Programming Parallel Algorithms (PPA-S25)
Project Report

Project Title: "Design and Implementation of a Parallel Relational Database from scratch"

Name: Darshan Dinesh Kumar
NetID: dd3888
University ID: N10942768

Project Group Members:

1. **Darshan Dinesh Kumar** (NetID: dd3888, University ID: N10942768)

GitHub Repository Link: https://github.com/darshand15/PPA_Project

Introduction and Motivation

The exponential growth of data, driven by innovations like the internet, smartphones, and personal computers, to name a few, has made databases a cornerstone of modern software applications. As data scales up, storing, managing and querying this data efficiently has become increasingly complex. In this regard, databases serve as the backbone to store, manage, and retrieve vast amounts of structured information in an organized manner. However, designing large-scale, efficient databases presents many challenges in terms of both design and performance.
With the diminishing returns of Moore's Law—which historically predicted continuous improvements in processor speed and computing power through more transistors—there has been a significant shift towards leveraging Parallel Computing. This shift has become increasingly pronounced with the rise of multicore and multiprocessor systems, which have become ubiquitous in modern hardware. Thus, parallelism offers a potential solution for improving database performance by utilizing the power of multiple cores and processors. This project is precisely such an effort to design and implement a Relational Database from scratch and a subset of its many operations while incorporating the various concepts gleaned from our *"Programming Parallel Algorithms"* course and related ideas inherent to the domain of Parallel Computing.

Related Work

The ubiquity and pervasive nature of databases can be contributed to the significant research and development towards various kinds of databases as summarized below:

- **Relational Databases (SQL-based):** Relational databases store data in structured tables with rows and columns, using SQL (Structured Query Language) for querying.

Examples include MySQL, PostgreSQL, SQLite, Microsoft SQL Server, and Oracle Database.

- **NoSQL Databases (non-relational):** NoSQL databases are designed for flexibility, scalability, and handling unstructured or semi-structured data. Examples include MongoDB, Redis, Cassandra, CouchDB, and DynamoDB.

- **Graph Databases:** Graph databases are optimized for managing and querying relationships between data points. Examples include Neo4j, Amazon Neptune, and OrientDB.

- **Time-Series Databases:** Time-series databases are specialized for storing and querying data that is time-stamped, such as sensor logs, financial data, or monitoring metrics. Examples include InfluxDB, TimescaleDB, and Prometheus.

- **Cloud-native / Serverless Databases:** These databases are designed to scale in the cloud and often handle infrastructure and provisioning automatically. Examples include Firebase Realtime Database, Firestore, Amazon Aurora, and PlanetScale.

- **Embedded / Lightweight Databases:** Embedded databases are designed to be bundled inside applications with no need for a separate server process. Examples include SQLite, LevelDB, and RocksDB.

Inspiration

This project is inspired by the well-known SQL-based Relational Databases. It has been attempted to simulate the typical CRUD (Create, Read, Update, Delete) operations of these SQL-based Relational Databases while trying to incorporate various aspects related to Parallelism as part of the design and implementation of the Database. The Textbook titled *"Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe, 7th Edition, Pearson Publications* has been used as the primary reference for various concepts related to Database Systems.

Goals of the Project:

The following are the goals of the project:

- Design and implement a parallel relational database supporting a subset of the typical CRUD (Create, Read, Update, Delete) operations, including queries like different variants of the Select Query, Update Query, Insert Into Query, Delete Query, Order By Query, Group By Query, Aggregate Queries like Count, Min, Max, Sum, Average.

- Incorporate aspects related to Parallelism for various candidate queries as below:

    - **Select query based on the primary key field:** As part of the initial table creation and while performing the operation to set the primary key, an indexing data structure such as a B-Tree can be created. This B-Tree can contain the primary key fields and a pointer to the corresponding row. Thus, the B-Tree would be ordered based on the primary key fields. Therefore, if the select query is based on the primary key field, this indexing data structure would allow for efficient retrieval. It could be explored to see how variants of the Select Query can be optimized using the B-Tree indexing data structure. This could be in relation to searching for a specific row/tuple of the table based on a particular value of the primary key field or searching for rows/tuples based on a range of values of the primary key field.

  - **Select query based on non-primary key fields:** Here, the table could be divided into different chunks that can be parallelly searched through to retrieve the different rows for the Select query. This could use a mechanism similar to the parallel_for primitive. Further, as the retrieved rows must be displayed, explicit care must be taken while printing out the rows parallelly. This could be accounted for by introducing an additional data structure to contain the indices of the rows to be printed. This data structure could be handled through a lock to push the indices and then finally used to sequentially print the rows.

