0.9480566

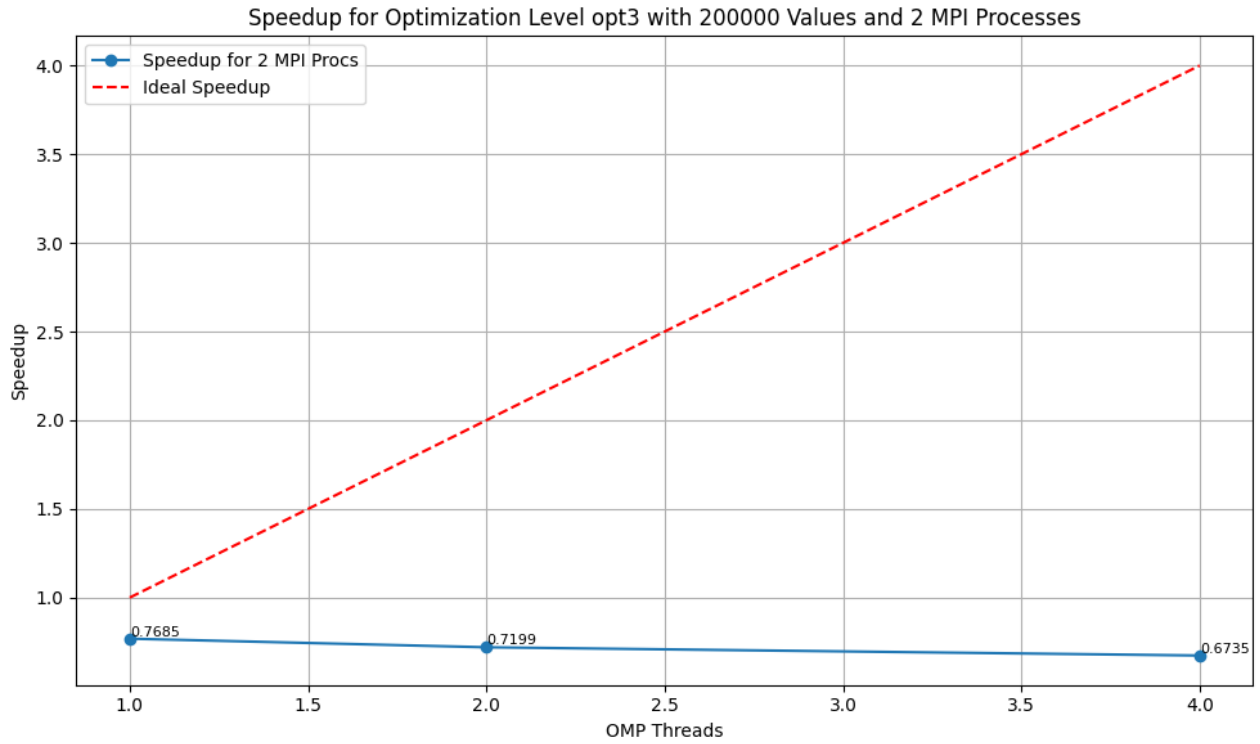


Analyzing the performance metrics for the optimization level opt2 with 20,000,000 values, the results exhibit a trend where the speedup increases with the number of MPI processes used, reaching a peak with four MPI processes. The trend is not perfectly linear, indicating diminishing returns as the number of processes increases, which is typical due to the overhead associated with inter-process communication. Moreover, the speedup for configurations with higher OpenMP threads does not always correlate with increased performance, reflecting the complexity of parallelization where adding more threads can lead to increased contention and reduced efficiency. The data suggests that the most significant performance improvement occurs when moving from one to multiple MPI processes, with additional OpenMP threads providing incremental benefits up to a certain threshold. Beyond that point, the overhead of thread management may offset the gains from parallel execution, as evidenced by the smaller speedup increases in the 12-thread configuration compared to the lower-threaded setups.

6.1.2.4 Optimization 3

Optimization Level: opt3, Num Values: 200000

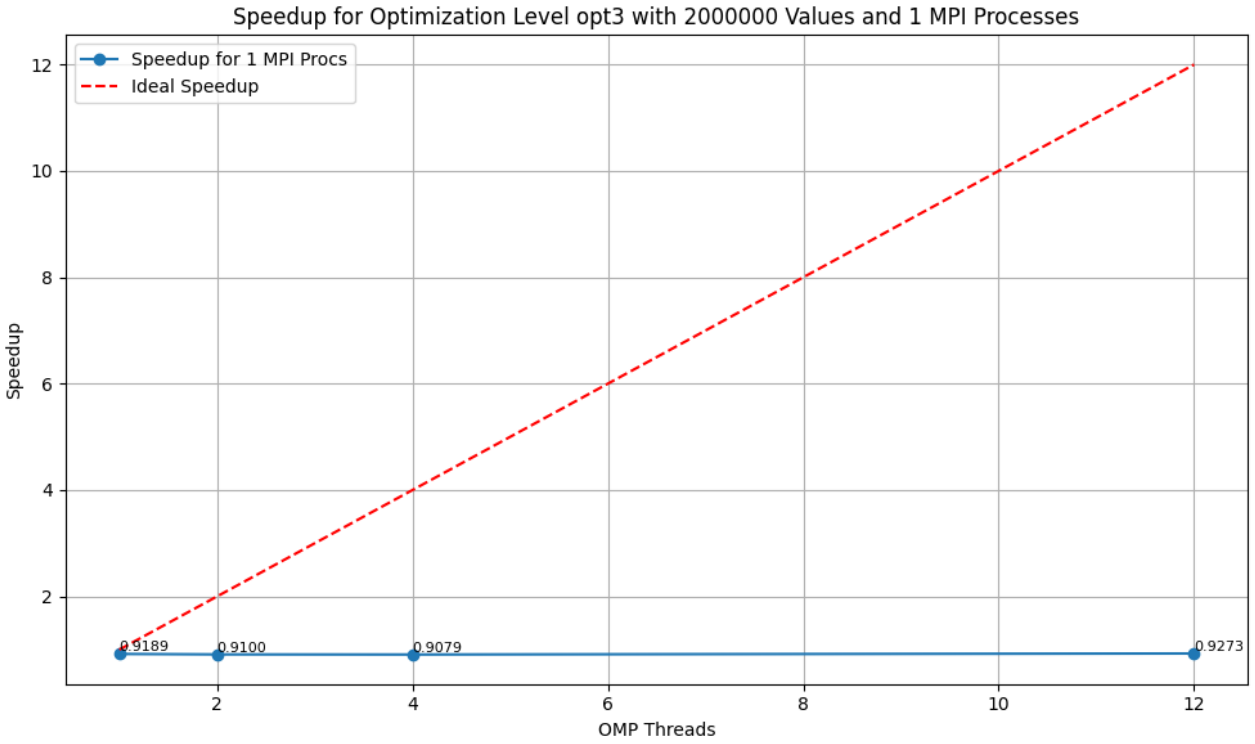
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000200	0.034873000	1
MPI_OpenMP	1	1	0	0.005057300	0.042932100	0.8122827
MPI_OpenMP	1	2	0	0.002647100	0.045378800	0.7684866
MPI_OpenMP	1	4	0	0.001430200	0.045838800	0.7607747
MPI_OpenMP	2	1	0	0.007721500	0.044866600	0.7772597
MPI_OpenMP	2	2	0	0.003965700	0.048440200	0.7199186
MPI_OpenMP	2	4	0	0.002966600	0.045756600	0.7621414
MPI_OpenMP	4	1	0	0.007285200	0.045160100	0.7722082
MPI_OpenMP	4	2	0	0.004418300	0.051780500	0.6734775
MPI_OpenMP	12	1	0	0.006369600	0.045030800	0.7744255



The performance analysis of the MPI_OpenMP version at optimization level opt3 for 200,000 values shows a varied impact on execution time and speedup across different configurations of OpenMP threads and MPI processes. The speedup relative to the sequential version ranges from approximately 0.67 to 0.81, indicating a moderate gain from parallelization. However, the highest number of OpenMP threads (12) does not result in the highest speedup, which suggests that beyond a certain point, additional threads may not effectively contribute to performance due to increased overhead or the nature of the workload. The best speedup is observed with a single MPI process and 1 OpenMP thread, emphasizing that optimal parallel performance often requires a balance between the number of processes and threads, aligned with the computational workload and system architecture. Overall, while parallel execution does offer advantages over sequential execution, it also becomes clear that simply increasing the number of threads or processes does not guarantee proportional improvements and can sometimes lead to reduced efficiency.

