UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING AND ELECTRICAL AND APPLIED MATHEMATICS

Master's Degree in Computer Engineering

Project Report

Parallel version of the Search Algorithm on a RB Tree with OMP+MPI and OMP+CUDA approaches

Lecturer:
Francesco Moscato
fmoscato@unisa.it

Giuseppe D'Aniello
gi.daniello@unisa.it

Student:
Christian Conato
Mat.0622702273
c.conato@studenti.unisa.it

ACADEMIC YEAR 2023/2024

Contents

Chapter 1

Problem Description (Single machine parallelization (F.Moscato)

Provide a parallel version of the RB Search algorithm to find a key in all of its nodes. Implementation MUST use an hybrid message passing / shared memory paradigm and has to be implemented by using MPI and OpenMP. Students MUST provide parallel processes on different nodes, and each process has to be parallelized by using OpenMP (i.e.: MPI will spawn OPENMP-compiled processes). Furthermore, an implementation with OpenMP and CUDA must be provided.
The parallel algorithm used in "OpenMP + MPI" solution could not be the same of the "OpenMP + CUDA" approach.

1.1 Introduction to the problem

A red–black tree is a specialised binary search tree data structure noted for fast storage and retrieval of ordered information, and a guarantee that operations will complete within a known time. Compared to other self-balancing binary search trees, the nodes in a red-black tree hold an extra bit called "color" representing "red" and "black" which is used when re-organising the tree to ensure that it is always approximately balanced.
When the tree is modified, the new tree is rearranged and "repainted" to restore the coloring properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that this rearranging and re-coloring can be performed efficiently. For simplicity we will consider the presence of duplicates in the tree because if we did not allow duplicates we would have to perform a search each time on all the nodes in the tree that are already present, and the meaning of searching for a node in a tree that already has all the nodes would be lost. The search on a RB Tree is a classic search on a binary search tree implementing all its native methods; the complexity of the search in all the cases (best, average and worst) is *O(logn)* considering all the (re-)balancing that could occur.

1.2 Solution

We'll start from the sequential version and after that we'll consider the OMP+MPI implementation and only eventually the OMP+CUDA approach.
The sequential version is the classic RB Search on a RB Tree that proceeds in a recursive way.
We could proceed with the iterative implementation but this will require a complexity of *O(n)* (with n the number of the nodes in a tree) at the expense of the recursive version, which employs a complexity of $O(\log_2(n))$.
Thus, the decision of the implementation of the serial search algorithm falls to its recursive implementation.
Since the RB Tree is a classic binary search tree, the nodes are organized in an ascending order where the key of the left child of the root is less than the key of its root (vice versa for the right child).

Thus, the search firstly will compare the key to be found with the key of the root:

- if its key is equal the key is found;
- if the key is less than the key of the root the search will proceeds to the left child;
- if the key is greater than the key of the root the search will proceeds to the right child;

The search will end when a leaf is reached or the key is found. Otherwise, for this reason, the parallelization with OMP and MPI could be problematic for several reasons:

- **Data dependence**: Recursive algorithms often depend heavily on the partial results of recursion. If you parallelize one instance of the algorithm, you need to ensure that the intermediate results are available to the other instances. This can lead to complex management of synchronization and data sharing, which can result in additional overhead and complexity.
- **Parallelization overhead**: The introduction of parallelism inevitably involves some overhead. Creating threads or processes, managing synchronization and communication between work units may require more resources than serial execution of the algorithm. In some cases, this overhead may outweigh the benefits of parallelization, especially for smaller problems.
- **Difficulties in work partitioning**: Some recursive algorithms may not be easily divided into independent, parallelizable subproblems. In some cases, the division of labor may be complex.

The classic idea for parallelizing an algorithm is to divide the entire work structure into smaller parts and assign each part to a different process.
Since the tree structure is not such a simple structure, the idea for dividing the structure into smaller parts is not so easy to think of. So the choice fell on moving the whole structure to a structure on which it is easier to partition the data, namely an array.
In this array we will apply the classical binary search while also respecting the search complexity in a binary search tree i.e. $O(\log_2(n))$.

Chapter 2

OMP+MPI Parallelization

MPI, also known as Message Passing Interface, serves as a widely adopted library for parallel computing on distributed memory systems. It furnishes a collection of functions facilitating inter-process communication, enabling multiple processes to collaborate and enhance problem-solving efficiency by distributing the workload among them. Being a standard, MPI is not bound to any specific system or architecture, rendering it portable and applicable across a diverse range of computers, spanning from small clusters to extensive supercomputers. One of MPI's principal merits lies in its capacity to afford substantial flexibility in designing parallel algorithms. The library offers different communication operations, encompassing point-to-point communication (message exchange between two processes), collective communication (message exchange among all processes), and non-blocking communication (permitting processes to persist in execution while awaiting a message). Despite these advantages, MPI has its limitations. Notably, it lacks support for shared memory programming, a feature valuable for certain algorithmic scenarios. In contrast, OpenMP provides a valuable solution for parallelizing existing code initially crafted for single-core execution. However, realizing benefits from OpenMP is not universally guaranteed. Several factors may limit its suitability or the advantages derived from parallelization. One circumstance where OpenMP might be less fitting is when the code is already optimized for single-core execution. In such instances, parallelizing the code may not yield a significant performance boost. Additionally, OpenMP is predominantly designed for parallelizing code heavily reliant on loops, such as code involved in solving problems within scientific computing. If the code exhibits a lower dependence on loops, reaping benefits from parallelization might prove challenging. Another scenario where OpenMP may not be the most suitable choice is when the code necessitates extensive communication between cores. In cases demanding a high level of inter-core communication, it might be more apt to utilize another parallel library, such as MPI, specifically crafted to manage heightened communication requirements between cores.

2.1 OMP+MPI Approach

For OMP+MPI approach there is a MPI process (the process with rank=0) that will send different partitions of this array to other MPI processes (and to himself) applying the binary search on their portion of the array.
At this point each process will instruct its OMP threads to work on an even smaller portion of this array, creating subarrays and applying binary search on each subarray.
If a thread won't find the result it will return **-1**; otherwise, when a thread will find the key to be found within its array, a result variable will be overwritten and only at the end the rank 0 will gather all the results of all the threads making the max of all the results and notify that the result has been found.
If any thread won't find the result the rank 0 will say that the result hasn't been found.
Before being able to distribute the elements of the sorted array, for simplicity, we have defined a new MPI_datatype called **MPI_SimpleNode**, which simply contains information about the node's key and its color.

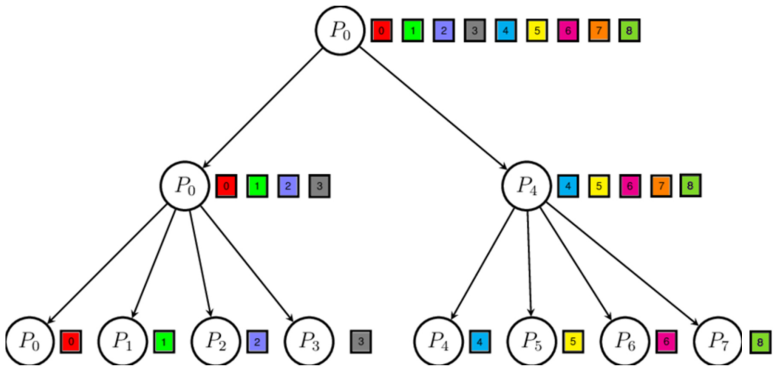

Figure 2.1: Scatterv distribution elements

To enable rank 0 to distribute portions of the array to all other processes, the **MPI_Scatterv** function will be used, which is different from **MPI_Scatter**.
Below is the function signature:

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts,
                 const int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

The choice of MPIScatterv is not random but dictated by the fact that the array may contain an uneven number of elements. Therefore, each rank will not have exactly the same number of elements.

Scatterv allows for the distribution of contiguous elements of the array in an uneven manner, thanks to two parameters, namely two arrays (sendcounts and displs).
In particular, sendcounts stores in each of its positions the number of elements to be distributed for each rank, while displs stores the starting offset from which rank 0 should retrieve the next counts[rank] items for the subsequent rank.

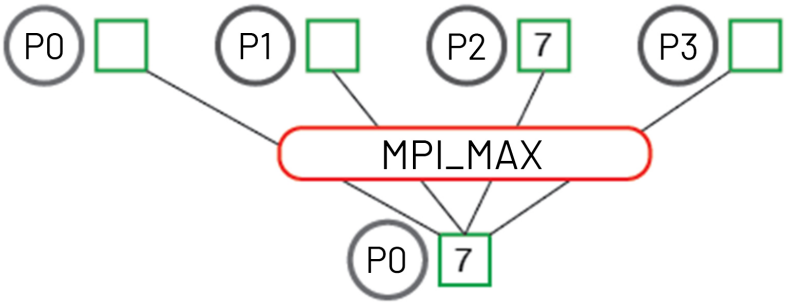

Figure 2.2: MPI_Reduce

At this point, when all the process will receive their elements into a localArray, what happens is that each rank will distribute its elements received from rank 0 among its own OpenMP threads, with the subsequent check to ensure that the received elements are an odd number to avoid loss of information.
A dedicated distribution function, as seen among processes, will not be used. Instead, each OpenMP thread within the process will seek elements in subportions of the local array assigned to all other ranks by rank 0. The local array will be conceptually divided among all available OpenMP threads, searching in a limited number of elements equal to the ratio of the local array's length to the number of available threads.

In this way, based on the OpenMP thread ID, each thread will commence its search after **ID * numElementsPerThread** until the entire local array has been thoroughly examined. It becomes the actual task of the threads to search for the result, and when found, it will be inserted into a shared variable and returned only when all threads within a single process have completed their search tasks. Once all threads from all processes have finished searching within their local arrays, another MPI function will be used to gather all the localresults from all processes and consolidate them into a globally defined result.

The useful function for our purpose is precisely **MPI_Reduce**.
Below is the signature of the function.

```
int MPI_Reduce( void* sendbuf, void* recvbuffer, int count,
                MPI_Datatype datatype, MPI_Op op,
                int root, MPI_Comm communicator)
```

What MPI_Reduce does is to reduce all the results of all the processes according to an MPI_Op op that can be sum, difference, maximum, etc... To get a single result and put it in the global result, the operation that comes in handy is just the maximum. If the key within the local array is not found by the threads, the local result is never valued, but if it is found the very result is returned. So by making the maximum out of all these local results, the maximum will be just the result found and so it will end up in a variable that we will call "**globalResult**." This operation also covers the case where there are multiple threads finding the same result (because we said that in our RB tree we accept duplicate keys) because the maximum between two equal results is precisely the same result. If, on the other hand, no thread finds the element, the local result will never be valued and so doing the maximum between no results is equivalent to not doing the maximum at all.

2.2 Experimental setup

Now it's time to begin the experiments, starting by first presenting the hardware characteristics of the CPU.

2.2.1 Hardware configuration

Socket 1	: ID = 0
Number of cores	: (max 14)
Number of threads	: (max 20)
Hybrid	: yes, 2 coresets
Core Set 0	: P-Cores, 6 cores, 12 threads
Core Set 1	: E-Cores, 8 cores, 8 threads
Manufacturer	: GenuineIntel
Name	: Intel Core i7 12700H
Codename	: Alder Lake
Specification	: 12th Gen Intel(R) Core(TM) i7-12700H
Package (platform ID)	: Socket 1744 FCBGA (0x7)
CPUID	: 6.A.3
Extended CPUID	: 6.9A
Core Stepping	: L0
Technology	: 10 nm

At the top all the CPU features are displayed, but only a portion is shown. In particular, our focus will be on the number of cores and logical threads, as they will be crucial for conducting the experiments correctly to avoid undesired results.

2.2.2 Softwares

- **VisualStudio** used for coding
- **Python** used for plots
- **R Studio** used for tables
- **Makefile** used for creating the script

2.2.3 Time Measures

Understanding the chosen approach for parallelization with OMP and MPI, what is useful for comprehending the effectiveness or lack thereof of the carried out parallelization is precisely time, as there is nothing more immediately visible than to see whether the time elapsed for the execution of the parallel algorithm is greater or less than the well-known serial version.
We will carry out a total of four different time measurements, considering the following times:

- **RB_Tree_creation_time**: Time elapsed between the start and end of the creation of the RB tree.
- **RB_Tree_communication_time**: Communication time among MPI processes.
- **RB_Tree_search_time**: Execution time of the binary search algorithm.
- **RB_Tree_execution_time**: Total program execution time.

2.2.4 Performance, Speed-Up & Efficiency

In the realm of high-performance computing, the correlation between speed-up and efficiency holds significant significance.
Speed-up gauges the enhanced speed achieved through the utilization of multiple processors in comparison to a solitary processor.
On the other hand, efficiency assesses the actual amount of work accomplished by the processors relative to the maximum theoretical workload.
Traditionally, speed-up is quantified as the ratio of the execution time on a single processor to the execution time on multiple processors:

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

In the context where T_1 represents the duration of the most efficient known serial algorithm, and $T_p(n)$ denotes the execution time of the parallel algorithm using n processors. For instance, if a particular application takes 100 seconds to execute on a single processor and 50 seconds on 4 processors, the resulting speed-up is 2. In theory, the utilization of multiple processors should ideally yield linear speed-up concerning the number of processors employed. Nonetheless, practical implementation faces challenges related to communication and synchronization among processors, making achieving perfect linearity not always feasible.

Efficiency, particularly relative efficiency, is given by the ratio of the time to execute an algorithm on a single processor to n multiplied by the time to execute a parallel algorithm on n processors:

$$E_r = \frac{T_1}{n \cdot T_n}$$

Using the previous example, relative efficiency would be $100/(4 \cdot 50)$ on 4 processors. In theory, efficiency should always be greater than or equal to 1, as more processors are used to perform the work. However, in practice there may be losses in efficiency due to issues of communication and synchronization between processors.

2.2.5 Analysis

For the analysis of timings, speedup, and efficiency, a well-defined criterion will be followed: the serial version of the algorithm and the parallel versions implemented with OMP and MPI will be considered.
Analyses will be conducted for all four optimization levels (**-O0, -O1, -O2, -O3**).
For each optimization level, three analyses will be performed based on the problem size, specifically, the number of nodes belonging to a binary tree.
In particular, three case studies will be examined:

1. The sequential and parallel programs compiled and executed with a tree of **10000 nodes**;
2. The sequential and parallel programs compiled and executed with a tree of **250000 nodes**;
3. The sequential and parallel programs compiled and executed with a tree of **3500000 nodes**.

This selection is made to observe the trends of timings, speedup, and efficiency across different types of trees.
Additionally, depending on the problem size, two different search keys will be searched to highlight differences in search performance, whether the key to be found is located toward the center of the search range or at the extremes.
Finally, for each individual case study, considering the number of cores and logical threads discussed in the previous sections, analyses will be conducted following this pattern:

- 0 MPI Processes - No OpenMP threads (Sequential version)
- 1 MPI Process - 1, 2, 4, 8, 16 OpenMP threads (Parallel version)
- 2 MPI Processes - 1, 2, 4, 8, 16 OpenMP threads (Parallel version)
- 4 MPI Processes - 1, 2, 4, 8, 16 OpenMP threads (Parallel version)
- 8 MPI Processes - 1, 2, 4, 8, 16 OpenMP threads (Parallel version)

Chapter 3

OMP+MPI Analysis of the plots

This chapter will show the anlysis of the plots made by analyzing the sequential version and the parallel version of the RB Tree Search algorithm carried out with OMP and MPI. We will start from the optimization 0 (no optimization) level discussing about the three case studies for each optimization level showing both tables and graphs related to them.

3.1 Optimization 0

3.1.1 10000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0015490	0.0000000	0.0000003	0.0017614	-	-
OMP+MPI	1	1	0.0015136	0.0000341	0.0000190	0.0017960	1	100
OMP+MPI	1	2	0.0015941	0.0002921	0.0000811	0.0021540	0.83379759	41.6899
OMP+MPI	1	4	0.0016221	0.0006893	0.0001982	0.0025802	0.69607007	17.4018
OMP+MPI	1	8	0.0017139	0.0015760	0.0002899	0.0035659	0.50369967	6.2957
OMP+MPI	2	1	0.0014875	0.0000311	0.0001685	0.0019234	0.93376313	46.6882
OMP+MPI	2	2	0.0016032	0.0003024	0.0002774	0.0023460	0.7655584	19.139
OMP+MPI	2	4	0.0016228	0.0007880	0.0002913	0.0027685	0.64872675	8.1091
OMP+MPI	2	8	0.0017799	0.0015573	0.0003970	0.0036937	0.48623332	3.039
OMP+MPI	4	1	0.0014863	0.0000301	0.0003952	0.0021385	0.83984101	20.996
OMP+MPI	4	2	0.0016164	0.0003129	0.0003729	0.0024824	0.72349339	9.0437
OMP+MPI	4	4	0.0016397	0.0007581	0.0003655	0.0028407	0.63223853	3.9515
OMP+MPI	4	8	0.0017069	0.0017853	0.0005080	0.0039472	0.45500608	1.4219
OMP+MPI	8	1	0.0015047	0.0000318	0.0005382	0.0023092	0.77775853	9.722
OMP+MPI	8	2	0.0016063	0.0003192	0.0005332	0.0026096	0.68822808	4.3014
OMP+MPI	8	4	0.0016268	0.0008334	0.0006093	0.0031263	0.57448102	1.7953
OMP+MPI	8	8	0.0018131	0.0018099	0.0006595	0.0042571	0.42188344	0.6592
OMP+MPI	16	1	0.0015129	0.0000313	0.0007614	0.0025404	0.70697528	4.4186
OMP+MPI	16	2	0.0016268	0.0003026	0.0008583	0.0029695	0.60481563	1.89
OMP+MPI	16	4	0.0016737	0.0008375	0.0009043	0.0034370	0.52254873	0.8165
OMP+MPI	16	8	0.0018312	0.0018684	0.0009727	0.0046303	0.38787983	0.303

Figure 3.1: table 10000 nodes

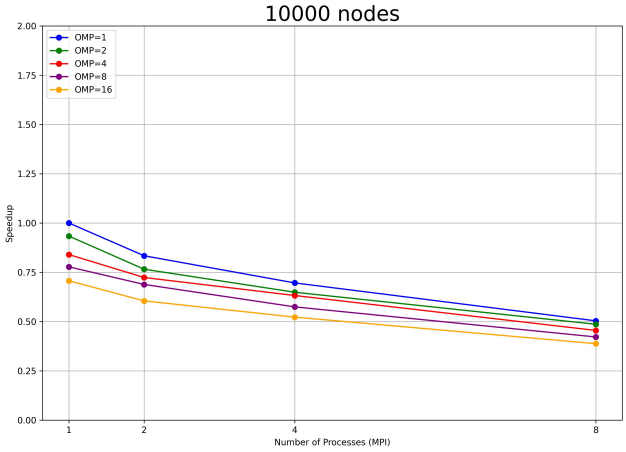

Figure 3.2: plot 10000 nodes

In this first case, the numbers of nodes isn't particularly high, but it's enough to show the initial differences in terms of speedup. We observe that the reference curve, namely the single-core single-thread one, is never surpassed by the other versions with a higher number of MPI processes. This is reasonable, as versions with more MPI processes have a longer execution time due to the overhead caused by the communication time between processes. Indeed, what we primarily notice is an increase in communication time as the number of MPI processes increases, and it decreases when MPI returns to being 1.