  - **Order By Operation:** The Order By operation that orders the rows of the table based on the mentioned column(s) could be parallelized. As this query would essentially perform the sort operation, it could be explored to implement the sort in parallel by using a method similar to parallel merge sort.

  - **Group By and Aggregate Operations:** The Group By Operation usually involves grouping the tuples based on some condition and then applying some aggregate operation like Count, Min, Max, Sum, Average. Therefore, it could be explored to see how this can be parallelized by using a parallel sort initially to group the tuples and then performing the aggregate operations parallelly using the parallel primitives like filter, tabulate and reduce.

- **Perform Functionality Testing** to verify the behaviour of the different operations that have been implemented. Further, the parallel implementations should be thoroughly tested and compared with their sequential counterparts to verify their correctness.

- **Present a Performance comparison and Evaluation** showing the benefits due to the implemented Parallelism. For example, speedup comparison could be shown to highlight the benefits of the different queries with parallelism implemented as compared to their sequential counterparts.

Summary of the outcomes/results of the Project:

The outcomes/results of the Project can be summarized as follows:

- A parallel relational database supporting a subset of the typical CRUD (Create, Read, Update, Delete) operations has been designed and implemented.

- Various CRUD queries fundamental to the functionality of a Database have been implemented as below:
    - Creating a Table
    - Setting a Primary Key
    - Inserting rows into the table
    - Enforcing the Primary Key Constraint
    - Updating the rows of the table including conditional updation
    - Deleting the rows of the table including conditional deletion
    - Different variants of the Select Query including Selecting all the columns of all the rows, Selecting all the columns of rows meeting a condition, Selecting a subset of the columns of all rows, Selecting a subset of the columns of rows meeting a condition.

- Various aspects related to Parallelism have been identified and implemented as follows:

    - **Select Search Query based on equality of primary key field:** Here, a B-Tree has been implemented for indexing the table based on the primary key field. It was observed that the Select Search Query using the B-Tree implementation performed approximately **20x faster** than the corresponding sequential implementation.

    - **Select Range Query based on primary key field:** Here, a B-Tree has been implemented for indexing the table based on the primary key field. It was observed that the Select Range Query using the B-Tree implementation performed approximately **2x faster** than the corresponding sequential implementation.

    - **Select all Columns for rows meeting a condition:** A parallel for loop was used to implement this query. It was observed that the speedup was suboptimal (below 1) in this case with the speedup further decreasing with the increase in the number of threads. The main reason is that as this query requires displaying of the rows, the implementation uses a lock-based data structure to serialize the prints. As these overheads due to lock-contention are significant, there are no benefits due to parallelism for this query and it in fact worsens the performance as compared to the sequential version.

    - **Select a subset of the Columns for rows meeting a condition:** A parallel for loop was used to implement this query. It was observed that the speedup was suboptimal (below 1) in this case with the speedup further decreasing with the increase in the number of threads. The main reason is that as this query requires displaying of the rows, the implementation uses a lock-based data structure to serialize the prints. As these overheads due to lock-contention are significant, there are no benefits due to parallelism for this query and it in fact worsens the performance as compared to the sequential version.

- **Order by Ascending Query:** A Parallel merge sort was used to implement this query. It was observed that the Parallel version achieved a **maximum speedup of 2.2 for 16 threads**, thereby denoting that there were some benefits due to parallelism.

- **Order by Descending Query:** A Parallel merge sort was used to implement this query. It was observed that the Parallel version achieved a **maximum speedup of 1.36 for 8 threads**, thereby denoting that there were some benefits due to parallelism.

- **Group by Count Query:** Here, two different parallel versions were implemented:

  - Parallel Version 1: This initially performs a parallel merge sort followed by sequential traversal of the sorted rows to aggregate the count of the groups. This version achieves a **maximum speedup of 6 using 32 threads** as compared to the sequential version.

  - Parallel Version 2: This initially performs a parallel merge sort followed by a parallel algorithm to aggregate the count of the groups using parallel primitives like tabulate and filter. This version achieves a **maximum speedup of 5.5 using 32 threads** as compared to the sequential version.

    The Parallel Version 1 performs slightly better than the Parallel Version 2. This could be because the overheads introduced by version 2's algorithm which requires the creation of additional data structures using tabulate and filter is quite considerable that it is outweighing any benefits due to parallelism.

- **Group by Min Query:** Here, two different parallel versions were implemented:

  - Parallel Version 1: This initially performs a parallel merge sort followed by sequential traversal of the sorted rows to aggregate the min of the groups. This version achieves a **maximum speedup of 5.3 using 32 threads** as compared to the sequential version.