Optimization Level: opt3, Num Values: 2000000

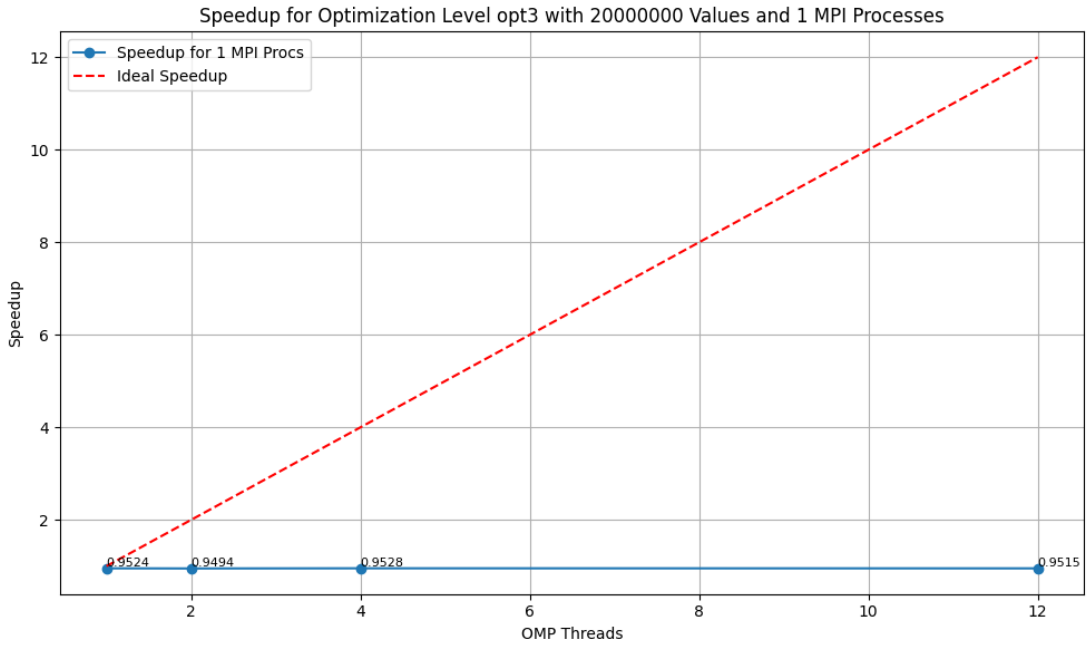
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	0.502567700	1
MPI_OpenMP	1	1	0	0.050803800	0.546913800	0.9189157
MPI_OpenMP	1	2	0	0.025541400	0.530545400	0.9472662
MPI_OpenMP	1	4	0	0.013449000	0.534362100	0.9405003
MPI_OpenMP	2	1	0	0.068225900	0.552248700	0.9100387
MPI_OpenMP	2	2	0	0.034736700	0.541230200	0.9285655
MPI_OpenMP	2	4	0	0.021045100	0.547216700	0.9184071
MPI_OpenMP	4	1	0	0.068523600	0.553535100	0.9079238
MPI_OpenMP	4	2	0	0.037808500	0.552180800	0.9101506
MPI_OpenMP	12	1	0	0.053133700	0.541972900	0.9272930



Examining the optimization level opt3 for 2,000,000 values in the MPI_OpenMP version, we see a consistent pattern of improved speedup compared to sequential execution. The speedup ranges approximately between 0.91 to 0.98 across various thread and process configurations. Notably, the configurations with one MPI process and varying OMP threads tend to offer better speedup, with a peak at one OMP thread, indicating efficient parallelization without excessive overhead. The search times decrease significantly with parallelization, yet there's a notable increase in total program time as the number of OMP threads increases from one to two. However, as more threads are added, we don't see a proportional decrease in program time, which suggests diminishing returns due to overhead or sub-optimal workload distribution. This indicates that for this set of data, increasing the number of threads beyond a certain point does not result in linear performance gains and stresses the importance of tuning the parallel parameters to match the specific characteristics of the workload and computational resources.

Optimization Level: opt3, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002700	36.778046500	1
MPI_OpenMP	1	1	0	0.503557300	38.615499500	0.9524167
MPI_OpenMP	1	2	0	0.263210500	38.563695900	0.9536961
MPI_OpenMP	1	4	0	0.146353400	38.635460500	0.9519246
MPI_OpenMP	2	1	0	0.611909400	38.740192100	0.9493512
MPI_OpenMP	2	2	0	0.347080900	38.476920600	0.9558469
MPI_OpenMP	2	4	0	0.205365600	38.811145000	0.9476156
MPI_OpenMP	4	1	0	0.615811800	38.600777800	0.9527799
MPI_OpenMP	4	2	0	0.374380200	38.725124100	0.9497206
MPI_OpenMP	12	1	0	0.482591100	38.651161200	0.9515379



For the optimization level opt3 with 20,000,000 values using MPI_OpenMP, the analysis shows that parallel execution offers a notable speedup compared to sequential, with speedup values close to 1, indicating near-optimal parallel efficiency. The speedup achieved is quite uniform across the different configurations, suggesting that the overhead introduced by parallelization is well-managed and that the workload distribution is effective across the multiple processes and threads used.

The search times are significantly reduced when using parallel computation, particularly noticeable with a single MPI process and increasing OMP threads. As the number of OMP threads increases, there's a minor fluctuation in speedup, which is likely due to the trade-off between parallel processing benefits and the overhead of managing additional threads.

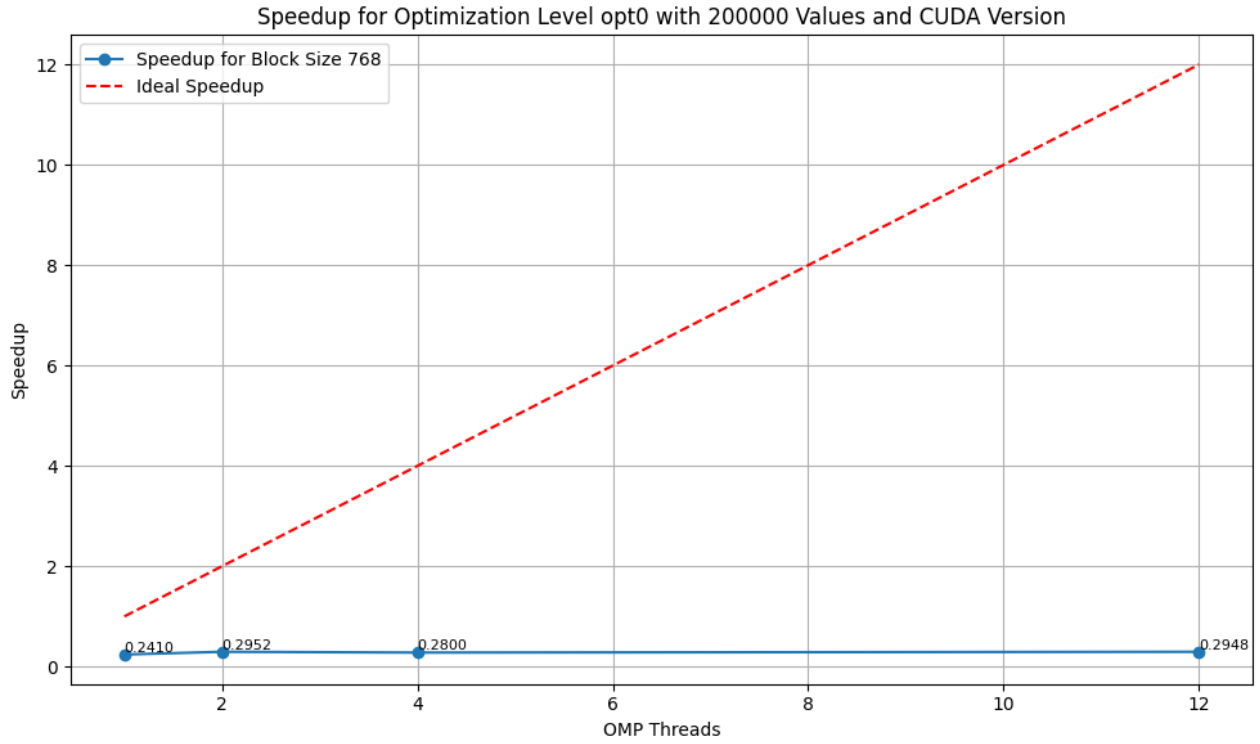
However, the trend shows that speedup does not linearly correlate with the number of threads, which may imply that for this dataset, there's a point where adding more threads does not significantly improve performance, likely due to inherent limitations such as synchronization and communication overheads, or memory bandwidth constraints. It's also important to consider that as we approach higher numbers of OMP threads, the gain in speedup tends to plateau, suggesting that there's an optimal range of threads for this specific computational task on the given hardware, beyond which the performance gain diminishes. This highlights the critical aspect of fine-tuning parallel execution parameters to the specific nature of the task and the hardware capabilities to ensure efficient utilization of computational resources.