3.1.2 250000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0702608	0.0000000	0.0000003	0.0705366	-	-
OMP+MPI	1	1	0.0728568	0.0003901	0.0000331	0.0831842	1	100
OMP+MPI	1	2	0.0770389	0.0014609	0.0000291	0.0929674	0.89476741	44.7384
OMP+MPI	1	4	0.0973660	0.0029769	0.0003026	0.1198830	0.6938782	17.347
OMP+MPI	1	8	0.1502683	0.0064659	0.0003205	0.1833241	0.45375485	5.6719
OMP+MPI	2	1	0.0736875	0.0003976	0.0003080	0.0857984	0.9695309	48.4765
OMP+MPI	2	2	0.0807508	0.0021048	0.0003740	0.0978554	0.85007266	21.2518
OMP+MPI	2	4	0.1029485	0.0020851	0.0004489	0.1255782	0.66240956	8.2801
OMP+MPI	2	8	0.1477870	0.0067290	0.0003857	0.1813605	0.45866768	2.8667
OMP+MPI	4	1	0.0721363	0.0003908	0.0005428	0.0827666	1.00504551	25.1261
OMP+MPI	4	2	0.0752334	0.0023921	0.0005828	0.0920681	0.9035073	11.2938
OMP+MPI	4	4	0.0970919	0.0028469	0.0005104	0.1208261	0.68846218	4.3029
OMP+MPI	4	8	0.1456862	0.0052924	0.0004709	0.1771608	0.46954067	1.4673
OMP+MPI	8	1	0.0705185	0.0003855	0.0009246	0.0825950	1.0071336	12.5892
OMP+MPI	8	2	0.0773490	0.0016779	0.0007540	0.0949637	0.87595787	5.4747
OMP+MPI	8	4	0.0977051	0.0031459	0.0006833	0.1217164	0.68342639	2.1357
OMP+MPI	8	8	0.1435900	0.0071893	0.0006379	0.1773444	0.46905456	0.7329
OMP+MPI	16	1	0.0719853	0.0003972	0.0014275	0.0846617	0.98254819	6.1409
OMP+MPI	16	2	0.0796819	0.0016742	0.0011849	0.0978365	0.85023687	2.657
OMP+MPI	16	4	0.0954338	0.0043120	0.0009275	0.1188852	0.6997019	1.0933
OMP+MPI	16	8	0.1368859	0.0077004	0.0011075	0.1734447	0.4796007	0.3747

Figure 3.3: Table 250000 nodes

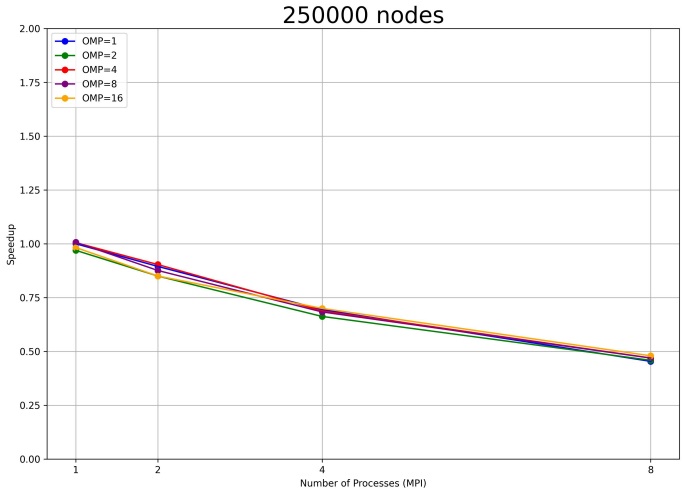

Figure 3.4: Plot 250000 nodes

In this second case, as the number of nodes increases, we see that the curves tend to become practically the same. Indeed, for each curve, the trend decreases as the number of MPI processes increases and rises again when OMP=1. This suggests that the communication overhead grows more rapidly than the benefits of parallelization as the number of MPI processes increases. The trend of the speedup curves is almost the same, suggesting consistency in the program's behavior for various configurations.Efficiency is lower when using multiple MPI processes and/or OpenMP threads, indicating that parallelization is not fully efficient in these configurations.

3.1.3 3500000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.966194	0.0000000	0.0000022	3.966566	-	-
OMP+MPI	1	1	3.978393	0.0062009	0.0000334	4.365535	1	100
OMP+MPI	1	2	4.473463	0.0120807	0.0000354	4.853790	0.89940742	44.9704
OMP+MPI	1	4	4.926431	0.0236574	0.0021060	5.335509	0.81820406	20.4551
OMP+MPI	1	8	6.311918	0.0250635	0.0008629	6.834380	0.63876092	7.9845
OMP+MPI	2	1	4.041840	0.0056711	0.0004588	4.418995	0.98790225	49.3951
OMP+MPI	2	2	4.428832	0.0072417	0.0003355	4.807928	0.90798671	22.6997
OMP+MPI	2	4	5.057320	0.0211367	0.0026066	5.485686	0.7958047	9.9476
OMP+MPI	2	8	6.353835	0.0330376	0.0020533	6.888689	0.63372512	3.9608
OMP+MPI	4	1	4.219114	0.0056691	0.0005750	4.609032	0.94716964	23.6792
OMP+MPI	4	2	4.518691	0.0065173	0.0004499	4.900653	0.89080687	11.1351
OMP+MPI	4	4	4.993151	0.0210282	0.0022267	5.399859	0.80845358	5.0528
OMP+MPI	4	8	6.239223	0.0235157	0.0010149	6.744687	0.64725545	2.0227
OMP+MPI	8	1	4.046847	0.0057801	0.0008756	4.427427	0.9860208	12.3253
OMP+MPI	8	2	4.508396	0.0072402	0.0008353	4.884095	0.89382689	5.5864
OMP+MPI	8	4	4.944506	0.0188690	0.0026507	5.353824	0.815405	2.5481
OMP+MPI	8	8	6.271818	0.0182431	0.0017194	6.777915	0.6440823	1.0064
OMP+MPI	16	1	3.959498	0.0056676	0.0011828	4.334059	1.00726252	6.2954
OMP+MPI	16	2	4.369408	0.0067158	0.0013056	4.759072	0.91730806	2.8666
OMP+MPI	16	4	5.069139	0.0129501	0.0012302	5.465558	0.79873545	1.248
OMP+MPI	16	8	6.254070	0.0255705	0.0011756	6.797475	0.64222895	0.5017

Figure 3.5: Table 3500000 nodes

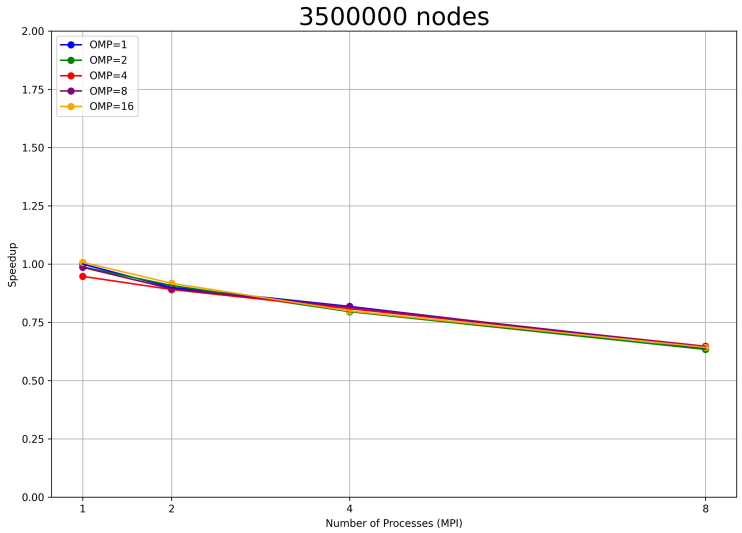

Figure 3.6: Plot 3500000 nodes

In this third case, with an increase in the number of nodes, we can observe more significant differences in the table, especially when MPI=1. It is noticeable that the performance is indeed faster compared to the single-core, single-thread scenario. However, this speed improvement is not sufficient to overcome the communication overhead, which becomes more pronounced with 100,000 nodes. Similarly to the previous cases, in this scenario, the speedup decreases significantly as the number of MPI processes increases, attempting to recover when MPI returns to 1 but increases OMP. In general, we notice that as the tree nodes increase, the other versions diverging from the serial one show a rapid increase in speedup.

3.2 Optimization 1

3.2.1 10000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0014435	0.0000000	0.0000003	0.0017930	-	-
OMP+MPI	1	1	0.0013913	0.0000354	0.0000219	0.0016858	1	100
OMP+MPI	1	2	0.0014672	0.0003654	0.0001128	0.0020975	0.80371871	40.1859
OMP+MPI	1	4	0.0015675	0.0008848	0.0002320	0.0027286	0.61782599	15.4456
OMP+MPI	1	8	0.0017552	0.0018499	0.0003371	0.0038997	0.43228966	5.4036
OMP+MPI	2	1	0.0013967	0.0000337	0.0001785	0.0018569	0.90785718	45.3929
OMP+MPI	2	2	0.0014166	0.0003063	0.0002185	0.0021149	0.79710625	19.9277
OMP+MPI	2	4	0.0014385	0.0008136	0.0003213	0.0026249	0.64223399	8.0279
OMP+MPI	2	8	0.0017997	0.0017795	0.0004775	0.0039868	0.42284539	2.6428
OMP+MPI	4	1	0.0013481	0.0000333	0.0003576	0.0019642	0.85826291	21.4566
OMP+MPI	4	2	0.0014122	0.0003351	0.0003442	0.0022479	0.74994439	9.3743
OMP+MPI	4	4	0.0014183	0.0008270	0.0004053	0.0026979	0.62485637	3.9054
OMP+MPI	4	8	0.0017960	0.0017748	0.0005658	0.0040776	0.41342947	1.292
OMP+MPI	8	1	0.0013498	0.0000336	0.0005644	0.0021795	0.77348016	9.6685
OMP+MPI	8	2	0.0014433	0.0003859	0.0005073	0.0024720	0.68195793	4.2622
OMP+MPI	8	4	0.0014469	0.0009562	0.0006226	0.0030030	0.56137196	1.7543
OMP+MPI	8	8	0.0018791	0.0019386	0.0007608	0.0045051	0.37419813	0.5847
OMP+MPI	16	1	0.0013826	0.0000347	0.0008326	0.0024856	0.67822659	4.2389
OMP+MPI	16	2	0.0014312	0.0003519	0.0008398	0.0027757	0.60734229	1.8979
OMP+MPI	16	4	0.0014563	0.0009591	0.0009716	0.0033866	0.49778539	0.7778
OMP+MPI	16	8	0.0018422	0.0021614	0.0011669	0.0050682	0.33262302	0.2599

Figure 3.7: Table 10000 nodes

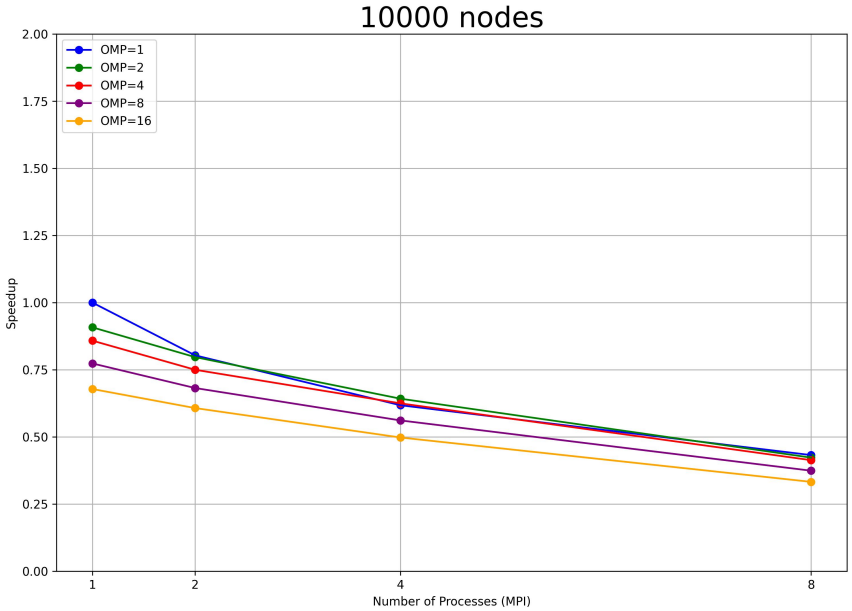

Figure 3.8: Plot 10000 nodes

In this first case, the situation is almost identical to that of optimization 0. What is significant, however, is the slight difference in the program's execution times when the number of MPI processes is not high, making it faster, especially for the single-thread single-core serial version.

3.2.2 250000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0736883	0.0000000	0.0000003	0.0739666	-	-
OMP+MPI	1	1	0.0703533	0.0004105	0.0000322	0.0796959	1	100
OMP+MPI	1	2	0.0792215	0.0016744	0.0000335	0.0933937	0.85333272	42.6666
OMP+MPI	1	4	0.1013530	0.0029569	0.0002920	0.1213901	0.65652718	16.4132
OMP+MPI	1	8	0.1477599	0.0114211	0.0002959	0.1823218	0.43711668	5.464
OMP+MPI	2	1	0.0746655	0.0004320	0.0002928	0.0856927	0.93001971	46.501
OMP+MPI	2	2	0.0834107	0.0022547	0.0003197	0.0982192	0.81140856	20.2852
OMP+MPI	2	4	0.1015025	0.0055697	0.0004026	0.1240299	0.64255393	8.0319
OMP+MPI	2	8	0.1482126	0.0064554	0.0003739	0.1789061	0.44546217	2.7841
OMP+MPI	4	1	0.0711299	0.0004226	0.0006240	0.0811847	0.98166157	24.5415
OMP+MPI	4	2	0.0808117	0.0015163	0.0005034	0.0947023	0.84154134	10.5193
OMP+MPI	4	4	0.1045633	0.0030490	0.0005108	0.1243838	0.64072572	4.0045
OMP+MPI	4	8	0.1459056	0.0104378	0.0004842	0.1805499	0.4414065	1.3794
OMP+MPI	8	1	0.0706778	0.0004136	0.0009403	0.0811936	0.98155396	12.2694
OMP+MPI	8	2	0.0764199	0.0016796	0.0007859	0.0909649	0.87611705	5.4757
OMP+MPI	8	4	0.1014878	0.0043648	0.0005946	0.1222553	0.65188094	2.0371
OMP+MPI	8	8	0.1481438	0.0098592	0.0006315	0.1807304	0.44096566	0.689
OMP+MPI	16	1	0.0712899	0.0004320	0.0013943	0.0827082	0.96357919	6.0224
OMP+MPI	16	2	0.0786560	0.0020747	0.0012457	0.0937494	0.85009504	2.6565
OMP+MPI	16	4	0.1020712	0.0040082	0.0009430	0.1238381	0.64354912	1.0055
OMP+MPI	16	8	0.1487155	0.0065582	0.0008933	0.1785521	0.44634535	0.3487

Figure 3.9: Table 250000 nodes

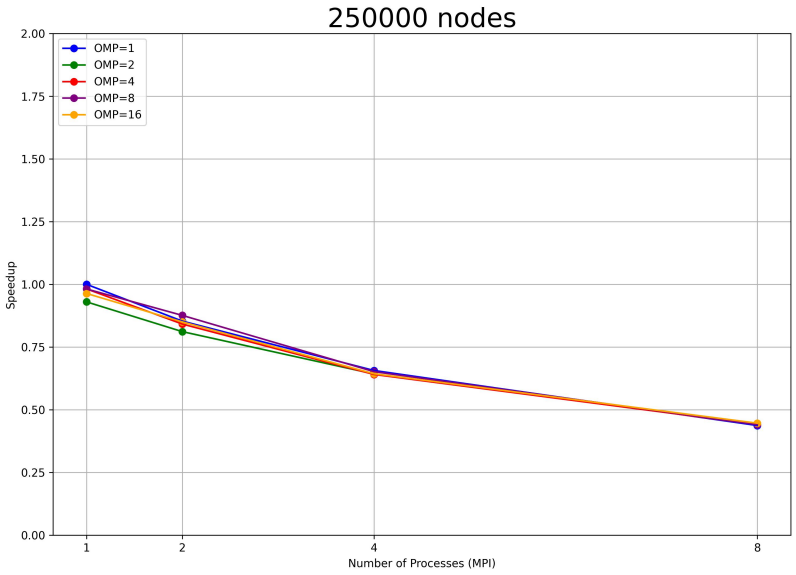

Figure 3.10: Plot 250000 nodes

In this second case, we can appreciate the effectiveness of the first level of optimization. Unlike the level zero optimization (or non-optimization), all the execution times have decreased significantly, but in a proportionate manner, making the difference not visually prominent.

3.2.3 3500000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.958478	0.0000000	0.0000016	3.958795	-	-
OMP+MPI	1	1	3.911557	0.0058597	0.0000366	4.214331	1	100
OMP+MPI	1	2	4.305267	0.0068372	0.0000284	4.598576	0.91644257	45.8221
OMP+MPI	1	4	5.033105	0.0171556	0.0012119	5.376437	0.783852	19.5963
OMP+MPI	1	8	6.389190	0.0171219	0.0008769	6.857594	0.6145495	7.6819
OMP+MPI	2	1	4.164600	0.0060910	0.0003915	4.468348	0.94315184	47.1576
OMP+MPI	2	2	4.577458	0.0065230	0.0002901	4.893927	0.86113478	21.5284
OMP+MPI	2	4	5.167959	0.0183741	0.0027981	5.518526	0.76366972	9.5459
OMP+MPI	2	8	6.500915	0.0215571	0.0008727	6.951219	0.60627226	3.7892
OMP+MPI	4	1	3.939778	0.0058845	0.0006327	4.233580	0.99545337	24.8863
OMP+MPI	4	2	4.461494	0.0072207	0.0004311	4.760218	0.8853232	11.0665
OMP+MPI	4	4	4.986765	0.0112809	0.0010983	5.303960	0.79456312	4.966
OMP+MPI	4	8	6.541566	0.0169567	0.0007430	7.038832	0.59872588	1.871
OMP+MPI	8	1	4.211925	0.0059924	0.0009683	4.516538	0.93308886	11.6636
OMP+MPI	8	2	4.580747	0.0068123	0.0006933	4.888444	0.86210067	5.3881
OMP+MPI	8	4	5.069925	0.0123285	0.0021166	5.410226	0.77895651	2.4342
OMP+MPI	8	8	6.584007	0.0284783	0.0033003	7.078744	0.59535012	0.9302
OMP+MPI	16	1	4.356607	0.0061070	0.0015144	4.671092	0.90221535	5.6388
OMP+MPI	16	2	4.885468	0.0089861	0.0011100	5.196779	0.81095058	2.5342
OMP+MPI	16	4	5.335989	0.0140981	0.0012701	5.678359	0.74217416	1.1596
OMP+MPI	16	8	6.506946	0.0202694	0.0030273	7.000184	0.60203145	0.4703

Figure 3.11: Table 3500000 nodes

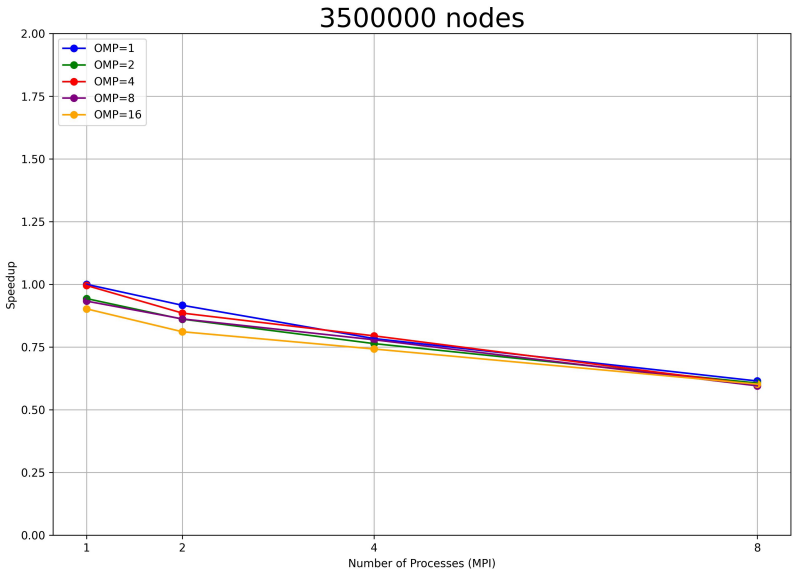

Figure 3.12: Plot 3500000 nodes

Similar to the level zero optimization but with the benefits of the first-level optimization. Here, too, the overall times have decreased but in a proportionate manner. What makes the difference compared to the previous case with 250000 nodes is that here the speedup of each OMP-MPI configuration is always lower than the case of optimization 0. In conclusion, this level of optimization also does not enhance the effectiveness of parallelization.