  - Parallel Version 2: This initially performs a parallel merge sort followed by a parallel algorithm to aggregate the min of the groups using parallel primitives like tabulate, filter and reduce. This version achieves a **maximum speedup of 6.5 using 32 threads** as compared to the sequential version.

    The Parallel Version 2 performs better than the Parallel Version 1. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **Group by Max Query:** Here, two different parallel versions were implemented:

  - Parallel Version 1: This initially performs a parallel merge sort followed by sequential traversal of the sorted rows to aggregate the max of the groups. This version achieves a **maximum speedup of 5.7 using 32 threads** as compared to the sequential version.

  - Parallel Version 2: This initially performs a parallel merge sort followed by a parallel algorithm to aggregate the max of the groups using parallel primitives like tabulate, filter and reduce. This version achieves a **maximum speedup of 8.1 using 32 threads** as compared to the sequential version.

    The Parallel Version 2 performs better than the Parallel Version 1. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **Group by Sum Query:** Here, two different parallel versions were implemented:

  - Parallel Version 1: This initially performs a parallel merge sort followed by sequential traversal of the sorted rows to aggregate the sum of the groups. This version achieves a **maximum speedup of 6 using 32 threads** as compared to the sequential version.

  - Parallel Version 2: This initially performs a parallel merge sort followed by a parallel algorithm to aggregate the sum of the groups using parallel primitives like tabulate, filter and reduce. This version achieves a **maximum speedup of 8.1 using 32 threads** as compared to the sequential version.

    The Parallel Version 2 performs better than the Parallel Version 1. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **Group by Average Query:** Here, two different parallel versions were implemented:

    - Parallel Version 1: This initially performs a parallel merge sort followed by sequential traversal of the sorted rows to aggregate the average of the groups. This version achieves a **maximum speedup of 5.5 using 32 threads** as compared to the sequential version.

    - Parallel Version 2: This initially performs a parallel merge sort followed by a parallel algorithm to aggregate the average of the groups using parallel primitives like tabulate, filter and reduce. This version achieves a **maximum speedup of 8.3 using 32 threads** as compared to the sequential version.

      The Parallel Version 2 performs better than the Parallel Version 1. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

Detailed Description of the technical details of the Project:

The technical details of the project are elaborated under the following categories:

1. **Design and implementation of a Parallel Relational Database with a subset of the typical CRUD (Create, Read, Update, Delete) operations:**

    - As the project was decided to be developed in C++, there were two different strategies under consideration with regards to the implementation:
        1. A template based programming strategy with the generation of a new C++ program to be compiled and executed
        2. A pre-processor based macro programming strategy with the expansion of the different operations in-place

    - The pre-processor based macro programming strategy was used as this only requires the inclusion of a header file in the client program rather than generating a completely new C++ program that needs to be compiled and executed
    - Consequently, every implemented operation is effectively a macro that gets expanded in-place with the required code for the operation under consideration
    - The C++ Parlay Library is used to avail the parallel fork-join interface in C++
    - The C++ Boost Library is used to aid in Macro Programming
    - All the graphs are generated for a Person relation of 10000 rows containing *id, first name, last name, age, country, salary* as its fields

- A subset of the typical CRUD (Create, Read, Update, Delete) operations inherent to SQL-based Relational Databases were designed and implemented as detailed below:

    - **CREATE_TABLE** Operation: This is used to create a table/relation in the database with the mentioned attributes. For every table/relation in the database, the following code is generated in-place using macro programming:
        - A struct representing a row/tuple of this relation with the appropriate attributes and their types
        - A vector of pointers to the above row struct, essentially representing the table as a collection of rows
        - A vector of strings representing the attribute names to provide support while displaying the table
        - A struct with bools associated with every field to denote if they are part of the Primary Key or not. These booleans are initialized to false.
        - A B-Tree of minimum degree 3

      Macro Signature: CREATE_TABLE(name, fields)
      Example Usage: CREATE_TABLE(Person,
            ((int, id))
            ((std::string, name))
            ((std::string, mob_no))
)

      Code Snippet:

```
#define FIELD(r, data, elem) \
    BOOST_PP_TUPLE_ELEM(2, 0, elem) BOOST_PP_TUPLE_ELEM(2, 1, elem);

#define CREATE_TABLE_STRUCT(name, fields) \
    struct row_##name \
    { \
        BOOST_PP_SEQ_FOR_EACH(FIELD, _, BOOST_PP_VARIADIC_TO_SEQ fields) \
    };\
    typedef struct row_##name row_##name;


#define CREATE_TABLE(name, fields) \
    CREATE_TABLE_STRUCT(name, fields) \
    std::vector<row_##name*> name; \
    std::vector<std::string> attr_##name; \
    BOOST_PP_SEQ_FOR_EACH(PUSH_ATTR_NAME, attr_##name, BOOST_PP_VARIADIC_TO_SEQ fields) \
    CREATE_KEY_STRUCT(name, fields) \
    BTree<row_##name*> pk_btree_##name(3);
```

    - **SET_PRIMARY_KEY** Operation: This is used to set the Primary Key for a particular table/relation in the database. This generates the following code in-place using macro programming:

    - Initializes the bools associated with the mentioned fields to true denoting that they are part of the primary key for this table.
    - Generates a comparator based on the primary key fields. This comparator is fundamental to the functionalities of the B-Tree associated with the primary key fields.