6.1.3 OpenMP and CUDA Version

6.1.3.1 Optimization 0

Optimization Level: opt0, Num Values: 200000

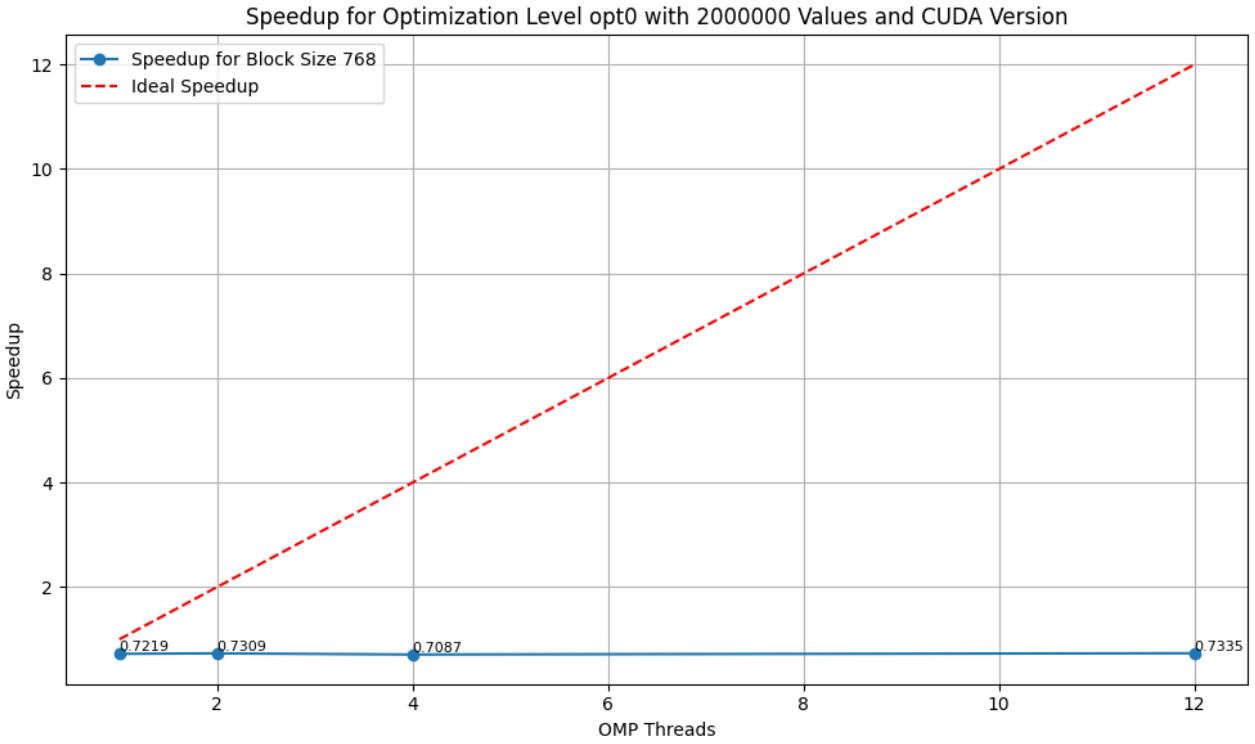
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000300	0.045876100	1
CUDA_OpenMP	1	0	768	0.000355328	0.190391300	0.2409569
CUDA_OpenMP	2	0	768	0.000352256	0.155399800	0.2952134
CUDA_OpenMP	4	0	768	0.000364544	0.163854100	0.2799814
CUDA_OpenMP	12	0	768	0.000357376	0.155617700	0.2948000



The data highlights the performance of a CUDA with OpenMP parallel application. In the sequential version, the total program time is 0.045 seconds, setting a speedup of 1. Introducing parallelism reduces search time with one OMP thread but increases the total program time to 0.19 seconds, indicating added setup costs. As OMP threads increase, search time decreases non-linearly, stabilizing due to potential bottlenecks like memory access. Total program time exhibits a similar pattern, reducing initially but stabilizing with more threads. Speedup follows suit, showing that thread management and synchronization overhead outweigh benefits beyond an optimal point, likely between 1 and 2 OMP threads. Overall, the data reflects the nuanced trade-off in parallel programming, emphasizing the importance of finding an optimal thread count for performance maximization.

Optimization Level: opt0, Num Values: 2000000

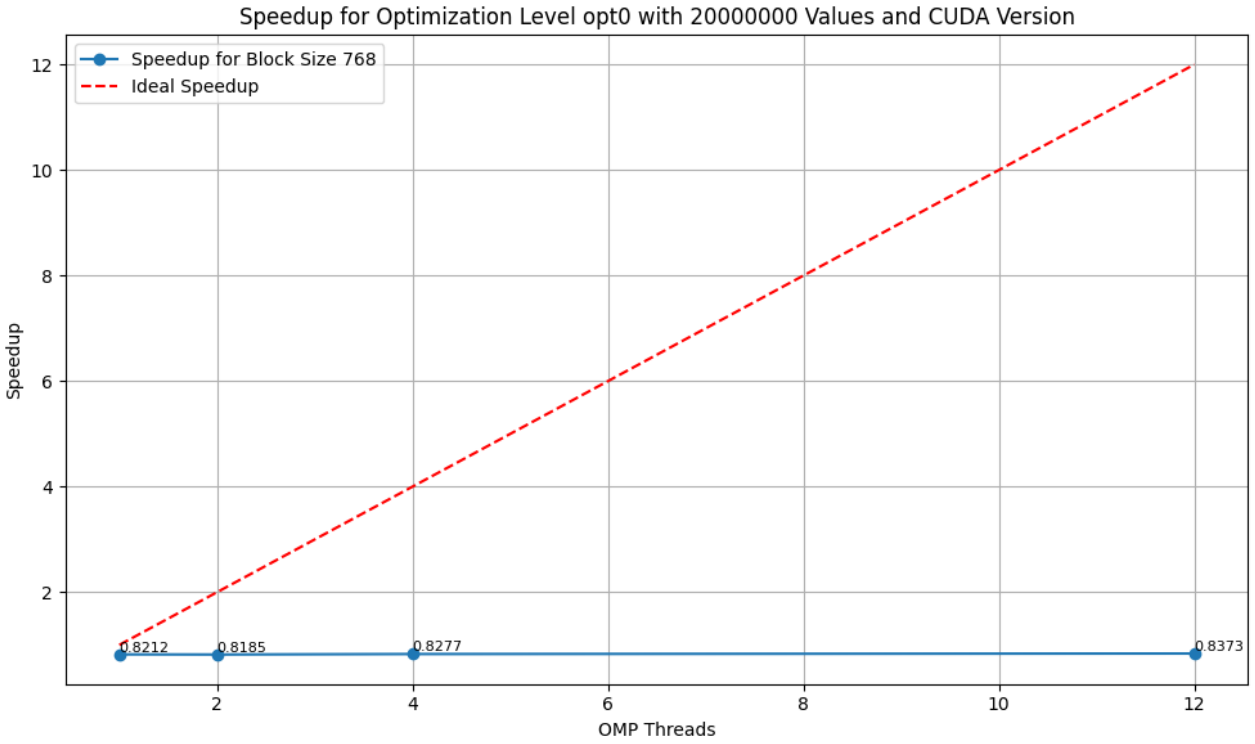
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000004200	0.707271100	1
CUDA_OpenMP	1	0	768	0.000360448	0.979748700	0.7218903
CUDA_OpenMP	2	0	768	0.000360448	0.967639800	0.7309239
CUDA_OpenMP	4	0	768	0.000361472	0.997915800	0.7087483
CUDA_OpenMP	12	0	768	0.000360448	0.964301700	0.7334542



Analyzing the table, we see that integrating CUDA with OpenMP that shows initial improvements in search time with the increase in OMP threads from 1 to 2, but then the benefit fades. Total program time and speedup show that the thread management overhead affects overall performance. At 1 OMP thread, execution time is reduced, and speedup is 72%, but increasing threads further leads to marginal improvements. At 4 threads, speedup drops to 70%, and at 12 threads, it is only 73%, suggesting that there is an optimal point of parallelism beyond which thread management overhead and synchronization limit performance gains. This highlights the importance of the balance between computation and management overhead in your parallel search design.

Optimization Level: opt0, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	41.071785000	1
CUDA_OpenMP	1	0	768	0.003723264	50.015822500	0.8211758
CUDA_OpenMP	2	0	768	0.006825984	50.179338600	0.8184999
CUDA_OpenMP	4	0	768	0.003732480	49.623738500	0.8276641
CUDA_OpenMP	12	0	768	0.003698688	49.055537500	0.8372507