3.3 Optimization 2

3.3.1 10000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0010798	0.0000000	0.0000001	0.0012924	-	-
OMP+MPI	1	1	0.0010783	0.0000335	0.0000202	0.0013214	1	100
OMP+MPI	1	2	0.0011016	0.0003023	0.0000872	0.0016253	0.81301913	40.651
OMP+MPI	1	4	0.0010820	0.0008160	0.0002090	0.0021078	0.62690957	15.6727
OMP+MPI	1	8	0.0012803	0.0015458	0.0003117	0.0030479	0.43354441	5.4193
OMP+MPI	2	1	0.0010427	0.0000326	0.0001684	0.0014264	0.92638811	46.3194
OMP+MPI	2	2	0.0011059	0.0003808	0.0002502	0.0018199	0.72608385	18.1521
OMP+MPI	2	4	0.0011047	0.0006970	0.0002856	0.0021023	0.62854968	7.8569
OMP+MPI	2	8	0.0013630	0.0016666	0.0004634	0.0034045	0.38813335	2.4258
OMP+MPI	4	1	0.0010742	0.0000327	0.0003069	0.0016187	0.8163341	20.4084
OMP+MPI	4	2	0.0011446	0.0003293	0.0003291	0.0019008	0.69518098	8.6898
OMP+MPI	4	4	0.0011593	0.0007979	0.0003852	0.0023634	0.55910976	3.4944
OMP+MPI	4	8	0.0014424	0.0017887	0.0005295	0.0036317	0.36385164	1.137
OMP+MPI	8	1	0.0010658	0.0000335	0.0004905	0.0017802	0.74227615	9.2785
OMP+MPI	8	2	0.0011076	0.0003084	0.0005030	0.0020235	0.65302693	4.0814
OMP+MPI	8	4	0.0011349	0.0007394	0.0005322	0.0024415	0.54122466	1.6913
OMP+MPI	8	8	0.0013759	0.0019081	0.0007202	0.0039131	0.33768623	0.5276
OMP+MPI	16	1	0.0010573	0.0000323	0.0008534	0.0021281	0.62092947	3.8808
OMP+MPI	16	2	0.0010917	0.0002823	0.0008608	0.0023705	0.55743514	1.742
OMP+MPI	16	4	0.0011107	0.0008270	0.0008825	0.0028198	0.4686148	0.7322
OMP+MPI	16	8	0.0012247	0.0020443	0.0010828	0.0042129	0.31365568	0.245

Figure 3.13: Table 10000 nodes

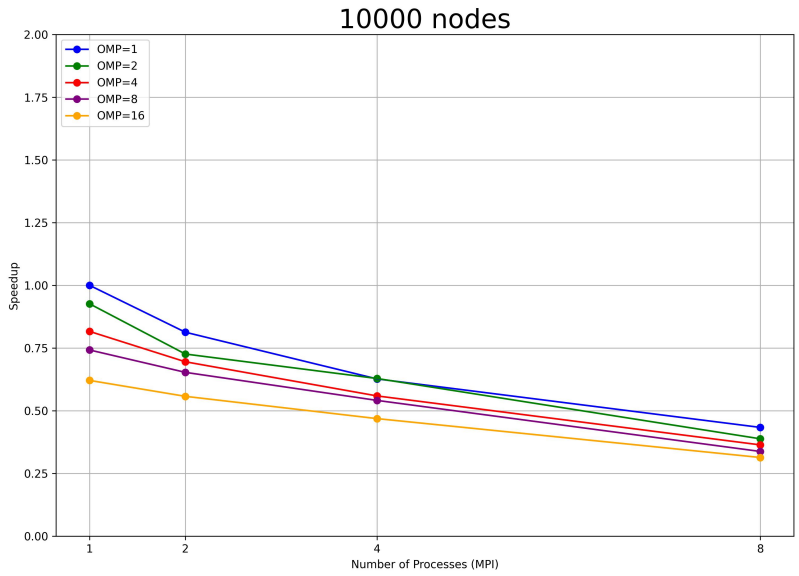

Figure 3.14: Plot 10000 nodes

With optimization level 2 and a small number of tree nodes, we observe the effectiveness of parallelization. It is certainly more efficient than the previous levels, and indeed, it consistently exhibits lower times compared to levels 1 and, especially, level 0. This is due to the fact that this level strikes the right balance between performance and compilation time. It incorporates more aggressive optimizations than O1 but avoids some of the more time-consuming optimizations present in O3. In fact, what happens is that parallelization benefits greatly, causing a greater time difference in the non-serial versions and bringing them closer to the serial version, which is the ideal one.

3.3.2 250000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0652465	0.0000000	0.0000003	0.0655422	-	-
OMP+MPI	1	1	0.0583194	0.0004168	0.0000318	0.0659851	1	100
OMP+MPI	1	2	0.0698487	0.0016230	0.0000496	0.0814735	0.80989647	40.4948
OMP+MPI	1	4	0.0897549	0.0045310	0.0002663	0.1071951	0.61556079	15.389
OMP+MPI	1	8	0.1415608	0.0077827	0.0003057	0.1665021	0.39630191	4.9538
OMP+MPI	2	1	0.0618930	0.0004019	0.0002992	0.0702448	0.93935921	46.968
OMP+MPI	2	2	0.0741312	0.0017585	0.0003451	0.0861594	0.76584911	19.1462
OMP+MPI	2	4	0.0937335	0.0029962	0.0004510	0.1106163	0.59652239	7.4565
OMP+MPI	2	8	0.1406484	0.0065816	0.0003880	0.1645392	0.40102966	2.5064
OMP+MPI	4	1	0.0657958	0.0004404	0.0005130	0.0742321	0.88890251	22.2226
OMP+MPI	4	2	0.0799098	0.0024211	0.0004555	0.0928558	0.71061905	8.8827
OMP+MPI	4	4	0.1020381	0.0061030	0.0004772	0.1214449	0.54333364	3.3958
OMP+MPI	4	8	0.1467717	0.0074413	0.0004663	0.1747557	0.37758482	1.18
OMP+MPI	8	1	0.0664278	0.0004167	0.0008719	0.0761716	0.86626906	10.8284
OMP+MPI	8	2	0.0739855	0.0024456	0.0007401	0.0867623	0.76052733	4.7533
OMP+MPI	8	4	0.0986037	0.0052201	0.0006393	0.1169229	0.5643471	1.7636
OMP+MPI	8	8	0.1456436	0.0069075	0.0006494	0.1692192	0.38993861	0.6093
OMP+MPI	16	1	0.0669110	0.0004157	0.0012209	0.0759173	0.8691708	5.4323
OMP+MPI	16	2	0.0745568	0.0013563	0.0010849	0.0878565	0.75105541	2.347
OMP+MPI	16	4	0.0965480	0.0041101	0.0009061	0.1136643	0.58052616	0.9071
OMP+MPI	16	8	0.1434138	0.0087510	0.0009269	0.1709338	0.38602722	0.3016

Figure 3.15: Table 250000 nodes

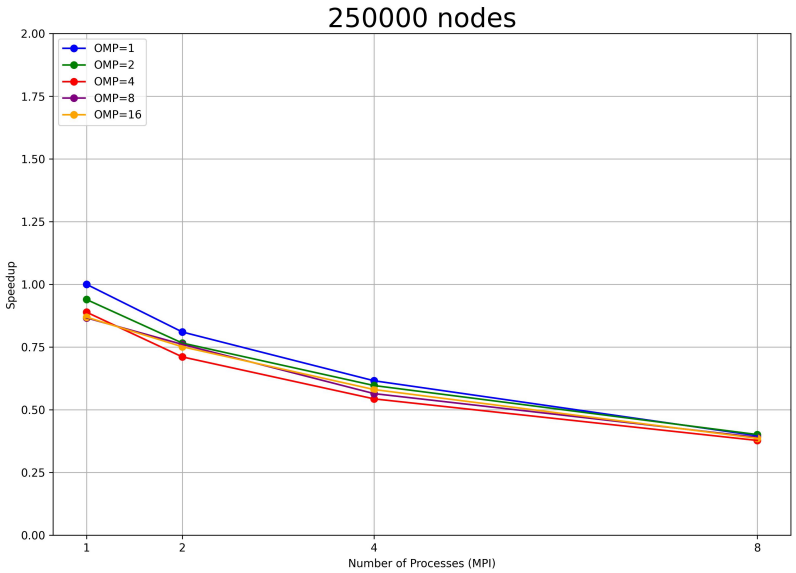

Figure 3.16: Plot 250000 nodes

A similar reasoning can be applied to this second case. The speedup decreases slowly as the number of processes increases, but more significantly compared to the previous optimization levels. The times, which benefit from optimization, consistently turn out to be the lowest among all three levels.

3.3.3 3500000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	4.166392	0.0000000	0.0000017	4.166676	-	-
OMP+MPI	1	1	3.941774	0.0057841	0.0000318	4.130461	1	100
OMP+MPI	1	2	4.354343	0.0064748	0.0000290	4.548042	0.90818439	45.4092
OMP+MPI	1	4	4.832301	0.0163627	0.0014344	5.064165	0.81562525	20.3906
OMP+MPI	1	8	6.372189	0.0250011	0.0006090	6.705569	0.61597473	7.6997
OMP+MPI	2	1	3.831802	0.0057293	0.0003623	4.017488	1.02812033	51.406
OMP+MPI	2	2	4.269732	0.0062522	0.0003149	4.466544	0.92475545	23.1189
OMP+MPI	2	4	4.921198	0.0106560	0.0006681	5.149241	0.8021495	10.0269
OMP+MPI	2	8	6.221249	0.0158205	0.0008486	6.554065	0.63021367	3.9388
OMP+MPI	4	1	4.026354	0.0057376	0.0006281	4.217265	0.97941697	24.4854
OMP+MPI	4	2	4.318110	0.0065454	0.0005149	4.513756	0.91508292	11.4385
OMP+MPI	4	4	4.939134	0.0132641	0.0019256	5.176357	0.79794749	4.9872
OMP+MPI	4	8	6.323778	0.0186017	0.0007951	6.650599	0.62106603	1.9408
OMP+MPI	8	1	4.030303	0.0057146	0.0010089	4.220371	0.9786963	12.2337
OMP+MPI	8	2	4.377582	0.0068680	0.0007907	4.573951	0.90304014	5.644
OMP+MPI	8	4	4.873911	0.0180221	0.0014680	5.112730	0.80787789	2.5246
OMP+MPI	8	8	6.332064	0.0156184	0.0008850	6.658203	0.62035677	0.9693
OMP+MPI	16	1	3.886028	0.0058452	0.0015930	4.076416	1.01325787	6.3329
OMP+MPI	16	2	4.310958	0.0062867	0.0011203	4.504649	0.91693297	2.8654
OMP+MPI	16	4	4.877561	0.0105531	0.0011369	5.106695	0.80883246	1.2638
OMP+MPI	16	8	6.284026	0.0198621	0.0039264	6.620932	0.62384887	0.4874

Figure 3.17: Table 3500000 nodes

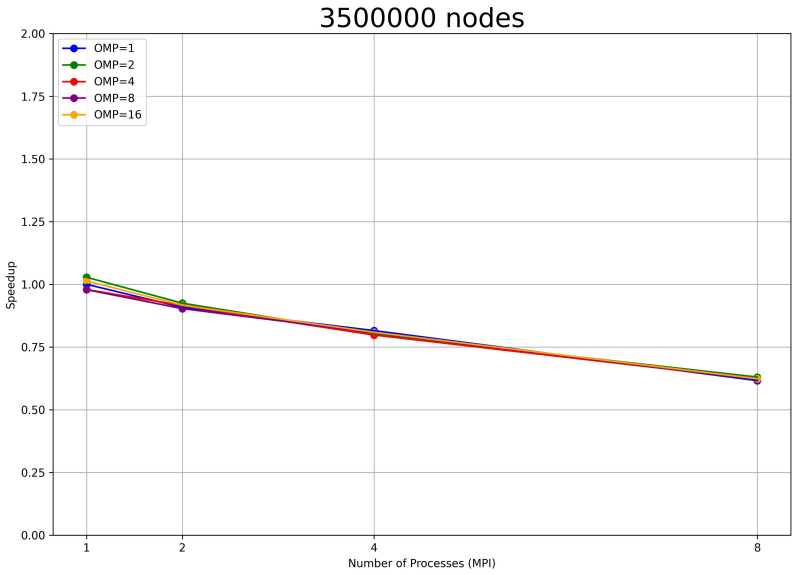

Figure 3.18: Plot 3500000 nodes

In this third case, the situation is almost the same with the usual improvements due to the level two optimization. However, we observe something peculiar that we hadn't seen until now: when the number of OpenMP threads and, especially, MPI processes is low, the speedup tends to rise and surpass the speedup of the serial version. This is the case for OMP=2, 16 and MPI=1, but the difference compared to the single-thread, single-core version is so small that it can be easily identified as noise. In conclusion, from this optimization level, we have experienced benefits in terms of execution speed but not in terms of speedup and efficiency.

3.4 Optimization 3

3.4.1 10000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0012703	0.0000000	0.0000004	0.0014777	-	-
OMP+MPI	1	1	0.0012089	0.0000307	0.0000200	0.0014461	1	100
OMP+MPI	1	2	0.0013117	0.0003247	0.0000884	0.0018626	0.77638784	38.8194
OMP+MPI	1	4	0.0013093	0.0006549	0.0002009	0.0021846	0.66195184	16.5488
OMP+MPI	1	8	0.0015091	0.0015690	0.0002969	0.0033172	0.43593995	5.4492
OMP+MPI	2	1	0.0012144	0.0000311	0.0001684	0.0016047	0.90116533	45.0583
OMP+MPI	2	2	0.0012704	0.0003352	0.0002412	0.0019615	0.73724191	18.431
OMP+MPI	2	4	0.0012922	0.0007065	0.0002807	0.0023112	0.62569228	7.8212
OMP+MPI	2	8	0.0014443	0.0016424	0.0004148	0.0033957	0.42586212	2.6616
OMP+MPI	4	1	0.0011867	0.0000313	0.0003675	0.0017664	0.81867074	20.4668
OMP+MPI	4	2	0.0012980	0.0003265	0.0003545	0.0020837	0.69400585	8.6751
OMP+MPI	4	4	0.0012922	0.0007358	0.0003781	0.0024197	0.59763607	3.7352
OMP+MPI	4	8	0.0015942	0.0017202	0.0005101	0.0037472	0.38591482	1.206
OMP+MPI	8	1	0.0012244	0.0000315	0.0005781	0.0020276	0.71320773	8.9151
OMP+MPI	8	2	0.0012586	0.0002960	0.0005442	0.0022091	0.65461047	4.0913
OMP+MPI	8	4	0.0013000	0.0008268	0.0005672	0.0026921	0.5371643	1.6786
OMP+MPI	8	8	0.0015296	0.0019493	0.0007318	0.0040959	0.35306038	0.5517
OMP+MPI	16	1	0.0012249	0.0000319	0.0008314	0.0022823	0.63361521	3.9601
OMP+MPI	16	2	0.0012558	0.0003284	0.0008841	0.0025756	0.56146141	1.7546
OMP+MPI	16	4	0.0012956	0.0008597	0.0009023	0.0030841	0.46888882	0.7326
OMP+MPI	16	8	0.0015772	0.0018588	0.0010679	0.0044161	0.32746088	0.2558

Figure 3.19: Table 10000 nodes

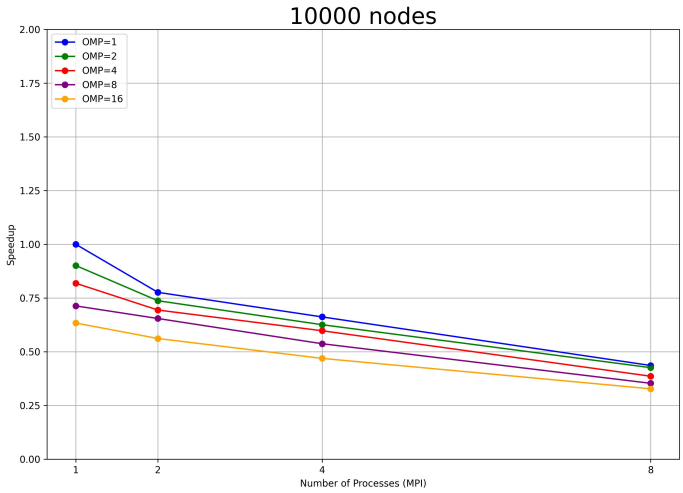

Figure 3.20: Plot 10000 nodes

From the analysis of this case study on the third level of optimization, it is immediately apparent that, compared to the second optimization level, it is less efficient. Indeed, in this case, the times are higher than the second optimization level, almost on par with the first optimization level. This is because, despite employing all the optimizations of -O2, what could cause this level is the introduction of more aggressive optimizations, including more aggressive function inlining, CPU register utilization, and other advanced code transformations. All of this could lead to a significant increase in compilation time and might result in unexpected outcomes or even a slowdown in performance due to phenomena such as increased memory consumption. Therefore, we observe that as the number of processes increases, all curves tend to decrease in both speedup and efficiency.