  Macro Signature: SET_PRIMARY_KEY(name, cols)
  Example Usage: SET_PRIMARY_KEY(Person, ((id)))

  Code Snippet:

```
#define PK_CMP_ELSE_IF(r, data, elem) \
    else if(left->BOOST_PP_TUPLE_ELEM(1, 0, elem) != right->BOOST_PP_TUPLE_ELEM(1, 0, elem))
\
    { \
        return left->BOOST_PP_TUPLE_ELEM(1, 0, elem) < right->BOOST_PP_TUPLE_ELEM(1, 0,
elem); \
    }

#define CREATE_PK_CMP(name, cols) \
    auto cmp_##name = [](row_##name* left, row_##name* right) \
    { \
        if(false){}\
        BOOST_PP_SEQ_FOR_EACH(PK_CMP_ELSE_IF, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        return false; \
    };


#define KEY_BOOL_STRUCT_INIT_TRUE(r, data, elem) \
    data.BOOST_PP_TUPLE_ELEM(1, 0, elem) = true;

#define SET_PRIMARY_KEY(name, cols) \
    BOOST_PP_SEQ_FOR_EACH(KEY_BOOL_STRUCT_INIT_TRUE, key_bool_var_##name[0],
BOOST_PP_VARIADIC_TO_SEQ cols) \
    CREATE_PK_CMP(name, cols)
```

- **INSERT_INTO** Operation: This is used to insert rows of values into the table. This generates the following code in-place using macro programming:
    - An instance of the row struct is created and its fields are populated with the mentioned values
    - The primary key constraint is enforced in order to verify that no two rows in the table have the same set of values for the attributes defined as the primary key.
    - The row is inserted into the vector for the table and into the B-Tree only if the primary key constraint is not violated.
    - If the primary key constraint is violated, the insertion of the row is skipped and an appropriate message is output to the screen after which the program flow continues.

Macro Signature: INSERT_INTO(name, values)
Example Usage: INSERT_INTO(Person,
((id, 0))
((name, "John"))
((mob_no, "12345"))
)

Code Snippet:

```
#define INSERT_FIELDS_INTO_ROW(r, data, elem) \
    data->BOOST_PP_TUPLE_ELEM(2, 0, elem) = BOOST_PP_TUPLE_ELEM(2, 1, elem);

#define INSERT_INTO(name, values) \
    { \
        row_##name *tab_row = new row_##name; \
        BOOST_PP_SEQ_FOR_EACH(INSERT_FIELDS_INTO_ROW, tab_row, BOOST_PP_VARIADIC_TO_SEQ
values) \
        bool primary_key_constraint_viol = false; \
        ENFORCE_PRIMARY_KEY_CONSTRAINT(name, values, primary_key_constraint_viol) \
        if(!primary_key_constraint_viol) \
        { \
            name.push_back(tab_row); \
            pk_btree_##name.insert(tab_row, cmp_##name); \
        } \
        else \
        { \
            std::cout << "Insertion of row skipped as Primary Key Constraint would be
violated\n"; \
        } \
    }
```

- **SELECT_ALL** Operation: This is used to retrieve all the rows from the mentioned table. This generates the following code in-place using macro programming:
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the vector of rows of the table to display the field values for each of these rows

Macro Signature: SELECT_ALL(name)
Example Usage: SELECT_ALL(Person)

Code Snippet:

```
#define SELECT_ALL(name) \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    }
```

- **SELECT_ALL_COND** Operation: This is used to retrieve specific rows from the mentioned table based on the mentioned conditions. This generates the following code in-place using macro programming:
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the vector of rows of the table to display the field values for each of these rows satisfying the mentioned conditions.