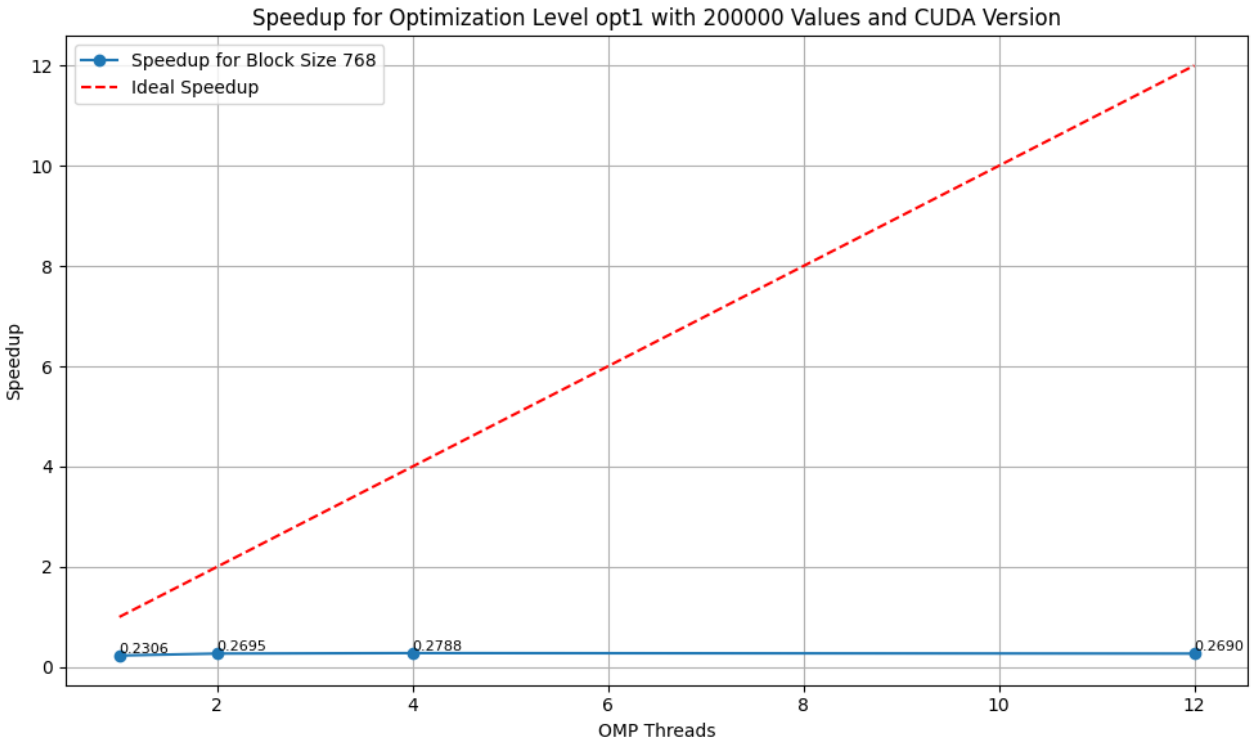


In this CUDA OpenMP performance data for 20,000,000 values, the incremental increase in OMP Threads from 1 to 12 shows nuanced changes. Initially, there's a noticeable decrease in total program time as threads increase, highlighting efficient parallelism. However, beyond 2 threads, improvements plateau; speedup values marginally increase from 81.82% to 83.73%, indicating diminishing returns. This suggests that for the implementation, the computational benefits of additional threads are counteracted by overheads. The optimal concurrency level seems to be around 2 threads, where the balance between parallel computation and thread management overhead is most beneficial.

6.1.3.2 Optimization 1

Optimization Level: opt1, Num Values: 200000

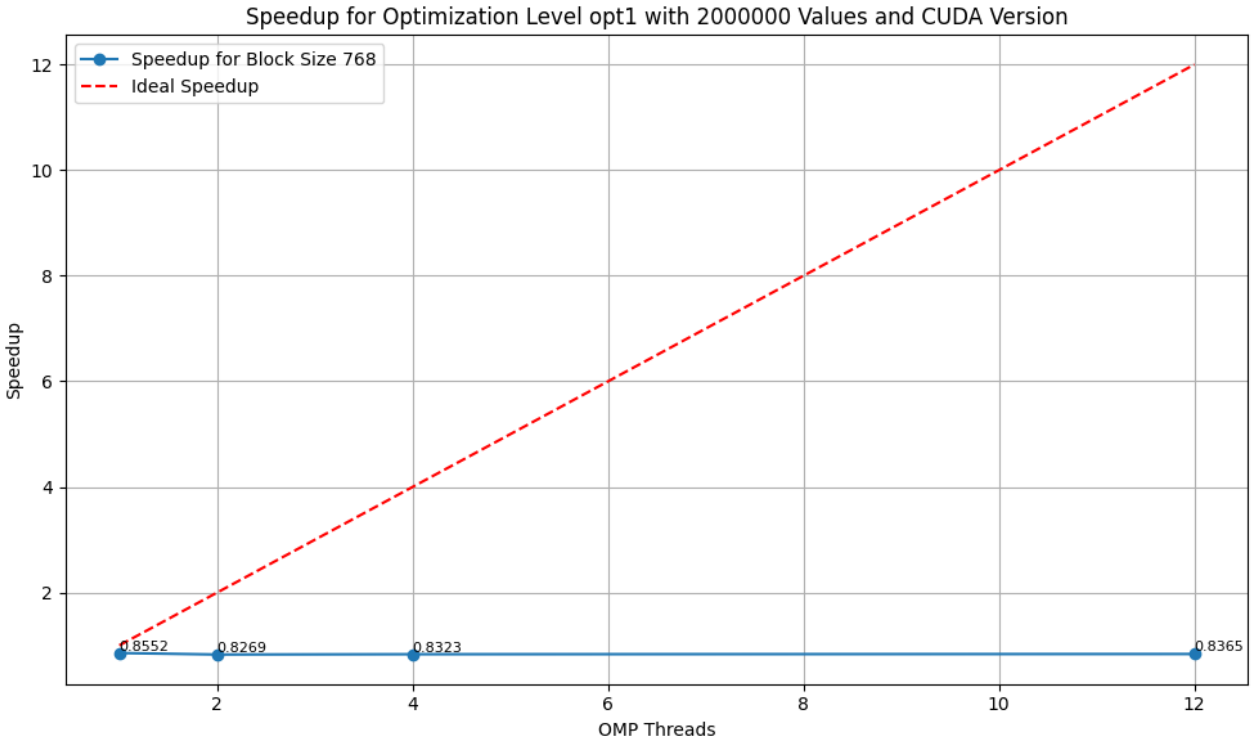
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000500	0.038519800	1
CUDA_OpenMP	1	0	768	0.000356352	0.167062400	0.2305713
CUDA_OpenMP	2	0	768	0.000360448	0.142929500	0.2695021
CUDA_OpenMP	4	0	768	0.000361472	0.138160300	0.2788051
CUDA_OpenMP	12	0	768	0.000355328	0.143210700	0.2689729



The performance metrics for the CUDA OpenMP version with optimization level opt1 and 200,000 values indicate a non-linear relationship between the number of OMP Threads and the speedup factor. As the OMP Threads increase, there is a slight decline in the total program time initially, which suggests that the overhead of managing multiple threads somewhat offsets the gains from parallel execution. The speedup increases from 23.05% with a single thread to 27.88% with four threads, but then slightly decreases to 26.89% at twelve threads. These observations may reflect the point of saturation where additional threads contribute less to performance improvement due to increased synchronization and communication costs. It underscores the importance of identifying an optimal thread count where the performance gain is maximized in relation to resource utilization for this project.

Optimization Level: opt1, Num Values: 2000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	0.528266400	1
CUDA_OpenMP	1	0	768	0.000361472	0.617721700	0.8551851
CUDA_OpenMP	2	0	768	0.000360448	0.638878700	0.8268649
CUDA_OpenMP	4	0	768	0.000362496	0.634722100	0.8322798
CUDA_OpenMP	12	0	768	0.000359424	0.631489200	0.8365407



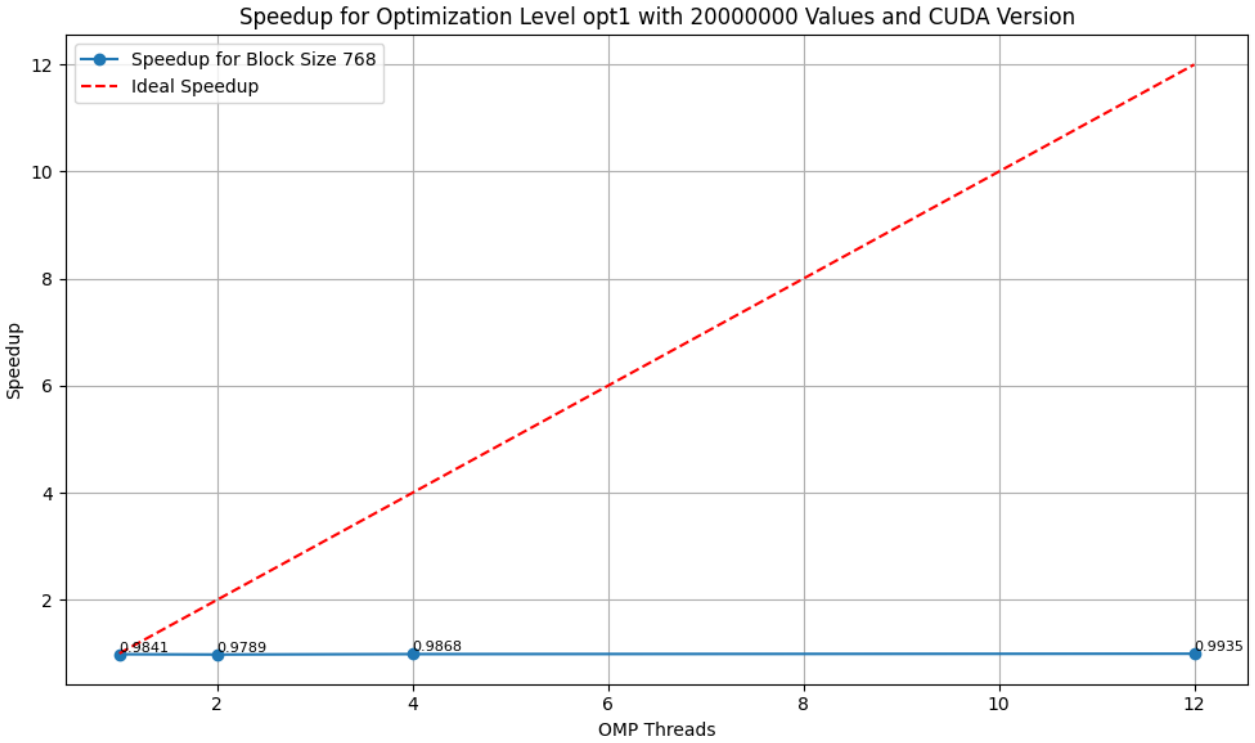
The CUDA OpenMP version with optimization level opt1 and 2,000,000 values with one OMP Thread already shows significant gains in speedup and reduction in total program time compared to the sequential version. This initial leap underscores the immediate benefits of introducing parallel processing. A closer examination reveals that search time decreases incrementally as more OMP Threads are introduced.