3.4.2 250000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0669427	0.0000000	0.0000002	0.0672165	-	-
OMP+MPI	1	1	0.0663788	0.0004062	0.0000318	0.0744032	1	100
OMP+MPI	1	2	0.0768200	0.0015608	0.0000300	0.0892514	0.83363622	41.6818
OMP+MPI	1	4	0.0947868	0.0052148	0.0003595	0.1124606	0.66159348	16.5398
OMP+MPI	1	8	0.1493323	0.0084538	0.0002988	0.1756651	0.42355141	5.2944
OMP+MPI	2	1	0.0673457	0.0003916	0.0003672	0.0751980	0.98943057	49.4715
OMP+MPI	2	2	0.0779582	0.0020738	0.0003835	0.0899109	0.82752147	20.688
OMP+MPI	2	4	0.0951427	0.0043621	0.0003844	0.1131785	0.65739694	8.2175
OMP+MPI	2	8	0.1453004	0.0051534	0.0003839	0.1686827	0.44108376	2.7568
OMP+MPI	4	1	0.0668543	0.0003946	0.0006355	0.0749703	0.99243567	24.8109
OMP+MPI	4	2	0.0755049	0.0013440	0.0005293	0.0872454	0.8528037	10.66
OMP+MPI	4	4	0.0976350	0.0041611	0.0005276	0.1160032	0.6413892	4.0087
OMP+MPI	4	8	0.1416663	0.0063412	0.0004265	0.1665763	0.44666138	1.3958
OMP+MPI	8	1	0.0673899	0.0004097	0.0007671	0.0760568	0.97825835	12.2282
OMP+MPI	8	2	0.0767817	0.0026647	0.0007774	0.0894910	0.83140428	5.1963
OMP+MPI	8	4	0.0953542	0.0035756	0.0005957	0.1126676	0.66037796	2.0637
OMP+MPI	8	8	0.1423248	0.0080660	0.0005765	0.1682585	0.44219579	0.6909
OMP+MPI	16	1	0.0729542	0.0003972	0.0010265	0.0820952	0.9063039	5.6644
OMP+MPI	16	2	0.0783451	0.0018442	0.0010853	0.0915100	0.81306087	2.5408
OMP+MPI	16	4	0.1008195	0.0029317	0.0009188	0.1181424	0.62977559	0.984
OMP+MPI	16	8	0.1402159	0.0055732	0.0008870	0.1651530	0.45051074	0.352

Figure 3.21: Table 250000 nodes

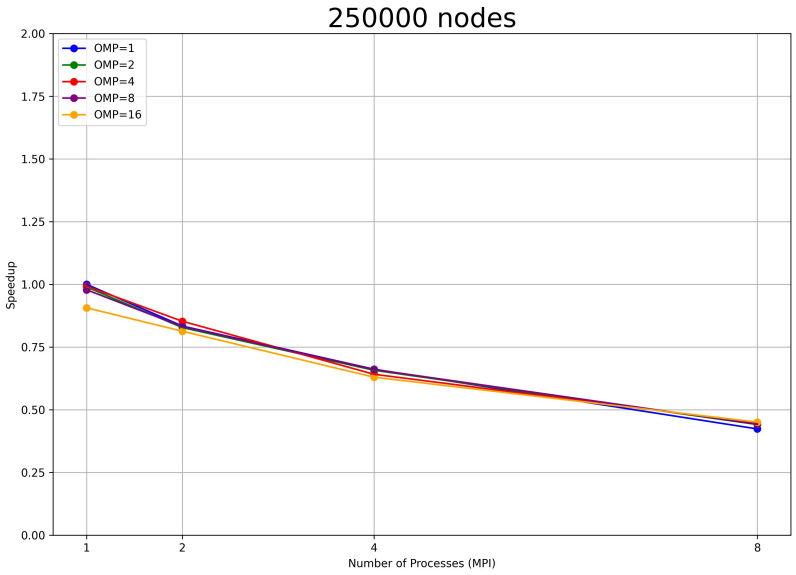

Figure 3.22: Plot 250000 nodes

In this second case, as the number of nodes increases, we observe that this optimization level not only leads to higher speedup, efficiency, and times compared to the second level but is actually worse than the first level. This is caused precisely by what we mentioned earlier, especially regarding memory consumption, which is exacerbated here by the quantity of nodes under consideration.

3.4.3 3500000 Nodes

Modality	OMP	MPI	tree_creation_time	communication_time	rb_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.921369	0.0000000	0.0000018	3.921662	-	-
OMP+MPI	1	1	3.911008	0.0060086	0.0000344	4.106331	1	100
OMP+MPI	1	2	4.690635	0.0067413	0.0000383	4.904064	0.8373323	41.8666
OMP+MPI	1	4	5.068678	0.0135818	0.0008959	5.297011	0.77521674	19.3804
OMP+MPI	1	8	6.299449	0.0281296	0.0011791	6.657487	0.61679907	7.71
OMP+MPI	2	1	3.992213	0.0056697	0.0003859	4.177122	0.98305278	49.1526
OMP+MPI	2	2	4.359688	0.0066519	0.0003193	4.553748	0.90174756	22.5437
OMP+MPI	2	4	5.019996	0.0103071	0.0006213	5.263473	0.78015631	9.752
OMP+MPI	2	8	6.225472	0.0163126	0.0007512	6.562797	0.62569835	3.9106
OMP+MPI	4	1	4.033894	0.0057303	0.0006245	4.222294	0.97253573	24.3134
OMP+MPI	4	2	4.346120	0.0064283	0.0004529	4.539560	0.90456603	11.3071
OMP+MPI	4	4	4.835772	0.0152252	0.0013116	5.066027	0.81056251	5.066
OMP+MPI	4	8	6.201865	0.0250435	0.0009579	6.539434	0.62793374	1.9623
OMP+MPI	8	1	3.941233	0.0056971	0.0011626	4.128301	0.99467825	12.4335
OMP+MPI	8	2	4.452086	0.0087602	0.0008203	4.650603	0.88296747	5.5185
OMP+MPI	8	4	4.898325	0.0144478	0.0013269	5.128707	0.8006563	2.5021
OMP+MPI	8	8	6.219279	0.0289944	0.0015816	6.552047	0.626725	0.9793
OMP+MPI	16	1	3.839197	0.0057733	0.0015821	4.026876	1.01973135	6.3733
OMP+MPI	16	2	4.250647	0.0064026	0.0009776	4.446121	0.92357613	2.8862
OMP+MPI	16	4	4.769162	0.0102143	0.0010948	4.999316	0.82137865	1.2834
OMP+MPI	16	8	6.244675	0.0235148	0.0014494	6.588710	0.62323753	0.4869

Figure 3.23: Table 3500000 nodes

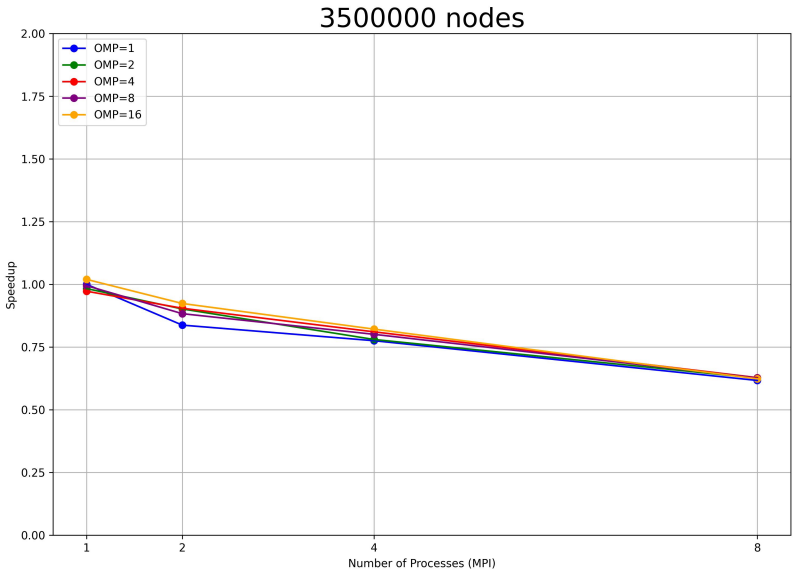

Figure 3.24: Plot 3500000 nodes

In this third and final level, as the number of nodes increases, the trend is the same as in the previous optimizations, where the execution time gradually increases as the number of MPI processes grows, always due to overhead, but increases slowly as the OMP threads increase. Ultimately, we understand that this optimization level, which was expected to be the best, has turned out to be the worst in terms of trade-offs, unexpectedly degrading overall performance.

3.5 Final Considerations

After carefully analyzing all types of optimizations with a significant difference across various problem sizes, it is evident that parallelization carried out using the OMP+MPI approach does not yield benefits in the program's execution. Indeed, it is observed that for every chosen configuration of OMP and MPI, none of them outperforms the serial single-thread single-core version.
Performance rapidly degrades with the increase in the number of OMP threads, but especially with the increase in the number of MPI processes. This is because, as the problem size and cores increase, the overhead caused by communication between processes becomes increasingly prominent.
It is clear that the serial version proves to be the best, primarily because the search in a tree or, as the approach was later modified, the binary search on an array, is already an optimized algorithm with a complexity of O(log(n)), as seen earlier. The performance degradation due to parallelization is not so much accentuated by the tree creation time, which remains practically the same, nor too much on the search time, but rather by the communication time. This degradation is exacerbated by the presence of just one additional OMP thread or MPI process, negatively impacting performance.

Chapter 4

OMP+CUDA Parallelization

CUDA, or Compute Unified Device Architecture, is a parallel computing platform and application programming interface model created by NVIDIA. It is designed to harness the power of Graphics Processing Units (GPUs) for general-purpose processing, enabling significant acceleration of various computational workloads. At its core, CUDA provides a programming model that allows developers to offload parallelizable tasks to the GPU, taking advantage of the parallel processing capabilities inherent in these devices. This is particularly useful for computationally intensive applications such as scientific simulations, image and signal processing, machine learning, and more. One of the main features of CUDA is that it allows developers to divide their tasks into parallel threads that can be executed concurrently on the GPU. This is achieved through the use of CUDA kernels, which are functions that are executed in parallel on the GPU. The GPU consists of numerous CUDA cores, which are small processing units that work in parallel to execute the tasks specified in the CUDA kernels. This parallelism significantly accelerates computation compared to traditional sequential execution on CPUs.

4.1 GPU Specifications

For the parallelized version with OMP and CUDA, I used the assistance of **Google Colaboratory**, a free cloud-based service provided by Google that offers a Jupyter notebook development environment with free access to GPU-based computing resources. It provides free access to GPUs, which is particularly useful for performing intensive calculations, such as those required in machine learning and deep learning.Colab features an **NVIDIA TESLA T4** GPU, which is a graphics card based on the NVIDIA Turing architecture.

Max regs number/SM	: 65535
GPU Name	: Tesla T4
CUDA Cores	: 40
Global Memory	: 14.75 GB
Shared Memory/block	: 48.00 KB
Max Threads/block	: 1024
Max Blocks/SM	: 16
Max Blocks Dim.	: 1024 x 1024 x 64
Max Grid Dim.	: 2147483647 x 65535 x 65535
Max SMs	: 40
Max Thread Grid Dim.	: 2147483647 x 65535 x 65535
Max Thread Block Dim.	: 1024 x 1024 x 64
Kernel Exec. Capacity	: 7.5

The specifications of Colab Tesla T4 GPU are obtained executing this code that uses specific methods from the library **cuda_runtime.h**:

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    int device = 0;
    cudaSetDevice(device);

    struct cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);

    printf("Registers per SM: %d\n", deviceProp.regsPerMultiprocessor);
    printf("Max blocks per SM: %d\n", deviceProp.maxThreadsPerMultiProcessor /
        deviceProp.maxThreadsPerBlock);
    printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);
    printf("GPU Name: %s\n", deviceProp.name);
    printf("Number of CUDA Cores: %d\n", deviceProp.multiProcessorCount);
    printf("Base Clock Rate: %.2f GHz\n", deviceProp.clockRate / 1000.0);
    printf("Available Global Memory: %.2f GB\n",
        static_cast<float>(deviceProp.totalGlobalMem) / (1024 * 1024 * 1024));
    printf("Shared Memory per block: %.2f KB\n",
        static_cast<float>(deviceProp.sharedMemPerBlock) / 1024);
    printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);
    printf("Max block dimensions (x, y, z): %d x %d x %d\n", deviceProp.maxThreadsDim[0],
        deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);
    printf("Max grid dimensions (x, y, z): %d x %d x %d\n", deviceProp.maxGridSize[0],
        deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);
    printf("Max SM (Streaming Multiprocessors): %d\n", deviceProp.multiProcessorCount);
    printf("Max thread grid dimensions: %d x %d x %d\n", deviceProp.maxGridSize[0],
        deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);
    printf("Max thread block dimensions: %d x %d x %d\n", deviceProp.maxThreadsDim[0],
        deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);
    printf("Kernel execution capability: %d.%d\n", deviceProp.major, deviceProp.minor);

    return 0;
}
```

4.2 OMP+CUDA approach

It might be thought that using OMP and CUDA together is not possible, even though OMP is designed primarily for parallelizing code on multicore CPUs, and CUDA is designed for parallelizing on the GPU. However, it is precisely on these distinctive characteristics that the combined approach will be developed.
The idea is to use the same approach designed for OMP+MPI but implemented slightly differently. The size of the problem, in this case, the number of nodes in a binary tree, will be divided in two: the first half will remain on the CPU, and OMP threads will handle parallelization on the host, while the second half will be sent to the device, which will work in parallel exploiting the capabilities of the kernel on the GPU.
Unlike the OMP+MPI approach, there won't be any process dividing the workload here, but it will be divided before the GPU and CPU work independently and operate the search on the assigned halves. To achieve this, we need to use CUDA APIs such as cudaMalloc(), which, when executed on the host, allocates a portion of memory based on the working data structure, depending on the quantity of elements that will be addressed to the GPU.
Here is the function signature:

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

Additionally, we require cudaMemcpy() to copy half of the elements from the host to a certain data structure, in another structure allocated with cudaMalloc(). In this case, as with the previous approach, we will insert all the tree nodes into a sorted array, and the idea is still to perform not a search on a tree but the classic binary search.

Here's the function signature:

```
cudaError_t cudaMemcpy(void* dst, const void* src,size_t count, cudaMemcpyKind kind);
```

The situation here, however, is a bit more delicate compared to the OMP+MPI approach. While with OMP, we can apply the same methodology used for combined parallelization with MPI, with CUDA, we need to slightly modify the approach because CUDA threads come into play, participating in parallelization on the GPU.
The CUDA threads are organized in blocks executed by **Streaming multiprocessors**. Each GPU has its configuration and its maximum streaming multiprocessors simultaneously active on GPU and consequently the maximum amount of blocks assignable for each streaming multiprocessor.
Since we want to assign to each thread a portion of elements to perform the binary search we need to know the max number of thread for each block assignable for a single streaming multiprocessor as well known as **BlockSize**.
The block size, denoted by **blockDim** represents the number of threads within a block. Threads within a block can cooperate and share data using fast on-chip shared memory. A block is the basic unit of execution on a GPU, and all threads within a block can synchronize and communicate with each other.
Associated with the block size, there is the **GridSize** denoted by **gridDim** represents the number of blocks in the grid. The grid is the overall set of blocks that execute a kernel on the GPU. The combination of block size and grid size determines the total number of threads launched on the GPU.
The gridsize, along with the block size, infleunces the overall parallelism and scalability of a CUDA program. Both blocks and grid can work up to three dimensions specifying the **dim3** type when declaring blockSize and gridSize.
We can determine the gridSize as follows:

```
dim gridSize((N-1)/blockSize.x + 1)
```

Since the gridSize depends on the blockSize, it is more important understand how many threads we can set into a block, so it is important to set the blockSize. In the Colab Tesla T4 GPU the maximum number of registers per SM is 65535 so this means that the threads within the block in the SM can use a maximum of this number of registers. To determine the maximum number of registers that a single thread uses to execute the kernel, we need to use a compilation option, namely **−ptxas-options=v**.
Analyzing the kernel, each individual thread uses a maximum of 10 registers. Therefore, if we calculate a simple ratio, namely 65535/10, we would find that there should be approximately 6500 threads in each block. However, this is not feasible because the maximum number of threads is indeed 1024, and it is also not possible to use more blocks per SM because there can be at most 1 block per SM. As a result, we choose the maximum size for the blockSize, so **blockSize = 1024**.
Furthermore, by using this calculatorand inputting the correct information derived from the characteristics of the GPU mentioned above and the number of registers each thread uses to execute the kernel, we see that the GPU is fully utilized, reaching 100% of its potential.

4.2.1 CUDA Kernel

After determining the maximum number of threads that can work simultaneously on a single streaming multiprocessor to execute the kernel, let's return to the discussion of fair element distribution. Since each thread should not work on a single element of the array but on multiple elements to perform the actual binary search, it is crucial to determine how many elements each thread will handle. Before delving into the actual division and before calling the kernel, it is necessary to use a scaling factor to reduce the grid size, ensuring that all 1024 threads fit within a single block. This is because, without introducing this scaling factor, each thread would operate on only one element, defeating the purpose of binary search. After this, we can call the kernel, passing a pointer to contain the result (also allocated in GPU memory using cudaMalloc()) since the kernel, by definition, returns VOID. The kernel has a section where, taking the thread index from the **int i = blockIdx.x * blockDim.x + threadIdx.x;** relationship, a certain number of elements, a starting index, and an ending index are assigned to this thread. Additionally, there are additional lines of code that further distribute the workload more evenly, as the last thread might work not only on the elements assigned to it but also on another large set of elements. By doing this, each thread, in the second half of the array passed to the GPU, will know where to start, where to end, and on how many elements to perform the binary search. In the kernel, there is then a call to the binary search function, which, if the element is found by a thread, will execute the **Atomicadd()** function, adding 1 to the result.

4.2.2 OMP approach

Since the kernel invocation is asynchronous, it means that while the GPU is being called, the CPU continues to work and execute the subsequent instructions. This situation can be leveraged by simultaneously working on the host, conducting the search on the first half of the array thanks to the parallelization achieved using OMP. Even with OMP, we perform the same data division based on the number of available OMP threads set using the **omp_set_num_threads()** function. Thus, there will be a period during which both the host and the device work in parallel by invoking the same function defined as **__host__ __device__**.
This allows it to be called by both the CPU and the GPU since they need to execute the same search. Additionally, there are no concurrency issues because the elements are read-only, and a specific quantity of elements will be seen by one and only one thread. When the host's search is successful, the thread finding the key will update the result in the **foundCPU** variable. When the kernel finishes, typically before the CPU, both the OMP threads and CUDA threads will synchronize through **cudaDeviceSynchronize()**. The kernel result will be copied to the host using cudaMemcpy() into the **foundGPU** variable.
Finally, this result will be added to foundCPU in **finalResult**, and if it is equal to 1 (or greater than one in the case of finding duplicates), then the key will have been globally found; otherwise, it will not.

4.2.3 A little optimization

Before starting with the analysis, it is necessary to make a small clarification: the code would be further optimized if, before initiating the search on both the kernel and the host, there were a check on the element to be searched. Since the array is sorted and divided in half, we could check the key to be searched with the element at position (arraysize/2) + 1 (the element at the center of the sorted array). This way, if the key is less than the central element, it is present in the first half of the array, activating only the CPU, and vice versa. It is clear that this check would be unnecessary if the element to be searched coincides with the central element of the array. However, the probability of this happening is 1/arraysize (arraysize is the dimension of the problem), and as arraysize approaches infinity, this would bring a significant optimization. In the code, this check is not performed to avoid distorting the purpose of parallelization, simultaneously utilizing both the CPU and OMP, as well as the GPU and CUDA.

4.3 Analysis

4.3.1 OMP Threads and L1 Cache

The analyses performed on OMP and CUDA will follow the same modus operandi as OMP and MPI. Therefore, tests will be conducted for all four optimization levels and on the same number of nodes chosen for OMP and MPI. However, this time, since OMP + CUDA is executed on Colab, we need to reduce the quantity of exploitable OMP threads. In particular, Colab has 2 cores and 2 logical threads, so we can use a maximum of 2 OMP threads. Still, for our experiments, we will push it to 4 OMP threads to show a potential performance degradation as we exceed the number of logical threads supported by Colab's CPU. Additionally, we will leverage a CUDA function, namely **cudaFuncSetCacheConfig()**:

```
cudaError_t cudaFuncSetCacheConfig(const void* func, enum cudaFuncCache cacheConfig);
```

This function, if the cacheConfig parameter is specified with **cudaFuncCachePreferL1** allows assigning more memory to the L1 cache present in global memory. We tell the CUDA system to prefer L1 cache as memory to optimize data access during the execution of the associated CUDA kernel. This cache is a level 1 cache (as the name suggests) and is located directly on the GPU chip. Since this cache is small, it is significantly faster, and this function allows telling the CUDA compiler to try to maximize the efficiency of using the L1 cache to store and retrieve data. It is clear, however, that the effect of cache preference depends on the specific GPU in use and the nature of the kernel.
Summarizing then, the tests will be as follows:

- OMP+CUDA - {1, 2, 4} OpenMP threads (Parallel version)
- OMP+CUDA+L1 - {1, 2, 4} OpenMP threads (Parallel version)

4.3.2 Performance, Speed-Up & Efficiency

Unlike what was done for OMP and MPI, the calculation of speedup will be based on the ratio between the execution time of the sequential algorithm and the execution time of the parallel algorithm. Instead, efficiency will be calculated as the ratio of the execution time of the sequential algorithm to the product of the execution time of the parallel algorithm and the number of OMP threads in use. Consequently, unlike before, the graphs will show the trend of the speedup for the OMP+CUDA version and OMP+CUDA+L1 version as the number of OMP threads varies.

4.3.3 Other memory types

No tests will be conducted on the other types of memory, namely shared, constant, and texture memory, because after a careful analysis performed on the CUDA code, it was deemed unnecessary to use any of the three for the following reasons:

- **Shared memory**: Shared memory is useful for accessing the same elements multiple times, even by different threads. In the examined case, after partitioning the array, the elements are divided among various threads, and each thread sees only its elements and only once.
- **Constant memory**: Constant memory is global, small but as fast as a cache. The size of constant memory is only 64 KB, and in my case, I would have been able to use it only in the first test, where the number of nodes is 10,000, totaling 80,000 bytes but 40,000 towards the GPU. For the other two tests, it would have been impossible to fit the elements addressed to the GPU.
- **Texture Memory**: Texture memory is optimized for executing graphic rendering operations where access occurs sequentially. It is also optimized for two-dimensional spatial locality. However, the case at hand uses a one-dimensional array, and therefore, the optimization would not have been clearly visible Finally the access to the elements doesn't occurr in a sequential way.

Chapter 5

OMP+CUDA Analysis of the plots

For CUDA plots, the scales for different problem sizes are varied because the speedup turns out to be significantly different.

5.1 Optimization 0

5.1.1 10000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0029057	0.0000000	0.0066556	1.0000000	100.0000
OMP+CUDA	1	1024	0.0026630	0.0003443	0.1853351	0.0359112	3.5911
OMP+CUDA	2	1024	0.0029210	0.0003545	0.1887230	0.0352665	1.7633
OMP+CUDA	4	1024	0.0029470	0.0003472	0.1838457	0.0362021	0.9051
OMP+CUDA_L1	1	1024	0.0027668	0.0000246	0.1825267	0.0364637	3.6464
OMP+CUDA_L1	2	1024	0.0026558	0.0000248	0.1870409	0.0355837	1.7792
OMP+CUDA_L1	4	1024	0.0025934	0.0000254	0.1810915	0.0367527	0.9188

Figure 5.1: Table 10000 nodes

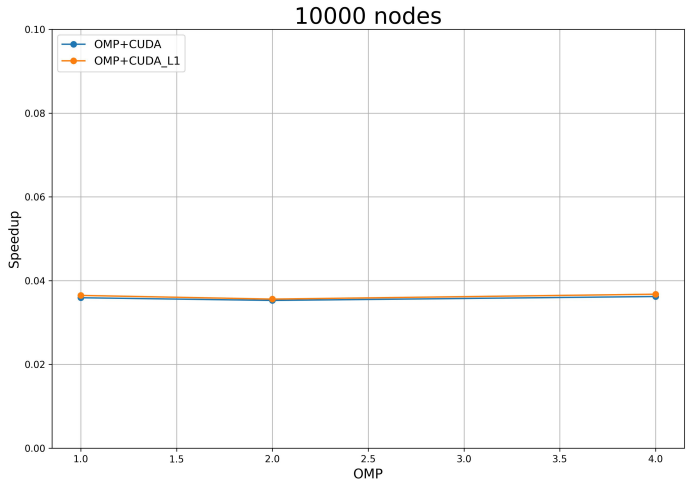

Figure 5.2: Plot 10000 nodes

With this initial test on zero-type optimization, we can see that due to a low number of elements, there is not a significant difference between the two curves; in fact, they remain practically identical. However, upon observing the table, kernel execution times are lower when L1 prioritization is enabled compared to when it is not. Both cases still result in slower performance than the sequential version, which performs significantly better.

5.1.2 250000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.1172753	0.0000004	0.1199208	1.0000000	100.0000
OMP+CUDA	1	1024	0.1143439	0.0003481	0.3257161	0.3681758	36.8176
OMP+CUDA	2	1024	0.1156090	0.0003371	0.3271363	0.3665775	18.3289
OMP+CUDA	4	1024	0.1103755	0.0003579	0.3169068	0.3784103	9.4603
OMP+CUDA_L1	1	1024	0.1139374	0.0000365	0.3234269	0.3707818	37.0782
OMP+CUDA_L1	2	1024	0.1137344	0.0000297	0.3215725	0.3729200	18.6460
OMP+CUDA_L1	4	1024	0.1142815	0.0000330	0.3218391	0.3726110	9.3153

Figure 5.3: Table 250000 nodes

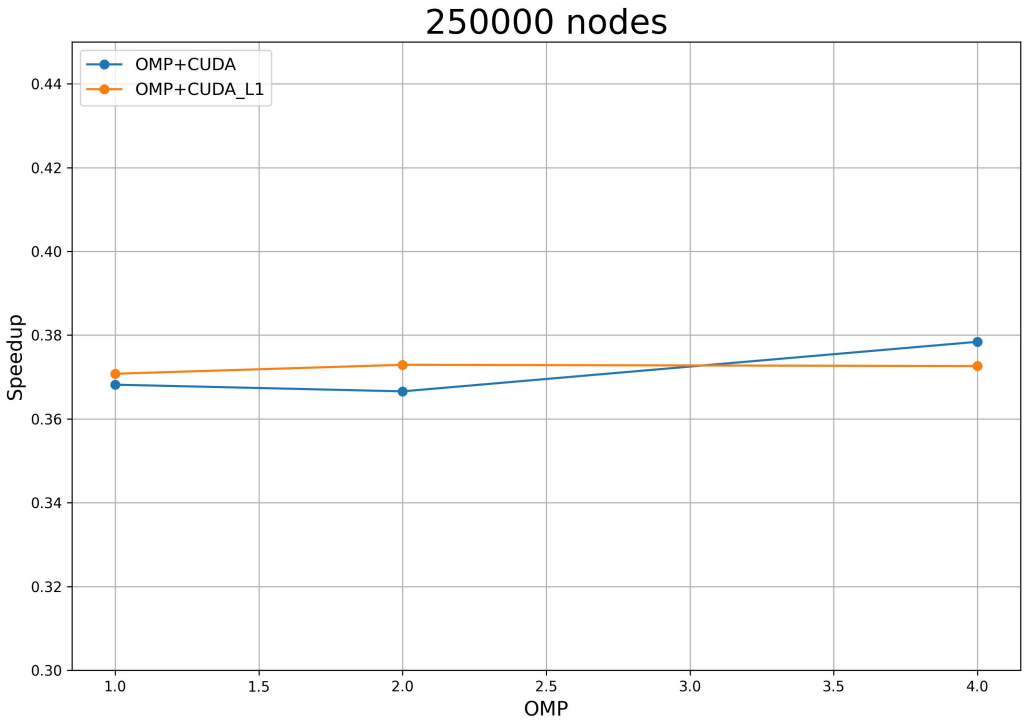

Figure 5.4: Plot 250000 nodes

In this second test, we observe a significant overall increase in speedup for all configuration types, benefiting from the fact that there are now a larger number of elements for the GPU to process. Here, L1 cache prioritization remains stable across all element counts, while the OMP+CUDA version proves to be more effective as the number of OMP threads increases. This is attributed to the fact that the GPU performs better with a larger amount of data and is able to cover parallelization times solely through OMP.

5.1.3 3500000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.774494	0.0000012	3.779051	1.0000000	100.0000
OMP+CUDA	1	1024	3.992026	0.0003700	4.989813	0.7573532	75.7353
OMP+CUDA	2	1024	3.767666	0.0004261	4.973629	0.7598177	37.9909
OMP+CUDA	4	1024	3.970087	0.0004186	4.976006	0.7594547	18.9864
OMP+CUDA_L1	1	1024	3.917999	0.0000928	5.055746	0.7474765	74.7476
OMP+CUDA_L1	2	1024	3.930390	0.0000916	4.961588	0.7616617	38.0831
OMP+CUDA_L1	4	1024	3.810758	0.0000936	4.993371	0.7568136	18.9203

Figure 5.5: Table 3500000 nodes

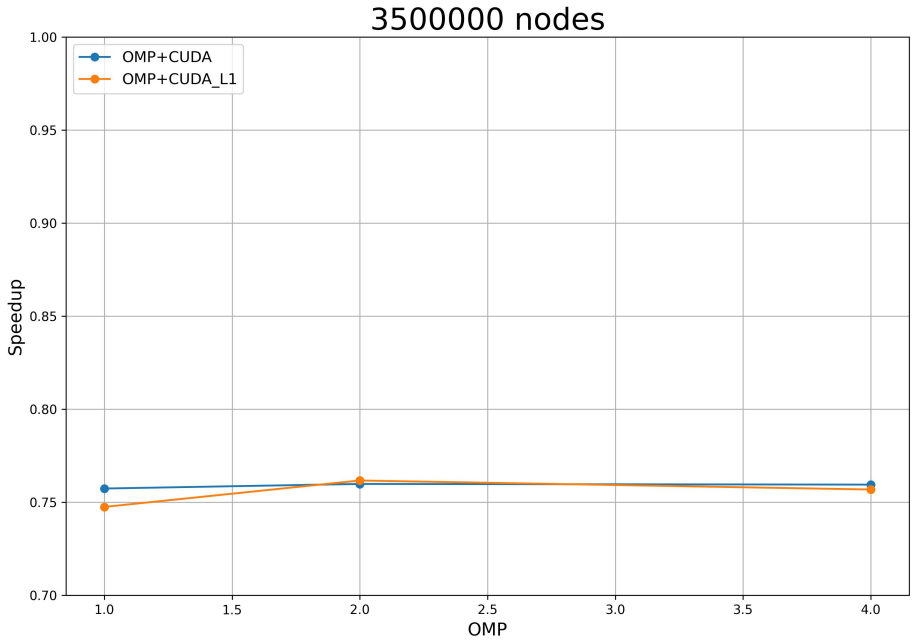

Figure 5.6: Plot 3500000 nodes

In this third case, we observe a significant increase in speedup, but the curves exhibit the same trend as in the previous scenario. The execution difference compared to the sequential version becomes progressively smaller relative to the quantity of nodes. Ultimately, we see that this type of optimization does not benefit parallelization, and the sequential version remains the best performing.

5.2 Optimization 1

5.2.1 10000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0022186	0.0000002	0.0059174	1.0000000	100.0000
OMP+CUDA	1	1024	0.0020613	0.0002039	0.1824153	0.0324392	3.2439
OMP+CUDA	2	1024	0.0021486	0.0001946	0.1834429	0.0322574	1.6129
OMP+CUDA	4	1024	0.0022076	0.0001917	0.1823648	0.0324482	0.8112
OMP+CUDA_L1	1	1024	0.0022500	0.0000268	0.1821300	0.0324900	3.2490
OMP+CUDA_L1	2	1024	0.0022739	0.0000252	0.1848705	0.0320083	1.6004
OMP+CUDA_L1	4	1024	0.0021057	0.0000246	0.1821301	0.0324900	0.8122

Figure 5.7: Table 10000 nodes

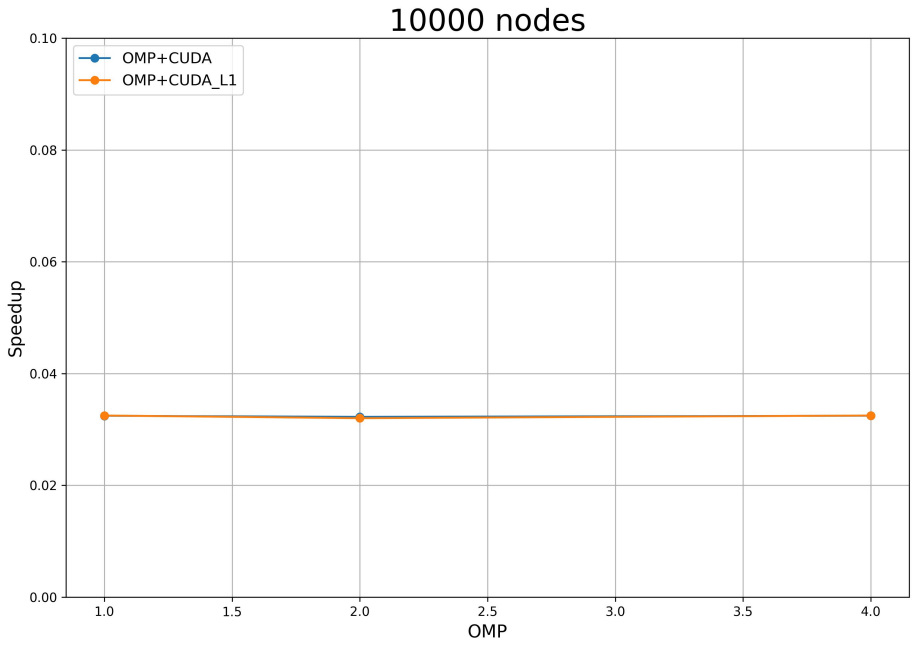

Figure 5.8: Plot 10000 nodes

In this first case of the initial level of optimization, we notice the first benefits of this type of optimization. However, these benefits are more pronounced for the sequential version, which consistently proves to be faster than the OMP+CUDA versions. Overall, especially with this first type of optimization, L1 prioritization shows a more significant impact compared to the version with reduced L1 in comparison to level 0 optimization. Again, the two curves exhibit the same trend as the number of nodes increases.

5.2.2 250000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0998173	0.0000001	0.1023183	1.0000000	100.0000
OMP+CUDA	1	1024	0.0944361	0.0002795	0.2974878	0.3439412	34.3941
OMP+CUDA	2	1024	0.0958215	0.0001602	0.3025708	0.3381632	16.9082
OMP+CUDA	4	1024	0.0940981	0.0001512	0.2917961	0.3506500	8.7662
OMP+CUDA_L1	1	1024	0.0994482	0.0000304	0.3002694	0.3407550	34.0755
OMP+CUDA_L1	2	1024	0.0968794	0.0000308	0.3009659	0.3399664	16.9983
OMP+CUDA_L1	4	1024	0.0965924	0.0000287	0.2970339	0.3444667	8.6117

Figure 5.9: Table 250000 nodes

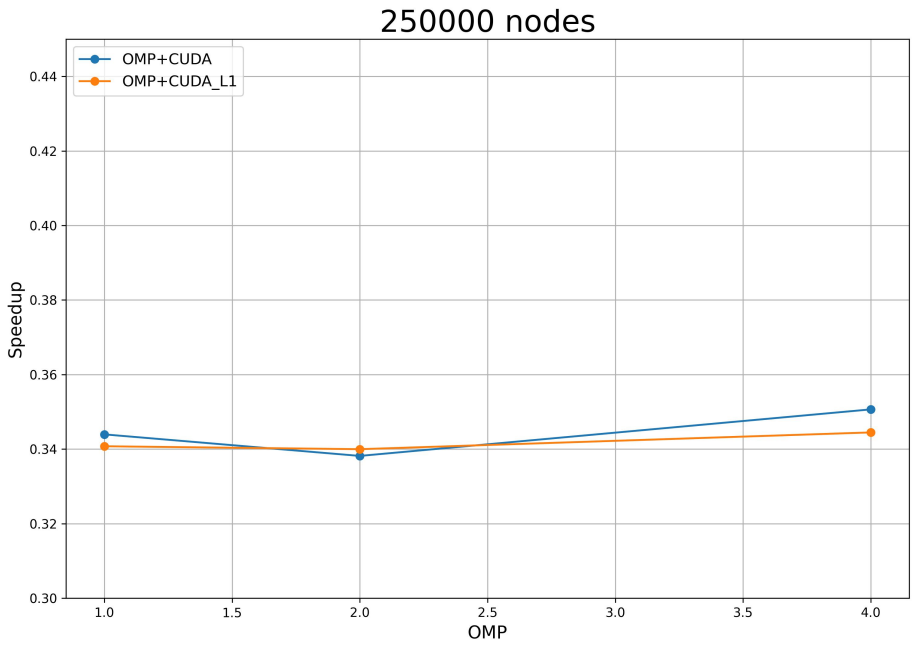

Figure 5.10: Plot 250000 nodes

In this second case, we observe a substantial improvement compared to level zero optimization, especially concerning the execution of the L1 version of the kernel, unlike the OMP+CUDA version alone. They follow the same trend as level 0 optimization but exhibit a significantly improved speedup.

5.2.3 3500000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.737946	0.0000014	3.741499	1.0000000	100.0000
OMP+CUDA	1	1024	3.500207	0.0002704	4.371942	0.8557979	85.5798
OMP+CUDA	2	1024	3.598778	0.0002719	4.509685	0.8296585	41.4829
OMP+CUDA	4	1024	3.473992	0.0002669	4.379152	0.8543889	21.3597
OMP+CUDA_L1	1	1024	3.532805	0.0000924	4.434851	0.8436582	84.3658
OMP+CUDA_L1	2	1024	3.512450	0.0000927	4.393094	0.8516773	42.5839
OMP+CUDA_L1	4	1024	3.652275	0.0000968	4.561907	0.8201611	20.5040

Figure 5.11: Table 3500000 nodes

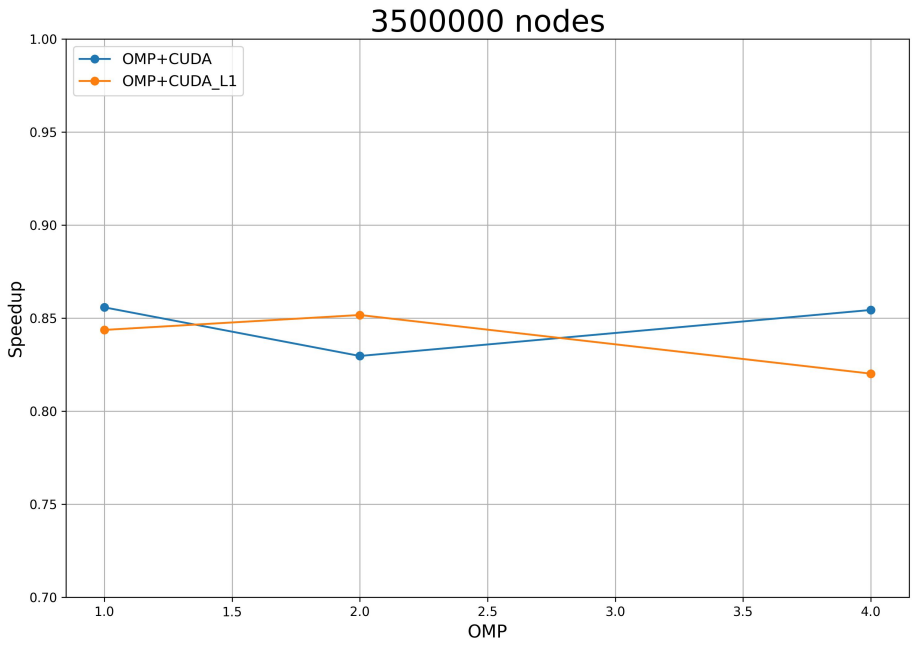