  Macro Signature: SELECT_ALL_COND(name, cond_lhs, cond_cmp, cond_rhs)
  Example Usage: SELECT_ALL_COND(Person, id, >=, 1)

  Code Snippet:

```
#define SELECT_ALL_COND(name, cond_lhs, cond_cmp, cond_rhs) \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        if(row_trav->cond_lhs cond_cmp cond_rhs) \
        { \
            boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
                std::cout << field << "; "; \
            }); \
            std::cout << "\n"; \
        } \
    }
```

- **SELECT_ALL_COND_PAR** Operation: This is a parallel version of the above SELECT_ALL_COND query and is used to retrieve specific rows from the mentioned table based on the mentioned conditions. This generates the following code in-place using macro programming:
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - parallel for loop iterating through the vector of rows of the table to push the indices of the rows satisfying the conditions into a vector

after acquiring a mutex lock. The lock is used to serialize the prints because if not, the prints from the parallel threads can be undefined and haphazard due to overlaps.

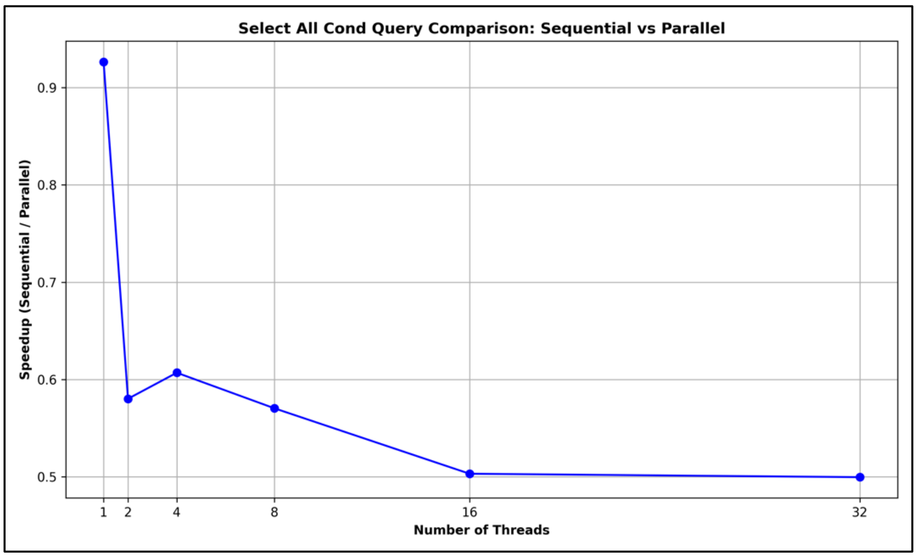
- for loop iterating through the above vector of indices to display the field values of the rows corresponding to the indices

Macro Signature: SELECT_ALL_COND_PAR(name, cond_lhs, cond_cmp, cond_rhs)
Example Usage: SELECT_ALL_COND_PAR(Person, id, >=, 1)

Code Snippet:

```
#define SELECT_ALL_COND_PAR(name, cond_lhs, cond_cmp, cond_rhs) \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    { \
        std::vector<int> print_ind; \
        print_ind.reserve(name.size()); \
        std::mutex print_ind_mutex; \
        parlay::parallel_for(0, name.size(), [&](size_t ind){ \
            if((name[ind])->cond_lhs cond_cmp cond_rhs) \
            { \
                print_ind_mutex.lock(); \
                print_ind.push_back(ind); \
                print_ind_mutex.unlock(); \
            } \
            return 0; \
        }); \
        \
        for(int i = 0; i<print_ind.size(); ++i) \
        { \
            boost::pfr::for_each_field(*(name[print_ind[i]]), [](const auto& field) { \
                std::cout << field << "; "; \
            }); \
            std::cout << "\n"; \
        } \
    }
```

The below graph summarizes the performance comparison between SELECT_ALL_COND and SELECT_ALL_COND_PAR for a query to retrieve the rows of the Person relation (of 10000 rows) with salary >= 250:



Here, the speedup is suboptimal (below 1) and the speedup decreases with the increase in the number of threads. The main reason is that as this query requires displaying of the rows, the implementation uses a lock-based data structure to serialize the prints. As these overheads due to lock-contention are significant, there are no benefits due to parallelism for this query.

- **SELECT_COLS** Operation: This is used to retrieve only the values for the mentioned columns for all the rows from the mentioned table. This generates the following code in-place using macro programming:
  - code to display the mentioned column names for the table
  - for loop iterating through the vector of rows of the table to display the field values corresponding to the mentioned column names for each of these rows

  Macro Signature: SELECT_COLS(name, cols)
  Example Usage: SELECT_COLS(Person,
      ((id))
      ((name))
)

  Code Snippet:

```
#define SELECT_COLS(name, cols) \
    BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, row_trav, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
    }
```

- **SELECT_COLS_COND** Operation: This is used to retrieve the values for the mentioned columns for all the rows satisfying the given condition from the mentioned table. This generates the following code in-place using macro programming:

- code to display the mentioned column names for the table
- for loop iterating through the vector of rows of the table to display the field values corresponding to the mentioned column names for each of these rows satisfying the mentioned conditions.