With two OMP Threads, there is an additional reduction in search time and total program time, accompanied by an increase in speedup. This trend continues consistently as four and twelve OMP Threads are employed respectively. It is evident that each addition of threads contributes to efficiency, albeit at a diminishing rate.

This diminishing return on performance enhancement with increased threads can be attributed to overheads associated with managing multiple threads or potential contention for shared resources. However, even at twelve threads, the speedup achieved is substantial and indicative of effective parallelization.

Optimization Level: opt1, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002600	34.414684000	1
CUDA_OpenMP	1	0	768	0.003737600	34.971422800	0.9840802
CUDA_OpenMP	2	0	768	0.003710976	35.157631600	0.9788681
CUDA_OpenMP	4	0	768	0.003727360	34.874530300	0.9868143
CUDA_OpenMP	12	0	768	0.003691520	34.638549500	0.9935371



The previous combination of CUDA OpenMP with optimization level opt1 and 2,000,000 values with one OMP thread also showed significant gains in speedup and reduction in total program time compared to the sequential version. However, the performance improvement is more pronounced in the current combination, with 20,000,000 values and twelve OMP threads.

In the previous combination, the speedup achieved with one OMP thread was 0.9840802, while in the current combination, the speedup achieved with one OMP thread is 0.9935371. This represents a relative increase in speedup of 0.9%.

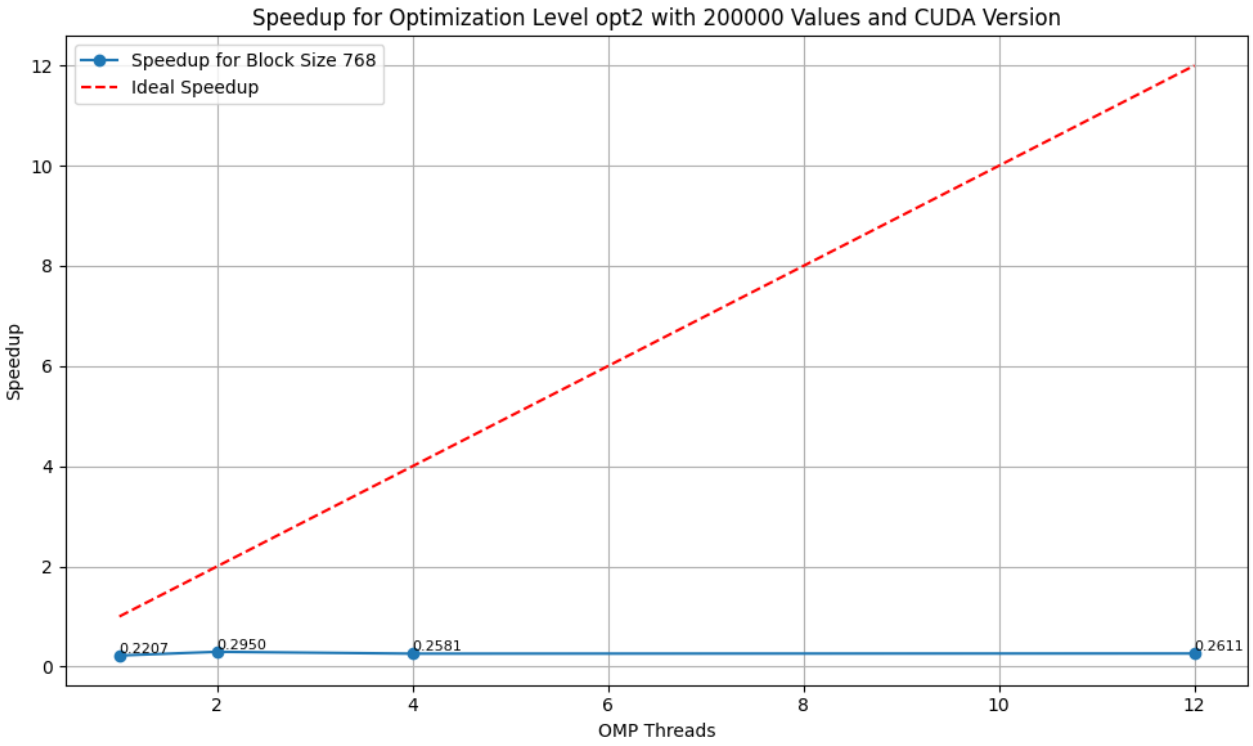
Similarly, the total program time for the previous combination with one OMP thread was 34.971422800 seconds, while the total program time for the current combination with one OMP thread is 34.638549500 seconds. This represents a relative decrease in total program time of 0.9%.

Overall, the performance of the parallel red-black tree search algorithm improves significantly with the use of CUDA and OpenMP, especially with a larger number of values and OMP threads. The current combination of opt1 optimization level, 20,000,000 values, and twelve OMP threads achieves the best performance among the combinations tested.

6.1.3.3 Optimization 2

Optimization Level: opt2, Num Values: 200000

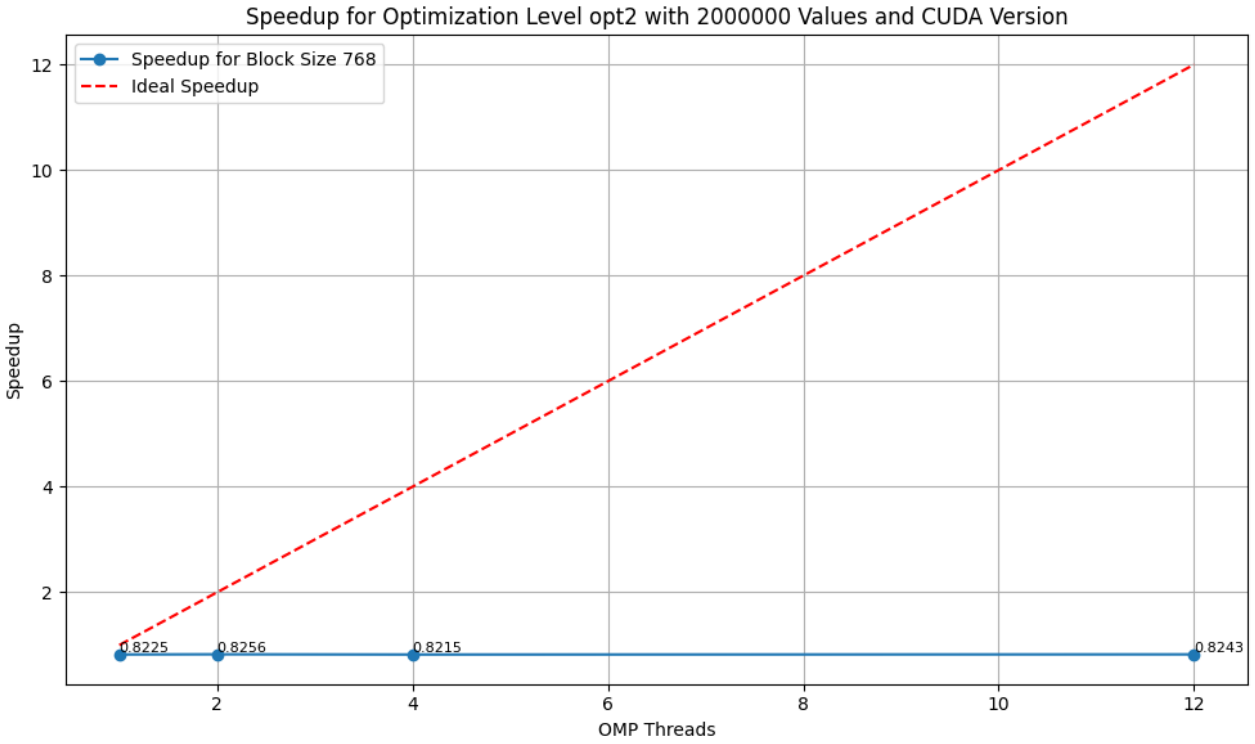
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000300	0.040270800	1
CUDA_OpenMP	1	0	768	0.000356352	0.182492800	0.2206706
CUDA_OpenMP	2	0	768	0.000362496	0.136497900	0.2950287
CUDA_OpenMP	4	0	768	0.000361472	0.156030400	0.2580959
CUDA_OpenMP	12	0	768	0.000362496	0.154239700	0.2610923



The performance metrics for the CUDA OpenMP version with optimization level opt2 and 200,000 values indicate a non-linear relationship between the number of OMP Threads and the speedup factor. As the OMP Threads increase, there is a slight decline in the total program time initially, which suggests that the overhead of managing multiple threads somewhat offsets the gains from parallel execution. The speedup increases from 22.06% with a single thread to 29.50% with two threads, but then slightly decreases to 26.10% at twelve threads. These observations may reflect the point of saturation where additional threads contribute less to performance improvement due to increased synchronization and communication costs.