Figure 5.12: Plot 3500000 nodes

In this third case, the impact of the first level of optimization is evident, significantly improving the speedup for both OMP+CUDA versions but notably enhancing the sequential version as well. Essentially, even here, working with a large amount of data, the GPU performs significantly better, substantially reducing latency and execution time relative to the quantity of data it has to handle (which is considerable). However, the sequential version still outperforms the others.

5.3 Optimization 2

5.3.1 10000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0018673	0.0000000	0.0040724	1.0000000	100.0000
OMP+CUDA	1	1024	0.0016999	0.0003540	0.1819941	0.0223765	2.2377
OMP+CUDA	2	1024	0.0018354	0.0003538	0.1890608	0.0215402	1.0770
OMP+CUDA	4	1024	0.0017189	0.0003574	0.1831890	0.0222306	0.5558
OMP+CUDA_L1	1	1024	0.0017815	0.0000251	0.1851614	0.0219938	2.1994
OMP+CUDA_L1	2	1024	0.0017552	0.0000260	0.1867987	0.0218010	1.0901
OMP+CUDA_L1	4	1024	0.0018586	0.0000249	0.1843086	0.0220955	0.5524

Figure 5.13: Table 10000 nodes

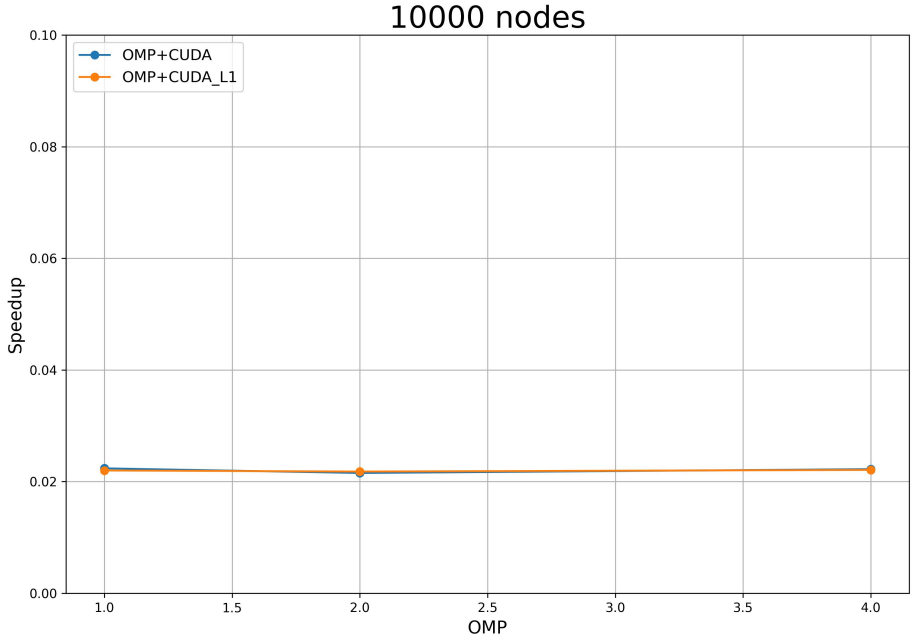

Figure 5.14: Plot 10000 nodes

In this first case of optimization level 2, we observe that the trend of the two curves turns out to be identical to the previous optimizations, with the difference that the overall times have decreased thanks to this type of optimization. The trend is practically similar due to the simple fact that the one benefiting the most is the sequential version, leading to a further decrease in speedup and efficiency. Overall, the time that has benefited the most is precisely the tree creation time, while the version with L1 turns out to be significantly better than the version without L1.

5.3.2 250000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0817646	0.0000002	0.0844295	1.0000000	100.0000
OMP+CUDA	1	1024	0.0797663	0.0003480	0.2709311	0.3116272	31.1627
OMP+CUDA	2	1024	0.0839543	0.0003758	0.2845281	0.2967352	14.8368
OMP+CUDA	4	1024	0.0839340	0.0003628	0.2815805	0.2998414	7.4960
OMP+CUDA_L1	1	1024	0.0837130	0.0000294	0.2783148	0.3033597	30.3360
OMP+CUDA_L1	2	1024	0.0835288	0.0000325	0.2844111	0.2968573	14.8429
OMP+CUDA_L1	4	1024	0.0815656	0.0000339	0.2762143	0.3056666	7.6417

Figure 5.15: Table 250000 nodes

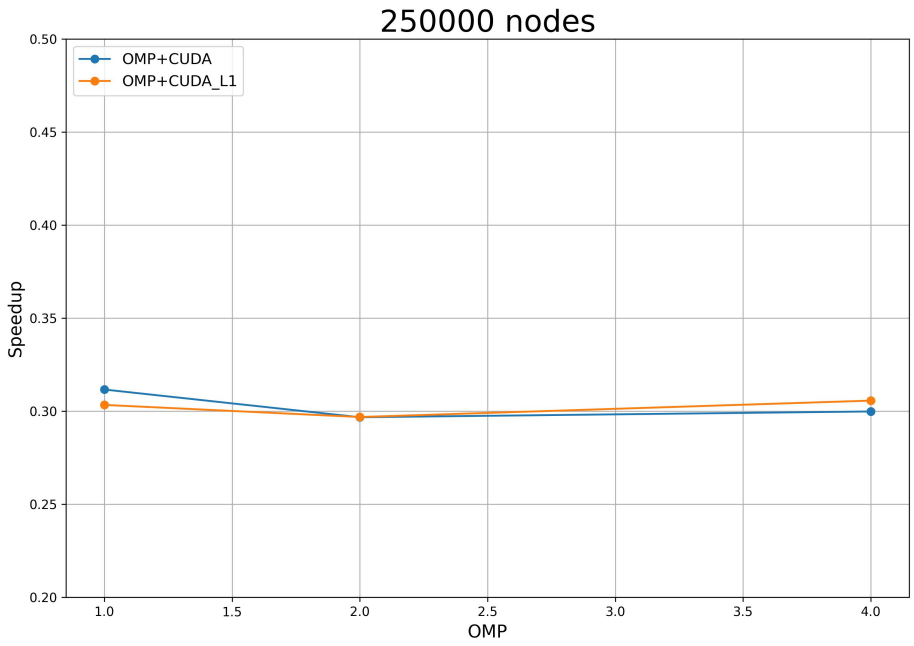

Figure 5.16: Plot 250000 nodes

In this second scenario, we notice a clear improvement in the OMP+CUDA version with L1 compared to the OMP+CUDA version when the number of threads is set to 4. Despite the maximum number of cores and logical threads being 2, both versions improve due to the optimization (with the L1 version improving more than the version without), unlike level 0, which caused the speedup to decrease and degrade performance.

5.3.3 3500000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.488492	0.0000010	3.493205	1.0000000	100.0000
OMP+CUDA	1	1024	3.265996	0.0004836	3.952399	0.8838189	88.3819
OMP+CUDA	2	1024	3.448885	0.0004163	4.202707	0.8311798	41.5590
OMP+CUDA	4	1024	3.343832	0.0003917	4.084781	0.8551757	21.3794
OMP+CUDA_L1	1	1024	3.378444	0.0000928	4.125821	0.8466691	84.6669
OMP+CUDA_L1	2	1024	3.343459	0.0000928	4.030787	0.8666310	43.3315
OMP+CUDA_L1	4	1024	3.466009	0.0000944	4.228142	0.8261798	20.6545

Figure 5.17: Table 3500000 nodes

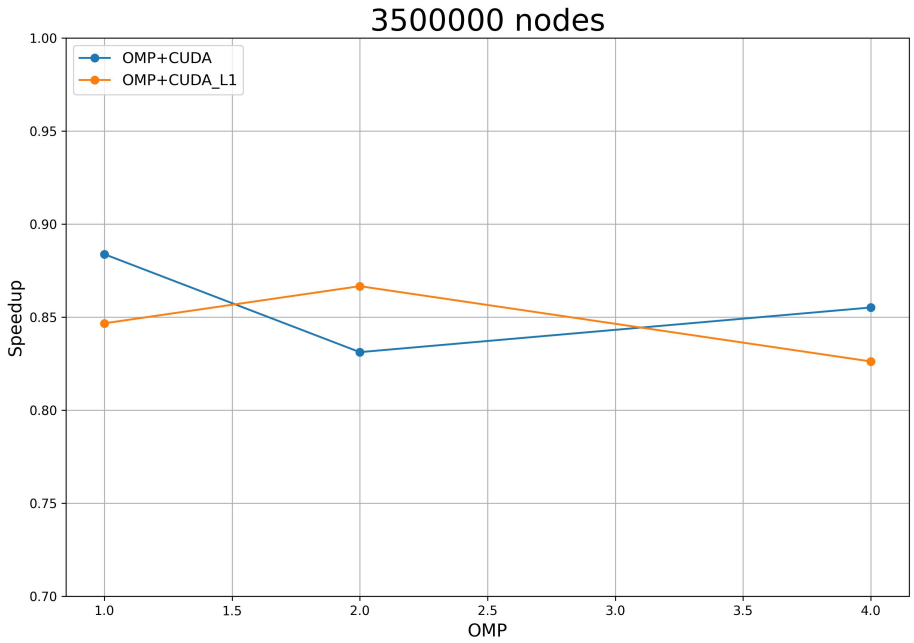

Figure 5.18: Plot 3500000 nodes

For this test, when OMP=1, both versions exhibit a high speedup compared to the norm, much larger than the version with level 1 optimization and especially the one with level 0 optimization. Overall, the times improve, but the improvements are particularly noticeable in the times calculated for operations performed on the host, such as the creation of the structure. The peculiar situation arises when the number of OMP threads is set to 4: the speedup of the OMP+CUDA version improves, while the speedup of the OMP+CUDA+L1 version worsens. This is likely due to the excessive number of threads used on the GPU, leading to strange and unexpected behaviors.

5.4 Optimization 3

5.4.1 10000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0021595	0.0000003	0.0047729	1.0000000	100.0000
OMP+CUDA	1	1024	0.0022141	0.0003541	0.1807691	0.0264033	2.6403
OMP+CUDA	2	1024	0.0021059	0.0003400	0.1880008	0.0253877	1.2694
OMP+CUDA	4	1024	0.0019782	0.0003611	0.1813726	0.0263154	0.6579
OMP+CUDA_L1	1	1024	0.0020260	0.0000247	0.1823865	0.0261691	2.6169
OMP+CUDA_L1	2	1024	0.0021008	0.0000264	0.1877780	0.0254178	1.2709
OMP+CUDA_L1	4	1024	0.0022193	0.0000247	0.1823390	0.0261760	0.6544

Figure 5.19: Table 10000 nodes

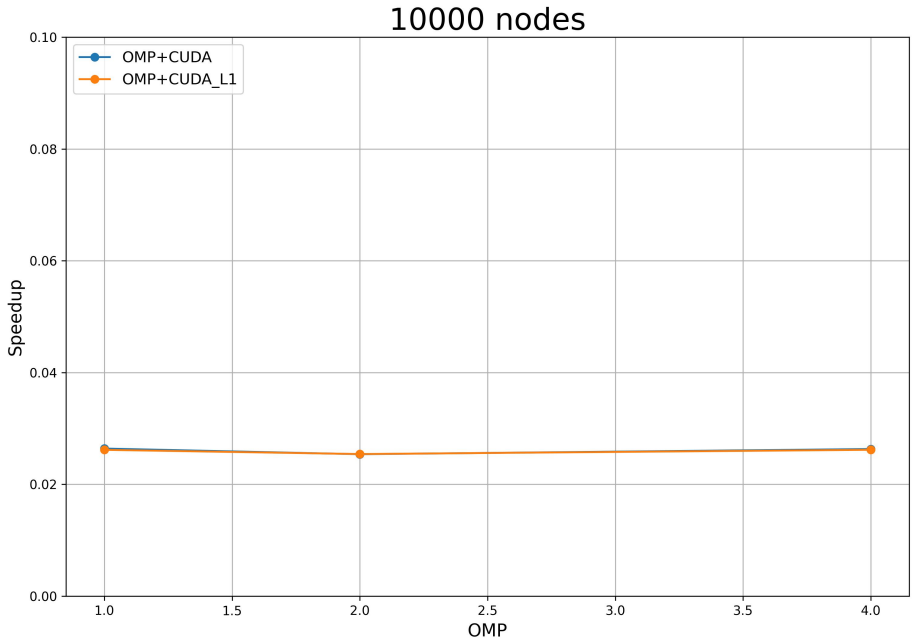

Figure 5.20: Plot 10000 nodes

The first test conducted at optimization level 3 exhibits a curve pattern similar to the previous ones, but with the notable difference that the speedup has increased compared to the version with level 2 optimization. This is because optimization level 3 typically involves a higher degree of parallelism and a more efficient utilization of GPU resources compared to level 2 optimization. Therefore, this encompasses better memory management, increased utilization of registers and caches, and a more efficient organization of threads. Indeed, this has led to a reduction in data transfer times to the GPU, improving execution times for a reduced number of threads.

5.4.2 250000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	0.0934905	0.0000002	0.0961858	1.0000000	100.0000
OMP+CUDA	1	1024	0.0914490	0.0003547	0.2857536	0.3366040	33.6604
OMP+CUDA	2	1024	0.0946213	0.0003463	0.2911814	0.3303295	16.5165
OMP+CUDA	4	1024	0.0933072	0.0007680	0.2875071	0.3345510	8.3638
OMP+CUDA_L1	1	1024	0.0927922	0.0000292	0.2831308	0.3397221	33.9722
OMP+CUDA_L1	2	1024	0.0914723	0.0000287	0.2892520	0.3325329	16.6266
OMP+CUDA_L1	4	1024	0.0915874	0.0000304	0.2842655	0.3383661	8.4592

Figure 5.21: Table 250000 nodes

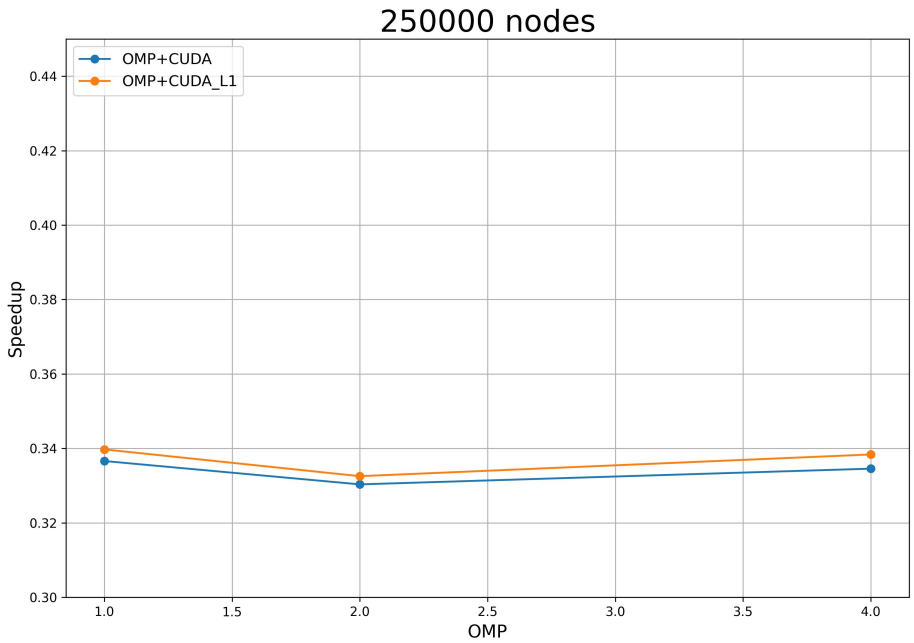

Figure 5.22: Plot 250000 nodes

In this second test, the two curves exhibit the same trend but are simply translated in terms of speedup for each number of threads. In fact, the speedup difference between the two curves remains almost constant, with the L1 version being consistently more optimized for this level of optimization.

5.4.3 3500000 Nodes

Modality	OMP	BlockSize	tree_creation_time	kernel_search_time	program_execution_time	speedup	efficiency
Sequential	0	0	3.679353	0.0000009	3.682017	1.0000000	100.0000
OMP+CUDA	1	1024	3.498973	0.0004626	4.234750	0.8694767	86.9477
OMP+CUDA	2	1024	3.609901	0.0004396	4.354121	0.8456394	42.2820
OMP+CUDA	4	1024	3.513209	0.0003874	4.247976	0.8667696	21.6692
OMP+CUDA_L1	1	1024	3.545318	0.0000955	4.278632	0.8605593	86.0559
OMP+CUDA_L1	2	1024	3.578910	0.0000933	4.309798	0.8543363	42.7168
OMP+CUDA_L1	4	1024	3.604774	0.0000934	4.351510	0.8461469	21.1537

Figure 5.23: Table 3500000 nodes

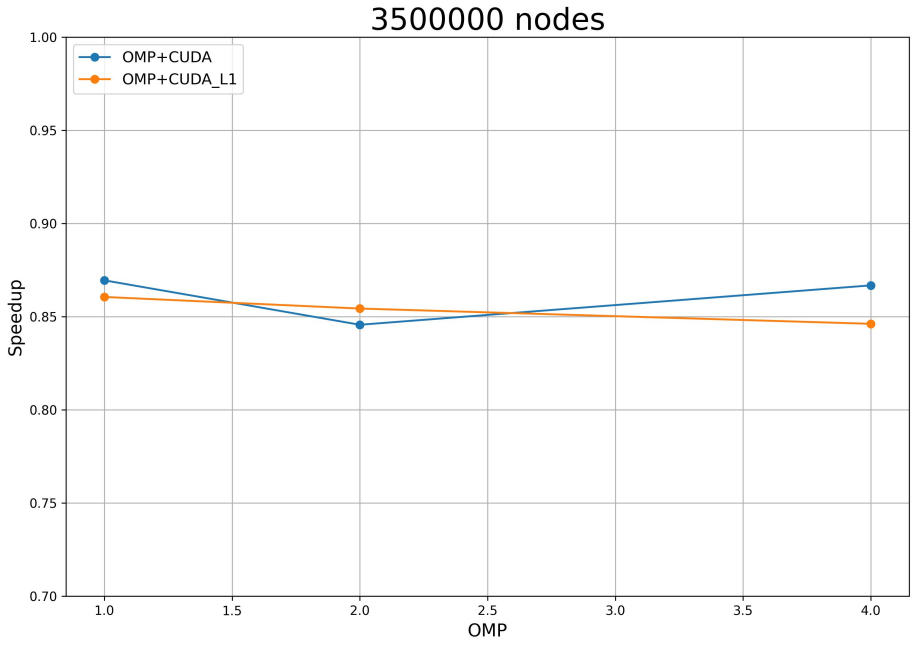

Figure 5.24: Plot 3500000 nodes

For this third and final test as well, we observe that compared to the previous optimizations, the speedup of both versions increases significantly. This leads them to exhibit the same trend as the previous versions but with an increased speedup, thanks to the benefits provided by the third-level optimization.

5.5 Final considerations

In light of what we have observed for all four types of optimization, we have noticed a practically similar trend for each problem size but with appropriate improvements—specifically, an increase in speedup as the number of elements grows. This is because the GPU performs better as the number of elements increases, striking a suitable trade-off between data transfer time to the GPU and processing results. It exploits the kernel, which works on half of the elements, while the other half is handled by the host, which often slows down the overall execution. We can see that in every situation, the sequential version turns out to be decidedly the best. However, this result stems from the fact that the comparison between various versions of OMP+CUDA and OMP+CUDA+L1 was made with a sequential version using a completely different approach, starting with how binary search is implemented. In the tree and, consequently, in the pure sequential version, we use the classic recursive binary search. Meanwhile, for CUDA, we perform binary search on an array containing half of the elements. Additionally, the discrepancy in execution times is caused by different checks, calls to other functions within the kernel itself, and the execution of multiple instructions that reduce the search size until the eventual result is returned. This result underscores that parallelizing the algorithm in this case is not advantageous, as performance significantly declines compared to the sequential version.

Chapter 6

Documentation

Online documentation of ParallelizedRBTreeSearch here.

6.1 RB_Tree_Generator.c