Macro Signature: SELECT_COLS_COND(name, cols, cond_lhs, cond_cmp, cond_rhs)
Example Usage: SELECT_COLS_COND(Person,
```
        ((id))
        ((name)),
        id, >=, 2
    )
```

Code Snippet:

```
#define SELECT_COLS_COND(name, cols, cond_lhs, cond_cmp, cond_rhs) \
    BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        if(row_trav->cond_lhs cond_cmp cond_rhs) \
        { \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, row_trav, BOOST_PP_VARIADIC_TO_SEQ cols)
\
            std::cout << "\n"; \
        } \
    }
```

- **SELECT_COLS_COND_PAR** Operation: This is a parallel version of the above SELECT_COLS_COND Query and is used to retrieve the values for the mentioned columns for all the rows satisfying the given condition from the mentioned table. This generates the following code in-place using macro programming:
  - code to display the mentioned column names for the table
  - parallel for loop iterating through the vector of rows of the table to push the indices of the rows satisfying the conditions into a vector after acquiring a mutex lock. The lock is used to serialize the prints because if not, the prints from the parallel threads can be undefined and haphazard due to overlaps.
  - for loop iterating through the above vector of indices to display the field values corresponding to the mentioned column names for each of the rows satisfying the mentioned conditions.

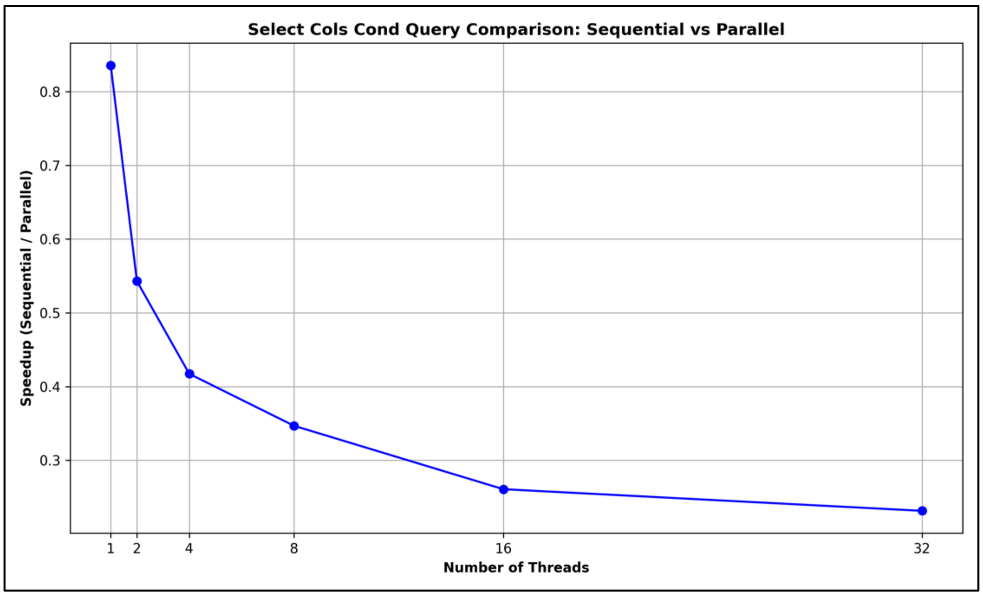
Macro Signature: SELECT_COLS_COND_PAR(name, cols, cond_lhs, cond_cmp, cond_rhs)
Example Usage: SELECT_COLS_COND_PAR (Person,

((id))
((name)),
id, >=, 2)

Code Snippet:

```
#define SELECT_COLS_COND_PAR(name, cols, cond_lhs, cond_cmp, cond_rhs) \
    BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
    std::cout << "\n"; \
    { \
        std::vector<int> print_ind; \
        print_ind.reserve(name.size()); \
        std::mutex print_ind_mutex; \
        parlay::parallel_for(0, name.size(), [&](size_t ind){ \
            if((name[ind])->cond_lhs cond_cmp cond_rhs) \
            { \
                print_ind_mutex.lock(); \
                print_ind.push_back(ind); \
                print_ind_mutex.unlock(); \
            } \
            return 0; \
        }); \
        \
        for(int i = 0; i<print_ind.size(); ++i) \
        { \
            if((name[print_ind[i]])->cond_lhs cond_cmp cond_rhs) \
            { \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, (name[print_ind[i]]),
BOOST_PP_VARIADIC_TO_SEQ cols) \
                std::cout << "\n"; \
            } \
        } \
    }
```

The below graph summarizes the performance comparison between SELECT_COLS_COND and SELECT_ COLS _COND_PAR for a query to retrieve the id, fname, age fields of the rows of the Person relation (of 10000 rows) with age >= 45:



Here, the speedup is suboptimal (below 1) and the speedup decreases with the increase in the number of threads. The main reason is that as this query requires displaying of the rows, the implementation uses a lock-based data structure to serialize the prints. As these overheads due to lock-contention are significant, there are no benefits due to parallelism for this query.