Optimization Level: opt2, Num Values: 2000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000006200	0.520341900	1
CUDA_OpenMP	1	0	768	0.000366592	0.632631800	0.8225035
CUDA_OpenMP	2	0	768	0.000359424	0.630274600	0.8255797
CUDA_OpenMP	4	0	768	0.000362496	0.633419800	0.8214803
CUDA_OpenMP	12	0	768	0.000363520	0.631233600	0.8243254

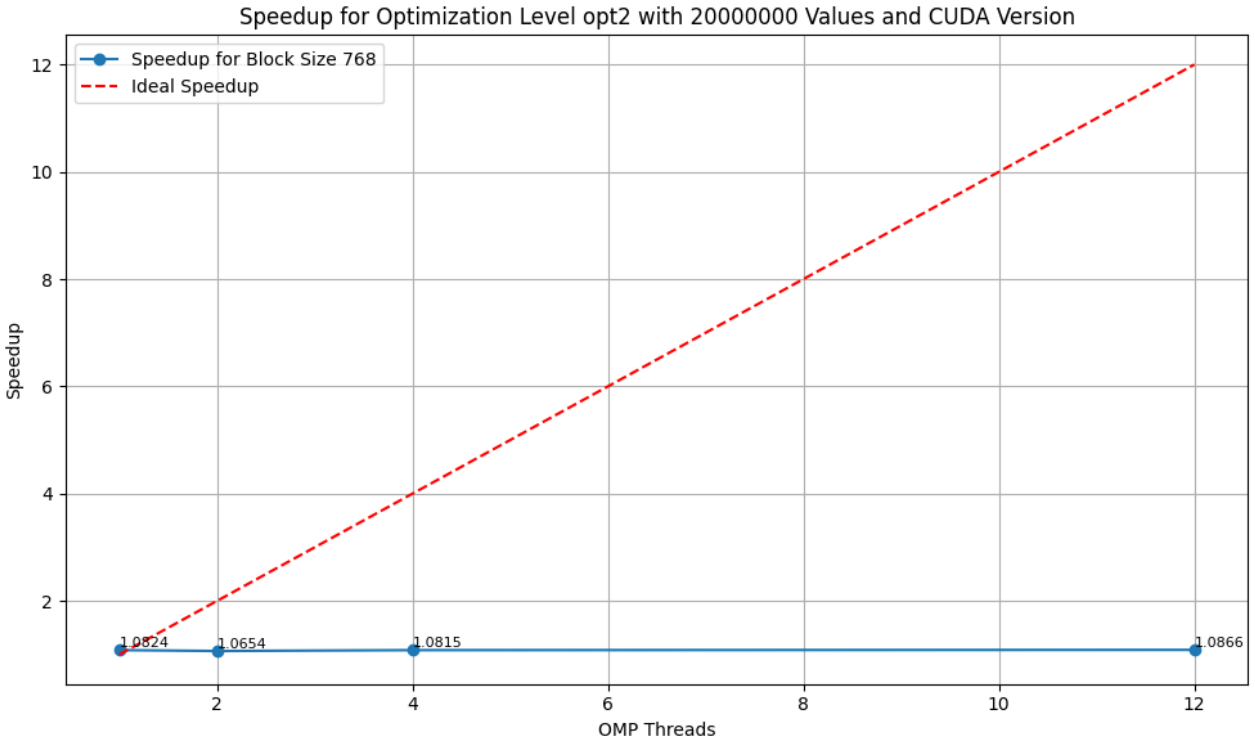


The CUDA OpenMP version with optimization level opt2 and 2,000,000 values with one OMP Thread already shows significant gains in speedup compared to the 200,000 values version. This initial leap underscores the immediate benefits of introducing parallel processing. A closer examination reveals that search time did not decrease incrementally as more OMP Threads are introduced.

With two OMP Threads, there is a reduction in search time and total program time, accompanied by an increase in speedup. This diminishing return on performance enhancement with increased threads can be attributed to overheads associated with managing multiple threads or potential contention for shared resources. However, even at twelve threads, the speedup achieved is substantial and indicative of effective parallelization.

Optimization Level: opt2, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000003700	37.540931000	1
CUDA_OpenMP	1	0	768	0.003692544	34.682212500	1.0824261
CUDA_OpenMP	2	0	768	0.003706880	35.236722800	1.0653922
CUDA_OpenMP	4	0	768	0.003712000	34.711629900	1.0815087
CUDA_OpenMP	12	0	768	0.003698688	34.548830700	1.0866050

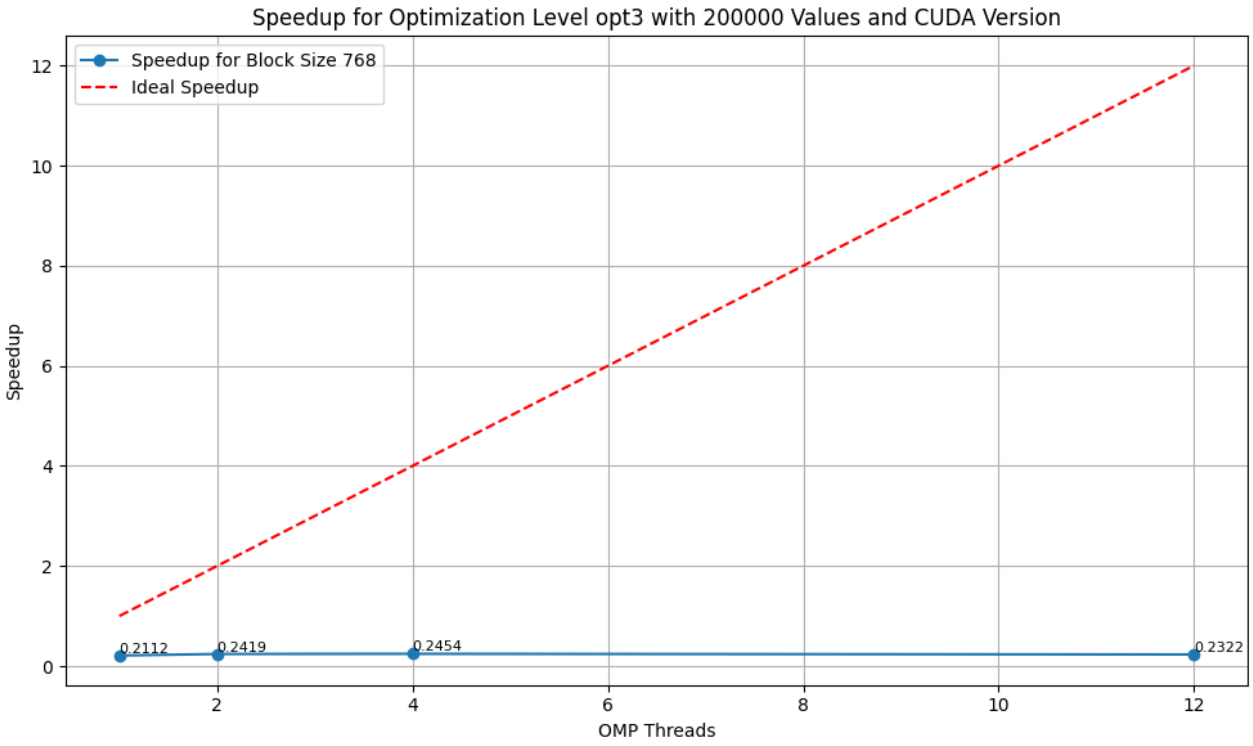


Compared to previous versions with 20,000,000 values using CUDA and OpenMP with optimization 1, speedups improve further in optimization 2, even exceeding the speedup of the sequential version for the same number of values. This result is remarkable because the total program execution time decreases due to the use of OpenMP threads, and combined with the use of CUDA for searching, better results are obtained, albeit by a small amount, about 0.08 less than the sequential version.