```
/**
 * @brief Create a new node with the given key and color.
 *
 * @param key The key value of the new node.
 * @param color The color of the new node (RED or BLACK).
 * @return A pointer to the newly created node.
 */
struct Node* createNode(int key, int color)

/**
* @brief Initialize a new Red-Black Tree.
*
* @return A pointer to the newly initialized Red-Black Tree.
*/
struct RBTree* initializeRBTree()

/**
* @brief Perform a left rotation on the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param x The node to be rotated.
*/
void leftRotate(struct RBTree* tree, struct Node* x)

/**
* @brief Perform a right rotation on the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param y The node to be rotated.
*/
void rightRotate(struct RBTree* tree, struct Node* y)

/**
* @brief Fix the Red-Black Tree properties after node insertion.
*
* @param tree The Red-Black Tree.
* @param z The newly inserted node.
*/
void RBInsertFixup(struct RBTree* tree, struct Node* z)
```

```
/**
* @brief Insert a new key into the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param key The key value to be inserted.
*/
void RBInsert(struct RBTree* tree, int key)

/**
* @brief Perform an in-order traversal of the Red-Black Tree and print the nodes.
*
* @param node The starting node for the traversal.
*/
void inOrderTraversal(struct Node* node)

/**
* @brief Generate a random number within the specified range [min, max].
*
* @param min The minimum value of the range.
* @param max The maximum value of the range.
* @return The generated random number.
*/
int getRandomNumber(int min, int max)

/**
* @brief Generate and insert random elements into the Red-Black Tree.
* The numbers generated are extracted randomly from the range (1,numElements)
*
* @param numElements The number of random elements to generate.
* @param tree The Red-Black Tree to insert elements into.
*/
void randomElementGenerator(int numElements, struct RBTree* tree)

/**
* @brief Generate a Red-Black Tree with random elements taken from the
* srand() function.
*
* @param numElements The number of random elements to generate.
* @param seed The seed for the random number generator.
* @param opt Unused parameter.
* @return A pointer to the generated Red-Black Tree.
*/
struct RBTree* generateTree(int numElements, int seed, int opt)
```

6.2 RB_Tree_Sequential.c

```
/**
* @brief Search for a key in a Red-Black Tree recursively.
*
* @param root The root of the Red-Black Tree.
* @param key The key to be searched.
* @return A pointer to the node containing the key if found, otherwise NULL.
*/
struct Node* RBSearch(struct Node* root, int key)

/**
* @brief Print performance information to a CSV file.
*
* @param n_nodes The number of nodes in the Red-Black Tree.
* @param opt An optimization parameter.
* @param creation_time Time taken for tree creation.
* @param communicationtime Time taken for communication (not used in this context).
* @param rbsearch_time Time taken for Red-Black Tree search.
* @param execution_time Total execution time.
*/
void printSeqToCSV(int n_nodes, int opt, double creation_time, double communicationtime,
    double rbsearch_time, double execution_time)

/**
* @brief Function useful to print the result of the execution to a TXT file.
* @param n_nodes The number of nodes in the Red-Black Tree.
* @param result The result of the RB Search
* @param valueToSearch key to search in the tree
*/
void printFoundToTXT(int n_nodes, struct Node* result, int valueToSearch)
```

6.3 RB_Tree_OMP_MPI.c

```
/**
* @brief Count the number of nodes in the Red-Black Tree.
*
* @param node The current node being checked.
* @param sentinel The sentinel node representing NULL.
* @return The number of nodes in the subtree rooted at the given node.
*/
int countNodes(struct Node* node, struct Node*sentinel)

/**
* @brief Fill an array with the information of the Red-Black Tree in in-order traversal.
*
* @param node The current node being processed.
* @param array The array to be filled.
* @param index The current index in the array.
* @param sentinel The sentinel node representing NULL.
*/
void fillArrayInOrder(struct Node* node, struct Node** array, int* index, struct Node*
    sentinel)

/**
* @brief Create and fill an array with the information of the Red-Black Tree in sorted
    order.
*
* @param tree The Red-Black Tree.
* @param arraySize Pointer to store the size of the created array.
* @return The sorted array of nodes.
*/
struct Node** createSortedArray(struct RBTree* tree, int* arraySize)

/**
* @brief Perform a binary search on a sorted array in parallel using OpenMP.
*
* @param localArray The local portion of the sorted array.
* @param extralength The size of the local portion.
* @param key The key to search for.
* @return The result of the binary search.
*/
struct SimpleNode binarySearch(struct SimpleNode* localArray, int extralength, int key)

/**
* @brief MPI reduction operation to find the maximum key among SimpleNode elements.
*
* @param in Input data for a process.
* @param inout Combined data for a process.
* @param len Length of the data.
* @param datatype MPI datatype.
*/
void maxKeyReducer(void *in, void *inout, int *len, MPI_Datatype *datatype)
```

```
/**
* @brief Print performance information to a CSV file for parallel execution.
*
* @param n_nodes The number of nodes in the Red-Black Tree.
* @param opt An optimization parameter.
* @param numThreads The number of OpenMP threads used.
* @param size The number of MPI processes.
* @param creation_time Time taken for tree creation.
* @param communicationtime Time taken for communication (MPI).
* @param rbsearch_time Time taken for parallel binary search.
* @param execution_time Total execution time.
*/
void printParToCSV(int n_nodes, int opt, int numThreads, int size, double creation_time,
    double communicationtime, double rbsearch_time, double execution_time)

/**
* @brief Function useful to print the result of the execution to a TXT file.
* @param n_nodes The number of nodes in the Red-Black Tree.
* @param result The result of the RB Search
* @param valueToSearch key to search in the tree
*/
void printFoundToTXT(int n_nodes, struct SimpleNode globalResult, int valueToSearch)

/**
* @brief Generate a random number within a specified range.
*
* @param min The minimum value of the range.
* @param max The maximum value of the range.
* @return The randomly generated number.
*/
int getRandomNum(int min, int max)
```

6.4 RB_Tree_OMP_CUDA.c

```
/**
 * @brief Create a new node with the given key and color.
 *
 * @param key The key value of the new node.
 * @param color The color of the new node (RED or BLACK).
 * @return A pointer to the newly created node.
 */
struct Node* createNode(int key, int color)

/**
* @brief Initialize a new Red-Black Tree.
*
* @return A pointer to the newly initialized Red-Black Tree.
*/
struct RBTree* initializeRBTree()

/**
* @brief Perform a left rotation on the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param x The node to be rotated.
*/
void leftRotate(struct RBTree* tree, struct Node* x)

/**
* @brief Perform a right rotation on the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param y The node to be rotated.
*/
void rightRotate(struct RBTree* tree, struct Node* y)

/**
* @brief Fix the Red-Black Tree properties after node insertion.
*
* @param tree The Red-Black Tree.
* @param z The newly inserted node.
*/
void RBInsertFixup(struct RBTree* tree, struct Node* z)


/**
* @brief Insert a new key into the Red-Black Tree.
*
* @param tree The Red-Black Tree.
* @param key The key value to be inserted.
*/
void RBInsert(struct RBTree* tree, int key)
```

```
/**
* @brief Perform an in-order traversal of the Red-Black Tree and print the nodes.
*
* @param node The starting node for the traversal.
*/
void inOrderTraversal(struct Node* node)

/**
* @brief Generate a random number within the specified range [min, max].
*
* @param min The minimum value of the range.
* @param max The maximum value of the range.
* @return The generated random number.
*/
int getRandomNumber(int min, int max)

/**
* @brief Generate and insert random elements into the Red-Black Tree.
* The numbers generated are extracted randomly from the range (1,numElements)
*
* @param numElements The number of random elements to generate.
* @param tree The Red-Black Tree to insert elements into.
*/
void randomElementGenerator(int numElements, struct RBTree* tree)

/**
* @brief Generate a Red-Black Tree with random elements taken from the
* srand() function.
*
* @param numElements The number of random elements to generate.
* @param seed The seed for the random number generator.
* @param opt Unused parameter.
* @return A pointer to the generated Red-Black Tree.
*/
struct RBTree* generateTree(int numElements, int seed, int opt)

/**
* @brief Count the number of nodes in the Red-Black Tree.
*
* @param node The current node being checked.
* @param sentinel The sentinel node representing NULL.
* @return The number of nodes in the subtree rooted at the given node.
*/
int countNodes(struct Node* node, struct Node*sentinel)

/**
* @brief Fill an array with the information of the Red-Black Tree in in-order traversal.
*
* @param node The current node being processed.
* @param array The array to be filled.
* @param index The current index in the array.
* @param sentinel The sentinel node representing NULL.
*/
void fillArrayInOrder(struct Node* node, struct Node** array, int* index, struct Node*
    sentinel)
```

```
/**
* @brief Create and fill an array with the information of the Red-Black Tree in sorted
    order.
*
* @param tree The Red-Black Tree.
* @param arraySize Pointer to store the size of the created array.
* @return The sorted array of nodes.
*/
struct Node** createSortedArray(struct RBTree* tree, int* arraySize)

/**
 * @brief Prints information about the execution of the program to a CSV file.
 *
 * @param n_nodes Number of nodes in the Red-Black Tree.
 * @param opt Optimization option.
 * @param numThreads Number of OpenMP threads.
 * @param RB_creation_time Time taken for Red-Black Tree creation.
 * @param kernel_execution_time Time taken for the CUDA kernel execution.
 * @param execution_time Total execution time of the program.
 */
void printCUDAToCSV(int n_nodes, int opt, int numThreads, double RB_creation_time, float
    kernel_execution_time, double execution_time)

/**
 * @brief Function useful to print the result of the execution to a TXT file.
 * @param n_nodes The number of nodes in the Red-Black Tree.
 * @param result The result of the RB Search
 * @param valueToSearch key to search in the tree
*/
void printFoundToTXT(int n_nodes, int finalResult, int valueToSearch)

/**
 * @brief Performs a binary search on a sorted array.
 *
 * @param dsortedArray Pointer to the sorted array in device memory.
 * @param start Starting index for the search.
 * @param end Ending index for the search.
 * @param valueToSearch Value to be searched in the array.
 * @return 1 if the value is found, 0 otherwise.
 */
__host__ __device__ int binarySearch(struct SimpleNode* dsortedArray, int start, int end,
    int valueToSearch)

/**
 * @brief function to find the minimum among two numbers.
 *
 * @param a first number.
 * @param b second number.
 */
__device__ int cuda_fmin(int a,int b)
```

```
/**
 * @brief CUDA kernel function to perform parallel binary search on the GPU.
 *
 * @param dsortedArray Pointer to the sorted array in device memory.
 * @param arraySizeGPU Size of the array in GPU memory.
 * @param valueToSearch Value to be searched in the array.
 * @param dfound Pointer to the variable storing the result.
 */
__global__ void RBSearchKernel(struct SimpleNode* dsortedArray, int arraySizeGPU,int
    valueToSearch, int* dfound)

/**
 * @brief Searches for a value on the GPU using CUDA and on the CPU using OpenMP, and
     measures the time taken.
 *
 * @param sortedArray Array of nodes sorted by the Red-Black Tree.
 * @param arraysize Size of the sortedArray.
 * @param numThreadsOMP Number of OpenMP threads to use for CPU search.
 * @param valueToSearch Value to be searched in the array.
 * @param numThreadCUDA Number of CUDA threads per block for GPU search.
 * @return Time taken for the GPU and CPU combined search.
 */
float searchOnDevice(struct Node** sortedArray, int arraysize, int numThreadsOMP, int
    valueToSearch)
```

Chapter 7

How to run

Dependencies

- OMP, Python3, R

How to run
Premise: The code for generating plots requires the python interpreter and two libraries to be installed, matplotlib, and pandas with the following commands:

- pip3 install matplotlib
- pip install pandas

When generating tables with R a package will be installed, known as **kableExtra** and **webshot**.

7.1 Compilation

1. Navigate to the project folder containing the makefile;
2. To clear previously obtained achievements and previous builds, enter the command **make clean**;

7.1.1 OMP+MPI

1. To generate the necessary directories and compile and linking the various source codes, enter the command: **make all**;
2. To run the algorithm for making omp+mpi tests and producing results enter the command **make mpitest**;
3. To produce plots and tables enter the command: **make mpi_plots_tables**;

7.1.2 OMP+CUDA

1. To linking the various source codes, enter the command: **make cudacompile**;
2. To run the algorithm for making omp+cuda tests and producing results enter the command **make cudatest**;
3. To produce plots and tables enter the command: **make cuda_plots_tables**.

The results of the algorithms can be viewed in the "**RB_Search_Report**" folder, which has three subfolders depending on the size of the problem (the nodes of the tree).
The execution times of the algorithms and their average values can be viewed respectively in the "**Informations**" and "**Results**" folders, divided by optimization.
The results in graphical and tabular format can be viewed respectively in the "**Plots**" and "**Tables**" folders, divided by optimization (each optimization iterations lasts about 20 minutes).

Chapter 8

Cluster parallelization (G.D'Aniello)

In the following section of the report, we will delve into **cluster programming**, encompassing both the utilization of the **Apache Hadoop** framework employing the **Map-Reduce**programming paradigm, and the utilization of the **Apache Spark** framework.

8.1 Dataset analysis

This dataset contains information about the matches played during the FIFA World Cup from 1930 to 1998. In particular, below there is the dataset in which we can observe the presence of **20 columns** indicating:

- **Year:** the year in which a specific edition of the World Cup was contested;
- **Datetime:** the day and time during which a match was played;
- **Stage:** the stage of the World cup in which a match was played;
- **Stadium:** the name of the stadium in which a match was played;
- **City:** the name of the city in which a match was played;
- **Home Team Name:** the team in that match that played at home;
- **Home Team Goals:** the number of goals of the Home Team;
- **Away Team Goals:** the number of goals of the Away Team;
- **Away Team Name:** the team in that match that played away;
- **Win conditions:** special conditions in which the match was not concluded within the regular 90 minutes by either of the two teams;
- **Attendance:** the number of spectators for that match;
- **Half-time Home Goals:** the number of goals of the Home Team before the 45 minutes;
- **Half-time Away Goals:** the number of goals of the Away Team before the 45 minutes;
- **Referee:** the name and the nationality of the referee;
- **Assistant1:** the name and the nationality of the first assistant;
- **Assistant2:** the name and the nationality of the second assistant;
- **RoundID:** the ID of the stage in which the match was played;
- **MatchID:** the ID of the match;
- **Home Team Initials:** the initials of the home team (ITA), (USA)...;
- **Away Team Initials:** the initials of the away team (ITA), (USA).

Year	Datetime	Stage	Stadium	City	Home Team Name	Home Team Goals	Away Team Goals	Away Team Name	Win conditions
1930	13 Jul 1930 - 15:00	Group 1	Pocitos	Montevideo	France	4	1	Mexico	
1930	13 Jul 1930 - 15:00	Group 4	Parque Central	Montevideo	USA	3	0	Belgium	
1930	14 Jul 1930 - 12:45	Group 2	Parque Central	Montevideo	Yugoslavia	2	1	Brazil	
1930	14 Jul 1930 - 14:50	Group 3	Pocitos	Montevideo	Romania	3	1	Peru	
1930	15 Jul 1930 - 16:00	Group 1	Parque Central	Montevideo	Argentina	1	0	France	
1930	16 Jul 1930 - 14:45	Group 1	Parque Central	Montevideo	Chile	3	0	Mexico	
1930	17 Jul 1930 - 12:45	Group 2	Parque Central	Montevideo	Yugoslavia	4	0	Bolivia	
1930	17 Jul 1930 - 14:45	Group 4	Parque Central	Montevideo	USA	3	0	Paraguay	
1930	18 Jul 1930 - 14:30	Group 3	Estadio Centenario	Montevideo	Uruguay	1	0	Peru	
1930	19 Jul 1930 - 12:50	Group 1	Estadio Centenario	Montevideo	Chile	1	0	France	
1930	19 Jul 1930 - 15:00	Group 1	Estadio Centenario	Montevideo	Argentina	6	3	Mexico	
1930	20 Jul 1930 - 13:00	Group 2	Estadio Centenario	Montevideo	Brazil	4	0	Bolivia	
1930	20 Jul 1930 - 15:00	Group 4	Estadio Centenario	Montevideo	Paraguay	1	0	Belgium	
1930	21 Jul 1930 - 14:50	Group 3	Estadio Centenario	Montevideo	Uruguay	4	0	Romania	
1930	22 Jul 1930 - 14:45	Group 1	Estadio Centenario	Montevideo	Argentina	3	1	Chile	
1930	26 Jul 1930 - 14:45	Semi-finals	Estadio Centenario	Montevideo	Argentina	6	1	USA	
1930	27 Jul 1930 - 14:45	Semi-finals	Estadio Centenario	Montevideo	Uruguay	6	1	Yugoslavia	
1930	30 Jul 1930 - 14:15	Final	Estadio Centenario	Montevideo	Uruguay	4	2	Argentina	
1934	27 May 1934 - 16:30	Preliminary round	Stadio Benito Mussolini	Turin	Austria	3	2	France	Austria win after extra time
1934	27 May 1934 - 16:30	Preliminary round	Giorgio Ascarelli	Naples	Hungary	4	2	Egypt	

Figure 8.1: First columns of the dataset.

Attendance	Half-time Home Goals	Half-time Away Goals	Referee	Assistant 1	Assistant 2	RoundID	MatchID	Home Team Initials	Away Team Initials
4444	3	0	LOMBARDI Domingo (URU)	CRISTOPHE Henry (BEL)	REGO Gilberto (BRA)	201	1096	FRA	MEX
18346	2	0	MACIAS Jose (ARG)	MATEUCCI Francisco (URU)	WARNKEN Alberto (CHI)	201	1090	USA	BEL
24059	2	0	TEJADA Anibal (URU)	VALLARINO Ricardo (URU)	BALWAY Thomas (FRA)	201	1093	YUG	BRA
2549	1	0	WARNKEN Alberto (CHI)	LANGENUS Jean (BEL)	MATEUCCI Francisco (URU)	201	1098	ROU	PER
23409	0	0	REGO Gilberto (BRA)	SAUCEDO Ulises (BOL)	RADULESCU Constantin (ROU)	201	1085	ARG	FRA
9249	1	0	CRISTOPHE Henry (BEL)	APHESTEGUY Martin (URU)	LANGENUS Jean (BEL)	201	1095	CHI	MEX
18306	0	0	MATEUCCI Francisco (URU)	LOMBARDI Domingo (URU)	WARNKEN Alberto (CHI)	201	1092	YUG	BOL
18306	2	0	MACIAS Jose (ARG)	APHESTEGUY Martin (URU)	TEJADA Anibal (URU)	201	1097	USA	PAR
57735	0	0	LANGENUS Jean (BEL)	BALWAY Thomas (FRA)	CRISTOPHE Henry (BEL)	201	1099	URU	PER
2000	0	0	TEJADA Anibal (URU)	LOMBARDI Domingo (URU)	REGO Gilberto (BRA)	201	1094	CHI	FRA
42100	3	1	SAUCEDO Ulises (BOL)	ALONSO Gualberto (URU)	RADULESCU Constantin (ROU)	201	1086	ARG	MEX
25466	1	0	BALWAY Thomas (FRA)	MATEUCCI Francisco (URU)	VALLEJO Gaspar (MEX)	201	1091	BRA	BOL
12000	1	0	VALLARINO Ricardo (URU)	MACIAS Jose (ARG)	LOMBARDI Domingo (URU)	201	1089	PAR	BEL
70022	4	0	REGO Gilberto (BRA)	WARNKEN Alberto (CHI)	SAUCEDO Ulises (BOL)	201	1100	URU	ROU
41459	2	1	LANGENUS Jean (BEL)	CRISTOPHE Henry (BEL)	SAUCEDO Ulises (BOL)	201	1084	ARG	CHI
72886	1	0	LANGENUS Jean (BEL)	VALLEJO Gaspar (MEX)	WARNKEN Alberto (CHI)	202	1088	ARG	USA
79867	3	1	REGO Gilberto (BRA)	SAUCEDO Ulises (BOL)	BALWAY Thomas (FRA)	202	1101	URU	YUG
68346	1	2	LANGENUS Jean (BEL)	SAUCEDO Ulises (BOL)	CRISTOPHE Henry (BEL)	405	1087	URU	ARG
16000	0	0	VAN MOORSEL Johannes (NED)	CAIRONI Camillo (ITA)	BAERT Louis (BEL)	204	1104	AUT	FRA
9000	2	2	BARLASSINA Rinaldo (ITA)	DATTILO Generoso (ITA)	SASSI Otello (ITA)	204	1119	HUN	EGY

Figure 8.2: Second columns of the dataset.

In the dataset, there are no specific problems, except for a few stadium and city names with accented letters such as **Maracanã** or **Düsseldorf** of that are being interpreted oddly. This is not a concern for us since we don't manually search for stadium names; the programs will merely read and compare them. Consequently, we won't invest effort in rectifying them. The only action we will take is to disregard the initial row, which serves as the header.

8.2 Hadoop Map-Reduce

Exercise 1 : Create the index of referees of a specific nationality that allows you to know which matches each referee has officiated, considering that a referee can take on the roles of Referee, Assistant 1, and Assistant 2 in different matches. The nationality should be specified by the user (e.g., ITA).

Now we will look at the solution code, then see how to run it, and finally evaluate its output.

8.2.1 Driver

Listing 8.1: RefereeDriver