- **SELECT_PK_EQ_SEQ** Operation: This is used to search and retrieve the row from the mentioned table satisfying the equality condition on the primary key fields. This generates the following code in-place using macro programming:
  - code to display the column names for the table
  - for loop iterating through the vector of rows of the table to retrieve and display the field values of the row satisfying the equality condition on the primary key

  Macro Signature: SELECT_PK_EQ_SEQ(name, field_pk, cond_cmp, eq_val)
  Example Usage: SELECT_PK_EQ_SEQ(Person, id, ==, 7312)

  Code Snippet:

```
#define SELECT_PK_EQ_SEQ(name, field_pk, cond_cmp, eq_val) \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        if(row_trav->field_pk == eq_val) \
        { \
            boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
                std::cout << field << "; "; \
            }); \
            std::cout << "\n"; \
            break;\
        } \
    }
```

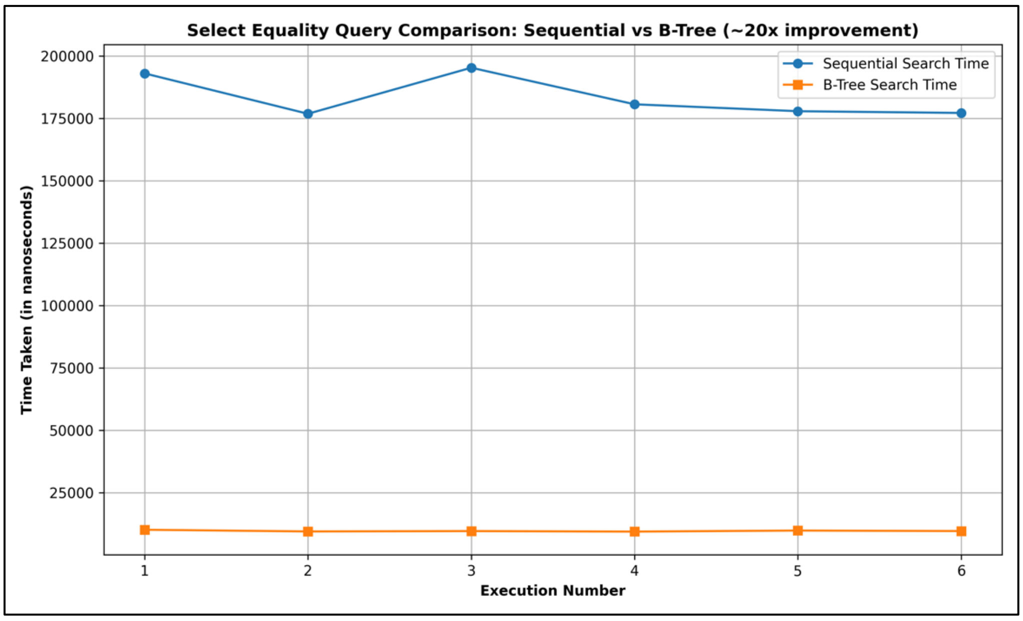
- **SELECT_PK_EQ** Operation: This is the B-Tree version of the above SELECT_PK_EQ_SEQ query and is used to search and retrieve the row from the mentioned table satisfying the equality condition on the primary key fields. This generates the following code in-place using macro programming:
    - code to call the *search_eq* function on the Primary key B-Tree of the table that searches and retrieves the row from the mentioned table satisfying the equality condition on the primary key
    - code to display the column names for the table
    - code to display the fields of the retrieved row

  Macro Signature: SELECT_PK_EQ(name, cond_lhs, cond_cmp, cond_rhs)
  Example Usage: SELECT_PK_EQ(Person, id, ==, 7312)

  Code Snippet:

```
#define SELECT_PK_EQ(name, cond_lhs, cond_cmp, cond_rhs) \
{ \
    row_##name *search_eq_row = new row_##name; \
    search_eq_row->cond_lhs = cond_rhs; \
    auto ret_search = pk_btree_##name.search_eq(search_eq_row, cmp_##name); \
    if(ret_search.first != nullptr) \
    { \
        for(auto ele_attr : attr_##name) \
        { \
            std::cout << ele_attr << "; "; \
        } \
        std::cout << "\n"; \
        boost::pfr::for_each_field(*(ret_search.second), [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    } \
    else \
    { \
        std::cout << "No matching Tuple found\n"; \
    } \
}
```

The below graph summarizes the performance comparison between SELECT_PK_EQ_SEQ and SELECT_PK_EQ for a query to retrieve the rows of the Person relation (of 10000 rows) with id == 7312:



It can be observed that the B-Tree implementation performs approximately 20x faster than the corresponding sequential implementation.