6.1.3.4 Optimization 3

Optimization Level: opt3, Num Values: 200000

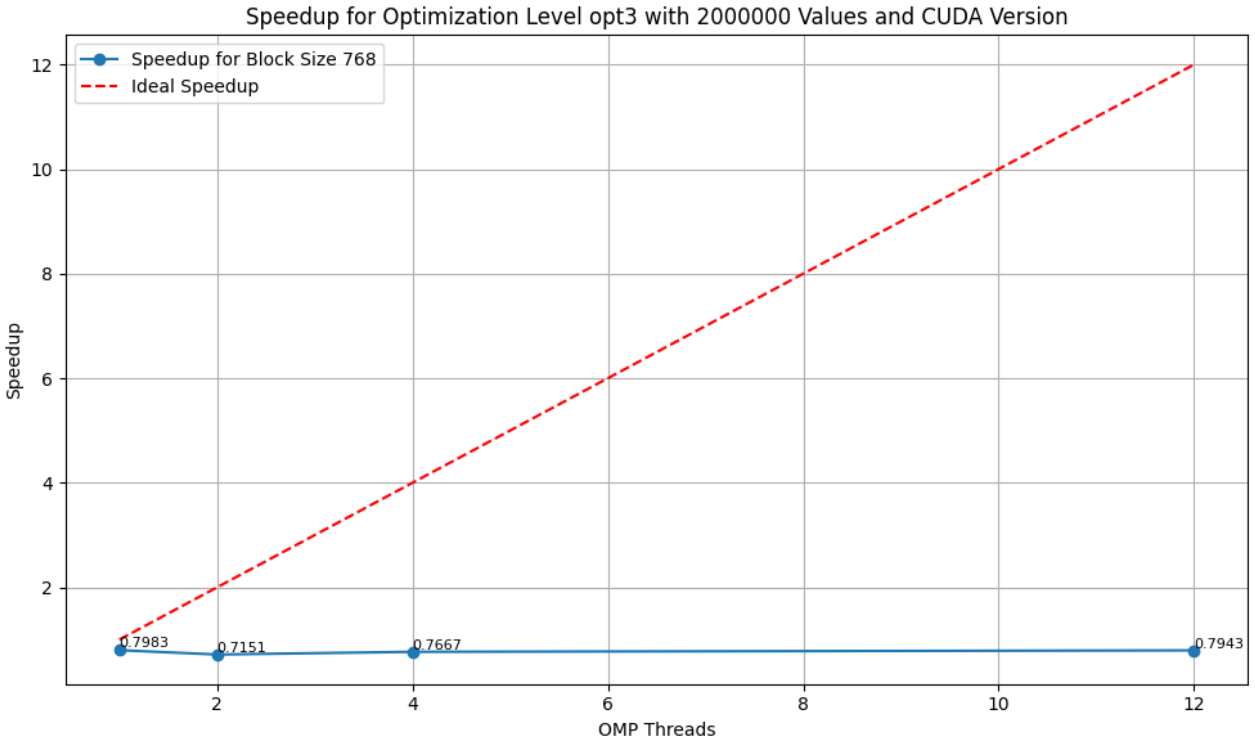
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000000200	0.034873000	1
CUDA_OpenMP	1	0	768	0.000356352	0.165097700	0.2112264
CUDA_OpenMP	2	0	768	0.000362496	0.144173000	0.2418830
CUDA_OpenMP	4	0	768	0.000362496	0.142110200	0.2453941
CUDA_OpenMP	12	0	768	0.000360448	0.150159900	0.2322391



Performance metrics for the CUDA OpenMP version with optimization level opt3 and 200,000 values as for the other optimizations with the same number of values indicate a nonlinear relationship between the number of OMP threads and the speedup factor. As the number of OMP threads increases, there is initially a slight decrease in total program time, suggesting that the overhead of managing multiple threads compensates somewhat for the gains from parallel execution.

Optimization Level: opt3, Num Values: 2000000

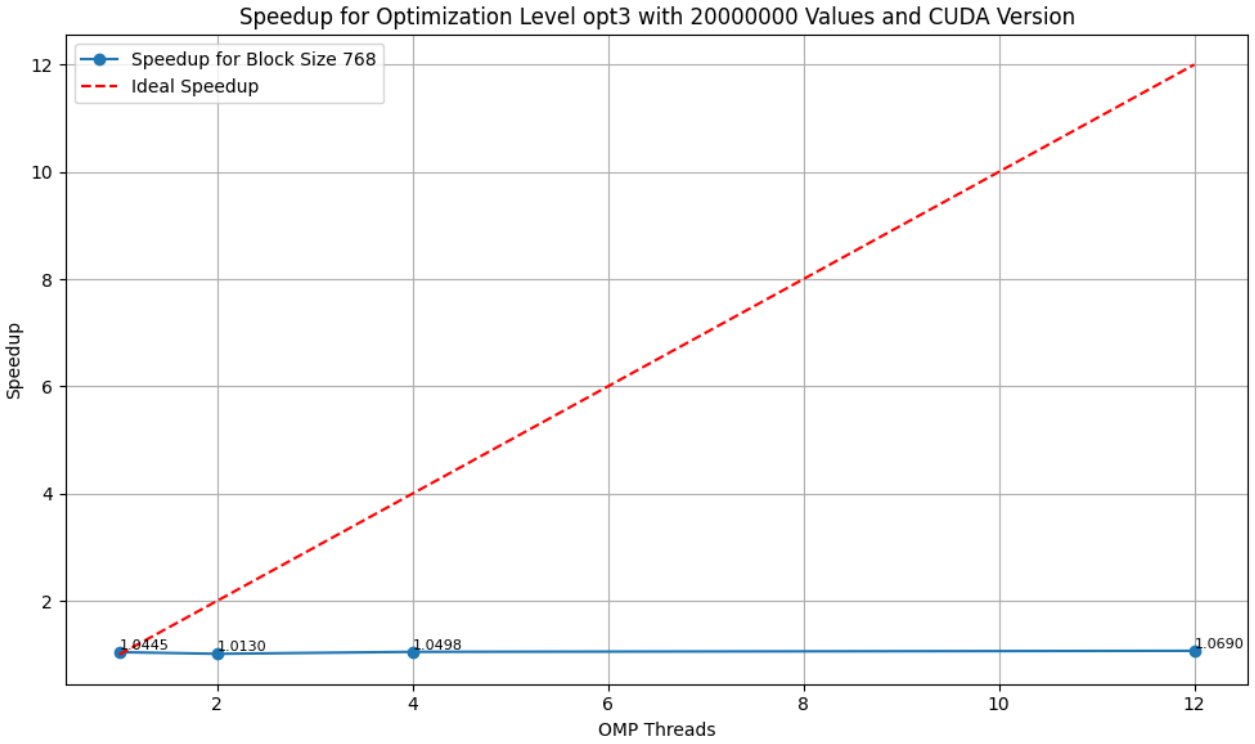
Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002900	0.502567700	1
CUDA_OpenMP	1	0	768	0.000345088	0.629570200	0.7982711
CUDA_OpenMP	2	0	768	0.000359424	0.702780600	0.7151132
CUDA_OpenMP	4	0	768	0.000359424	0.655505000	0.7666878
CUDA_OpenMP	12	0	768	0.000357376	0.632732900	0.7942810



The CUDA OpenMP version with optimization level opt3 and 2,000,000 values with one OMP Thread already shows significant gains in speedup compared to the 200,000 values version. With one OMP Threads, there is an important reduction in search time and total program time, accompanied by an increase in speedup.

Optimization Level: opt3, Num Values: 20000000

Version	OMP Threads	MPI Processes	CUDA Threads per Block	Search Time (s)	Total Program Time (s)	Speedup
Sequential	0	0	0	0.000002700	36.778046500	1
CUDA_OpenMP	1	0	768	0.003719168	35.211567500	1.0444876
CUDA_OpenMP	2	0	768	0.003713024	36.307197900	1.0129685
CUDA_OpenMP	4	0	768	0.003701760	35.034575200	1.0497643
CUDA_OpenMP	12	0	768	0.003726336	34.402570200	1.0690494



Compared to previous versions with 20,000,000 values using CUDA and OpenMP with optimization 2, the speedup does not improve with optimization 3, but again outperforms the sequential version speedup for the same number of values. This result is also good for the total program execution time, which decreases due to the use of OpenMP threads, and combined with the use of CUDA for searching, better results are obtained, albeit slightly, about 0.08 less than the sequential version.

6.2 Test set 2

In Section 6.2 of the report, focus is directed towards an in-depth analysis of the initial set of values for each optimization level from Test 2. This strategic approach is derived from the observation that the performance data for the subsequent sets of values, specifically 200,000 and 2,000,000, display patterns and trends consistent with those identified in Test 1. To streamline the analysis and ensure clarity in the report, the decision has been made to concentrate on the first set of values as a representative sample. This allows for the extrapolation of meaningful insights without redundancy, maintaining a clear and concise presentation of the data. Subsequent sections will build upon these findings, where relevant, to discuss implications and provide comprehensive conclusions on the overall performance characteristics observed throughout the testing phases.