```
/*
* Driver to manage the Job "RefereeIndex" used to obtain all the referee names of a certain
    nationality.
* The nationality is taken from the command line.
*/

public class RefereeDriver {

    public static void main(String[] args) throws Exception {
        // Check if the correct number of command line arguments is provided
        if (args.length != 3) {
            System.err.println("Usage: RefereeAnalysisHadoop <input path> <output path>
                <nationality>");
            System.exit(-1);
        }

        // Set up the Hadoop configuration
        Configuration conf = new Configuration();
        conf.set("nationality", args[2]); // Set the nationality from user input

        // Create a new MapReduce job named "RefereeIndex"
        Job job = Job.getInstance(conf, "RefereeIndex");

        // Set the classes for the MapReduce job
        job.setJarByClass(RefereeDriver.class);
        job.setMapperClass(RefereeMapper.class);
        job.setReducerClass(RefereeReducer.class);

        // Set the output key and value types
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // Set the input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // Execute the job and wait for its completion
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

8.2.2 Mapper Job: Inverted Index + Filter

Listing 8.2: RefereeMapperFilter

```
/*
*   Mapper class that for each row, if the nationality of the referee/assistant1/assistant2 is
    equal to the target nationality, write in the context the referee/assistant1/assistant2 name
    and his role in all the matches that he ruled,
*/
public class RefereeMapper extends Mapper<LongWritable, Text, Text, Text>{
    //Nationality taken from input
    private String targetNationality;
    //The name of the Referee/Assistant1/Assistant2
    private Text refereeName = new Text();
    //The role of the refereeName as Referee/Assistant1/Assistant2 in a match
    private Text matchRole = new Text();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        // Retrieve the target nationality from the configuration
        Configuration conf = context.getConfiguration();
        targetNationality = conf.get("nationality");
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException {
        // Convert the input line to a string
        String line = value.toString();
        // Ignore the header
        if (key.get() == 0) {
            return;
        }

       // Creation of an array containing the names of Referee, Assistant1
       // and Assistant2
       // Split the line into fields using a comma as the delimiter
        String[] fields = line.split(",");
        // These fields contain info about Referee-Assistant1-Assistant2
        String[] referees = {fields[13],fields[14],fields[15]};
        //String that will contain all the infos about a single referee
        String[] refereeNameFields;
        //Variable used for obtaining the Referee/Assistant1/Assistant2 nationality
        String refereeNationality;

        // Iterate through the referees (Referee, Assistant1, Assistant2)
        for(String referee: referees){

            // In the single cell of the dataset, name, surname and nationality
            // of a Referee/Assistant1/Assistant2 are separate with " ",
            // so we take all the fields one by one
            refereeNameFields = referee.split(" ");

            // From the previous array we take the nationality considering the last element
            // elementof the refereeNameFields array
            refereeNationality = refereeNameFields[refereeNameFields.length - 1];
```

```
            // Check if the referee's nationality matches the target nationality
            if (refereeNationality.contains(targetNationality)) {
                refereeName.set(referee);

                // We take the matchId from the right field to associate it with the
                // referee/assistant1/assistant that managed that match
                String matchId = fields[17];
                // Initialization of a String role variable to contain the role
                // of the Referee/Assistant1/Assistant2
                String role = "";

                //Section to check the role
                //If the referee is equal to `fields[13]`,
                //then their role is that of the referee.
                if (referee.equals(fields[13])) {
                    role = ": Referee";

                //If the referee is equal to `fields[14]`,
                //then their role is that of the Assistant1.
                } else if (referee.equals(fields[14])) {
                    role = ": Assistant1";

                //If the referee is equal to `fields[15]`,
                //then their role is that of the Assistant2.
                } else if (referee.equals(fields[15])) {
                    role = ": Assistant2";
                }

                matchRole.set(matchId + role); //Setting of the role of the match
                context.write(refereeName, matchRole); //Writing the role and the referee name
                    into the context
            }
        }
    }
}
```

8.2.3 Reducer Job: Inverted Index + Filter

Listing 8.3: RefereeReducerFilter

```
/*
*   Reducer class that concat all the referees with the target Nationality taken from input with
    all the matches they arbitrated.
*/

public class RefereeReducer extends Reducer<Text, Text, Text, Text> {
    private Text result = new Text();

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException {
        // Create a StringBuilder to store the matches arbitrated by the referee
        StringBuilder matches = new StringBuilder("\n");

        // Concatenate all the matches arbitrated by the referee
        for (Text val : values) {
            matches.append(val.toString()).append("\n");
        }

        // Set the result text to the concatenated matches
        result.set(matches.toString());

        // Write the referee and their associated matches to the output context
        context.write(key, result);
    }
}
```

8.2.4 How to run

Within the project archive, there are two directories. Currently, our attention will be directed to the contents of the directory named ***Hadoop***. Inside this directory, there are two subdirectories: the ***Source*** directory, which houses the NetBeans project containing the code, and the ***Cluster*** directory, intended to be placed within the volume folder of the Hadoop cluster.

After launching Docker, open a terminal and navigate to the main folder of the Hadoop cluster (where the compose file is located). Here we can execute the command to build the cluster:

```
docker compose up -d
```

Now that the cluster is built, we need to connect to the master by launching a bash shell on it and connecting interactively with the command:

```
docker container exec -ti master bash
```

Firstly, we need to format this node as the namenode of the cluster with the command:

```
hdfs namenode -format
```

After that, now we need to start the distributed file system to work on and the process manager to launch the program with the commands:

```
$HADOOP_HOME/sbin/start-dfs.sh
$HADOOP_HOME/sbin/start-yarn.sh
```

Now we can navigate to the ***Cluster*** folder that we copied into the volume using the command:

```
cd data/Cluster
```

Here we can upload our input file *worldcup1.csv* to HDFS using the dedicated protocol **hdfs://**, calling it */input* with the command:

```
hdfs dfs -put worldcup1.csv hdfs:///input
```

Finally we can launch the program. We have to specify three parameters:

- **inputPath:** the input file Path (*/input* in this case);
- **outputDir:** the output directory path for the first job;
- **nationality:** the nationality of the referees;

So, to execute the jar, we have to use this command:

```
hadoop jar RefereeAnalysisHadoop.jar /input /output ARG
```

After the program execution we can take it from HDFS with the command:

```
hdfs dfs -get /output/part-r-00000 outputProject.txt
```

Now, in the project's ***cluster*** folder, we can find the document *outputProject.txt*, which contains the output of the program.

8.2.5 Output and final considerations

The program output is as follows and it represents all the Argentine referees with their roles in a specific match (for the readability of the result, I arranged the output in two columns, but it appears in a single column). The nationality is chosen as example, but it works with all the inputs:

```
BROZZI Juan (ARG)
1382: Referee
1386: Assistant2
1387: Referee
1437: Assistant1

CASTRILLI Javier (ARG)
8780: Referee
8729: Referee

COEREZZA Norberto Angel (ARG)
1765: Assistant2
2394: Assistant1
2351: Referee
1823: Assistant1
2221: Assistant1
1752: Referee
1811: Referee

COMESANA Miguel (ARG)
2388: Assistant2
2351: Assistant2
2348: Assistant1
2349: Assistant2
2352: Assistant1

ESPOSITO Carlos (ARG)
378: Assistant2
428: Referee
568: Referee
533: Assistant1

GOICOECHEA Roberto (ARG)
1599: Referee
1602: Assistant1

ITHURRALDE Arturo Andres (ARG)
901: Referee
962: Assistant2
903: Assistant2
2246: Assistant2
2220: Assistant1
2351: Assistant1
2337: Assistant1

LAMOLINA Francisco Oscar (ARG)
3078: Referee
3051: Referee

LOUSTAU Juan (ARG)
196: Referee
56: Referee
175: Assistant2
127: Referee

MACIAS Jose (ARG)
1090: Referee
1089: Assistant1
1097: Referee

PESTARINO Luis (ARG)
2352: Assistant2
2167: Assistant2
2062: Assistant2
1986: Assistant2
2391: Assistant1
2182: Referee
2454: Assistant1

ROSSI Claudio (ARG)
8761: Assistant2
8744: Assistant2
8729: Assistant1
8780: Assistant1

TAIBI Ernesto (ARG)
3078: Assistant1
3082: Assistant2
3091: Assistant1
3103: Assistant2
3051: Assistant1
3060: Assistant2

VENTRE Luis Antonio (ARG)
1511: Assistant1
1458: Assistant2
1507: Assistant2
1510: Assistant2
```

Figure 8.3: Hadoop output result.

This is the only output we will see for this program, because the only combination used is without a combiner and with a single reducer. For this jobs in fact, it doesn't make sense to use a combiner or more than one reducer.
The job performs an inverted index+filter, and for both, the combiner is not needed. For the inverted-index pattern, the reducer does nothing, so does the combiner; including it or using more reducers would only weigh down the execution. Similarly, for the filtering pattern, the reducer, like the combiner, should not be used at all.

8.3 Spark

Exercise 2: Find the TOPK stadium with the most goals in a specific edition of the World cup.

Now we will look at the solution code, then see how to run it, and finally evaluate its output.

8.3.1 Driver

Listing 8.4: SparkDriver

```
public class SparkDriver {

    public static void main(String[] args) {
        // Read the input file and output path from command line arguments
        String inputFile = args[0];
        String outputPath = args[1];

        // Create a configuration object and set the name of the application
        SparkConf conf = new SparkConf().setAppName("TopKStadiumGoalperYear");

        // Create a Spark Context object
        JavaSparkContext sc = new JavaSparkContext(conf);

        // Build an RDD of Strings from the input textual file
        // Each element of the RDD is a line of the input file
        JavaRDD<String> lines = sc.textFile(inputFile);

        // Extract header and filter it out from the data
        String header = lines.first();
        JavaRDD<String> newData = lines.filter(row->!row.equals(header));

        // Create a Pair RDD with stadium and total goals per match
        // With flatMapToPair, each "row" in "newData" is splitted in fields
        // using the "," as delimiter putting the results into the "fields" array
        JavaPairRDD<String,Integer> stadiumGoalPerMatch = newData.flatMapToPair(row->{
            String[] fields = row.split(",");

            // Result is a list that contains the Tuple2 of String and Integer
            // that specifies the number of goals associated with that stadium
            List<Tuple2<String, Integer>> result = new ArrayList<>();

            // If the year is the same as the year taken from input,
            // will be added into result the stadium
            // with the sum of home goals and away goals scored into that stadium
            if(fields[0].equals(args[2]))
                result.add(new Tuple2(fields[3],
                    Integer.valueOf(fields[6])+Integer.valueOf(fields[7])));
            return result.iterator();
        });

        // Reduce by key to get the total goals per stadium scored in the specified year
        JavaPairRDD<String,Integer> totalGoalPerStadium =
            stadiumGoalPerMatch.reduceByKey((x,y)->x+y);

        // Get the top K stadiums using the TupleComparator that determine the order of Tuples
        //depending on the number of goals. The top k is taken from input
        List<Tuple2<String, Integer>> topStadiums =
            totalGoalPerStadium.top(Integer.valueOf(args[3]), new TupleComparator());
```

```
        // Convert the top K stadiums to a Pair RDD containing
        //the first top K tuple (stadium) with the total number of goals
        JavaPairRDD<String,Integer> topStadiumRDD = sc.parallelizePairs(topStadiums);

        // Save the top K stadiums and their total goals
        // to the specified output path
        topStadiumRDD.saveAsTextFile(outputPath);

        // Close the Spark Context object
        sc.close();

    }
}
```

8.3.2 Util Class: TupleComparator

Listing 8.5: TupleComparator

```
/*
* TupleComparator is a utility class whose function is to provide custom comparison logic for
    ordering the tuples <Stadium, goalsPerStadium> following two criteria:
*1) With '_2()' Tuples are ordered according the second element i.e., the Integer representing
    the number of goals for a *stage. Thus, the top K is ordered in the descending order of
    goals in each stadium;
*2) With '_1()' Tuples are ordered according the first element i.e., the String representing the
    name of the stadium. Thus the top K is ordered in an alphabetic order. This criterion is
    useful in case the total number of goals is the same for two *stadiums.
*/
public class TupleComparator implements Comparator<Tuple2<String, Integer>>, Serializable {

    @Override
    public int compare(Tuple2<String, Integer> tuple1, Tuple2<String, Integer> tuple2) {
        Integer result = tuple1._2().compareTo(tuple2._2());
        if (result.equals(0))
            result=tuple2._1().compareTo(tuple1._1());
        return result;
    }
}
```

8.3.3 How to run

Now we consider the ***Spark*** folder. Inside it, as the Hadoop one, we find two folders: the ***Source*** folder contains the NetBeans project with the code; the ***Cluster*** folder is the one that should be placed inside the Spark cluster volume folder.

After launching Docker, open a terminal and navigate to the main folder of the Spark cluster (where the compose file is located). Here we can execute the command to build the cluster:

```
docker compose up -d
```

Now that the cluster is built, we need to connect to the master by launching a bash shell on it and connecting interactively with the command:

```
docker container exec -ti sp_master bash
```

The Spark cluster will place us in an internal folder, so the first thing we need to do is move to the project folder within the volume using the command:

```
cd /testfiles/Cluster
```

Before running the program, make sure to have deleted the output folders ***outputsingle*** and ***outputcluster*** already present in the project folder within the volume. Subsequently, we can execute the two versions of the launch script:

```
./launch_single.sh
./launch_cluster.sh
```

In the appropriate folders generated in the project folder within the cluster volume, we will find the output.

8.3.4 Output and final considerations

The program output is as follows and represents the top K stadiums with the most goals in a specific edition of the world cup; in particular the following output is the result of the Top 4 stadium with the most goals in the World cup edition of the 1934.

```
(Nazionale PNF,15)
(Giorgio Ascarelli,11)
(Giovanni Berta,10)
(Stadio Benito Mussolini,10)
```

In particular, the last two stadiums have the same number of goals and for this reason they have been ordered in the alphabetical order thanks to the TupleComparator described above.

```
(Rasunda Stadium,27)
(Nya Ullevi,21)
(Idrottsparken,19)
(Malmo Stadion,12)
(Olympia Stadium,11)
(Arosvallen,7)
(Jarnvallen,6)
(Ryavallen,6)
(Tunavallen,6)
```

This second output is the cluster output that shows the top 9 stadium with the most goals in the World cup edition of the 1958. In particular we can see that the output is split but still maintains the same order both for the number of goals and for the alphabet.