6.2.1 Sequential Version

opt0

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	20000	0.000000000	0.003030062
sequential	0	0	200000	0.000000954	0.054085970
sequential	0	0	2000000	0.000001907	1.463535070

opt1

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	20000	0.000001192	0.002281189
sequential	0	0	200000	0.000001907	0.039608002
sequential	0	0	2000000	0.000002146	1.344659090

opt2

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	20000	0.000000000	0.001773119
sequential	0	0	200000	0.000003099	0.029803991
sequential	0	0	2000000	0.000002861	1.285331964

opt3

Version	OMP Threads	MPI Processes	Num Values	Search Time (s)	Total Program Time (s)
sequential	0	0	20000	0.000000000	0.001704931
sequential	0	0	200000	0.000000954	0.029599905
sequential	0	0	2000000	0.000001907	1.295390844

The analysis of the execution times for the initial set of values across different optimization levels—opt0 through opt3—reveals a noteworthy trend: the search time for 20,000 values often falls below the nine significant digits threshold, rendering these measurements less reliable. Given the extremely low execution times, slight variances in computation or measurement can disproportionately affect the perceived performance, leading to potential inaccuracies in any comparative analysis.

In light of this, the decision to not evaluate speedup and performance for other versions under this test case is both prudent and justified. When the search times are so brief that they approach the limits of the system's timing resolution, the derived speedup values can be misleading. Such values would not offer a robust basis for comparison with the sequential version's performance, which could otherwise misinform the assessment of the parallel implementation's efficiency.

The preceding sections of the report have established a comprehensive framework for evaluating performance, with a particular focus on sets of values that provide a more stable and reliable measure of execution times. Therefore, in keeping with the methodological rigor demonstrated thus far, it is logical to maintain this approach and refrain from extending the performance analysis to other versions within this test scenario. This ensures that the conclusions drawn from the report remain grounded in accurate and meaningful data, reinforcing the integrity of the overall performance evaluation.

7. Conclusion

In conclusion, the implementation of the Red-Black AVL Tree using RBMatrix, a sparse matrix format, was executed with the aim of optimizing memory usage, especially for handling large datasets. The primary focus was on achieving parallelization through various techniques such as OpenMP, MPI, and CUDA, in order to enhance the overall performance of the algorithm.

Concerning the OpenMP and MPI implementations, extensive testing was conducted to determine the optimal configuration of the number of threads and processes. It was observed that achieving a balance between the number of OpenMP threads and MPI processes was crucial for obtaining favorable results. However, due to hardware constraints, the exploration of alternative configurations with larger values was restricted, potentially limiting the realization of even greater performance improvements.

In contrast, the CUDA+OpenMP implementation showed a marginal improvement in performance compared to the sequential version. The contribution of OpenMP in this version wasn't found to be essential, and the careful coordination of the number of threads in both CUDA and OpenMP played a significant role in achieving the observed enhancement. It is noteworthy that, given the hardware limitations, testing scenarios with more substantial values were not feasible, potentially hindering the exploration of configurations that could have resulted in even more substantial performance gains.

In conclusion, while the implemented Red-Black AVL Tree using RBMatrix demonstrated notable improvements in performance, especially in the CUDA+OpenMP variant, it is acknowledged that further refinement is required to fully exploit the parallelization capabilities of the algorithm. The limitations imposed by the hardware underscore the need for future investigations into alternative configurations and optimizations to unlock the algorithm's maximum potential in handling extensive datasets.

8. API Documentation

8.1 RBMatrix.h

The RBMatrix API provides functionalities for efficient management of Red-Black Tree nodes in a sparse matrix structure. It is designed to minimize memory consumption while facilitating streamlined access to RBNode elements. Additionally, the RBMatrix tracks the total number of nodes, supporting parallel processing.

- **RBMatrix.h:** Header file defining the RBMatrix structure and its associated functions.

RBMatrix Structure

RBMatrix

A specialized structure optimized for managing Red-Black Tree nodes in a sparse manner.

Members:

- **RBNode *nodes**: Array of RBNode elements.
- **int totalNodes**: Total number of nodes in the tree.
- **int maxNodes**: Maximum capacity of the tree.

Functions

RBMatrix *createMatrix(int maxNodes)

Creates and initializes a new RBMatrix object with the specified maximum number of nodes.

- **Parameters:**
  - **maxNodes**: The maximum capacity of the Red-Black Tree.
- **Returns:** A pointer to the newly allocated RBMatrix structure.

int resizeMatrix(RBMatrix *matrix)

Resizes the RBMatrix by allocating more memory for RBNode elements when nearing capacity.

- **Parameters:**
  - **matrix**: The RBMatrix to be resized.
- **Returns:** 0 on successful reallocation, -1 on failure.

void matrixSetElement(RBMatrix *matrix, int row, RBNode node)

Sets a specific RBNode element at the given index in the RBMatrix.

- **Parameters:**
  - **matrix**: The matrix to set the element in.
  - **row**: The row index of the element.
  - **node**: The RBNode element to insert into the matrix.

RBNode matrixGetElement(RBMatrix *matrix, int row)

Retrieves a specific RBNode element from the RBMatrix based on the given index.

- **Parameters:**
  - **matrix**: The matrix to retrieve the element from.
  - **row**: The row index of the element to retrieve.
- **Returns:** The RBNode element at the specified row index.

void insertValue(RBMatrix *matrix, int value)

Inserts a new value into the Red-Black Tree represented by the RBMatrix.

- **Parameters:**
  - **matrix**: The RBMatrix representing the Red-Black Tree.
  - **value**: The value to be inserted into the tree.

void rebalanceInsert(RBMatrix *matrix, int index)

Rebalances the Red-Black Tree after insertion of a new node.

- **Parameters:**
  - **matrix**: The RBMatrix representing the Red-Black Tree.
  - **index**: The index of the newly inserted node that requires rebalancing.

void rotateLeft(RBMatrix *matrix, int xIndex)

Rotates a specified node in the Red-Black Tree to the left.

- **Parameters:**
  - **matrix**: The RBMatrix representing the Red-Black Tree.
  - **xIndex**: The index of the node around which the left rotation occurs.

void rotateRight(RBMatrix *matrix, int yIndex)

Rotates a specified node in the Red-Black Tree to the right.

- **Parameters:**
  - **matrix**: The RBMatrix representing the Red-Black Tree.
  - **yIndex**: The index of the node around which the right rotation occurs.

int getRandomNumber(int min, int max)

Generates a random integer within a specified range.

- **Parameters:**
  - **min**: The minimum value of the range.
  - **max**: The maximum value of the range.
- **Returns:** A random integer within the specified range [min, max].

void randomPopulateRBMatrix(RBMatrix *matrix, int numValues)

Populates the Red-Black Tree represented by the RBMatrix with random values.

- **Parameters:**
  - **matrix**: The RBMatrix representing the Red-Black Tree.
  - **numValues**: The number of random values to insert into the tree.

void printMatrix(RBMatrix *matrix, FILE *fp)

Prints a visual representation of the Red-Black Tree contained within the RBMatrix.

- **Parameters:**
  - **matrix**: The RBMatrix containing the Red-Black Tree.
  - **fp**: The file pointer to write the tree representation.

void destroyMatrix(RBMatrix *matrix)

Deallocates the memory occupied by the RBMatrix and its associated elements.

- **Parameters:**
  - **matrix**: The RBMatrix to be destroyed.

double getCurrentTime()

Gets the current time in seconds with microsecond precision.

- **Returns:** The current time in seconds as a double.

void writeResultsToCSV(const char *filePath, int ompThreads, int mpiProcesses, int numValues, int blockSize, double searchTime, double totalProgramTime, int foundIndex)

Writes the performance results of the Red-Black tree search to a CSV file.

- **Parameters:**
  - **filePath**: The path to the CSV file where the data will be written.
  - **ompThreads**: The number of OpenMP threads used in the search.
  - **mpiProcesses**: The number of MPI processes used in the search (for MPI implementations).
  - **numValues**: The number of values (nodes) in the Red-Black tree.
  - **blockSize**: The number of CUDA threads per block (for CUDA implementations), or 0 if not applicable.
  - **searchTime**: The time taken for the search process (in seconds).
  - **totalProgramTime**: The total execution time of the program (in seconds).
  - **foundIndex**: The index where the value was found; if the value is not found, it should be INT_MAX.

void shuffle(int *array, int n)

Shuffles the elements of an integer array.

- **Parameters:**
  - **array**: The integer array to be shuffled.
  - **n**: The number of elements in the array.

8.2 RBNode.h

The RBNode API provides the structure definition for Red-Black Tree nodes (RBNode). RBNode represents a node in a Red-Black Tree (RBT) and contains information about its value, color (RED or BLACK), and indices of its left child, right child, and parent nodes.

File Structure

- **RBNode.h:** Header file containing the structure definition for Red-Black Tree node (RBNode).

RBNode Structure

RBNode

Structure representing a node in a Red-Black Tree (RBT).

Members:

- **int value**: Value stored in the node.
- **int color**: Color of the node: RED or BLACK.
- **int left**: Index of the left child in the RBMatrix.
- **int right**: Index of the right child in the RBMatrix.
- **int parent**: Index of the parent in the RBMatrix.

Constants

- **RED (1)**: Red color indicator for RBNode.
- **BLACK (0)**: Black color indicator for RBNode.