- **SELECT_ALL_RANGE** Operation: This is used to retrieve the rows from the mentioned table whose primary key field values are within the mentioned range. This generates the following code in-place using macro programming:
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the vector of rows of the table to display the field values for those rows whose primary key field values are within the mentioned range

  Macro Signature: SELECT_ALL_RANGE(name, field_pk, ge_val, le_val)
  Example Usage: SELECT_ALL_RANGE(Person, id, 1291, 1524)

  Code Snippet:

```
#define SELECT_ALL_RANGE(name, field_pk, ge_val, le_val) \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : name) \
    { \
        if(row_trav->field_pk >= ge_val && row_trav->field_pk <= le_val) \
        { \
            boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
                std::cout << field << "; "; \
            }); \
            std::cout << "\n"; \
        } \
    }
```

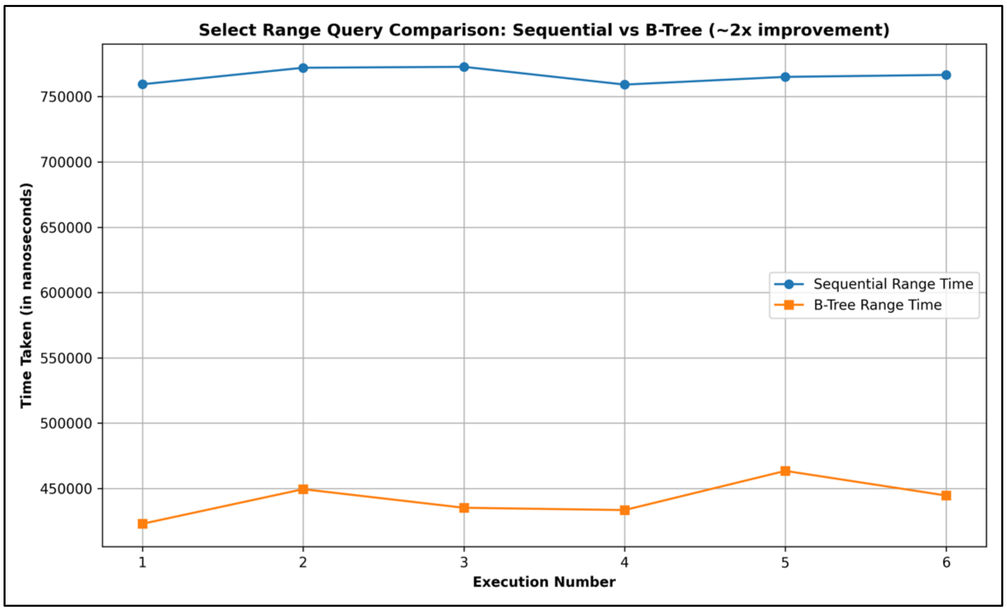
- **SELECT_PK_RANGE** Operation: This is the B-Tree version of the above SELECT_ALL_RANGE Query and is used to retrieve the rows from the mentioned table whose primary key field values are within the mentioned range. This generates the following code in-place using macro programming:
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - code to call the *range_traverse* function that traverses through the B-Tree and displays the field values for those rows whose primary key field values are within the mentioned range

  Macro Signature: SELECT_PK_RANGE(name, field_pk, ge_val, le_val)
  Example Usage: SELECT_PK_RANGE(Person, id, 1291, 1524)

  Code Snippet:

```
#define SELECT_PK_RANGE(name, field_pk, ge_val, le_val) \

    row_##name *range_ge_row = new row_##name; \

    range_ge_row->field_pk = ge_val; \

    row_##name *range_le_row = new row_##name; \

    range_le_row->field_pk = le_val; \

    for(auto ele_attr : attr_##name) \

    { \

        std::cout << ele_attr << "; "; \

    } \

    std::cout << "\n"; \

    pk_btree_##name.range_traverse(range_ge_row, range_le_row, cmp_##name); \
```

The below graph summarizes the performance comparison between SELECT_ALL_RANGE and SELECT_PK_RANGE for a query to retrieve the rows of the Person relation (of 10000 rows) with id >= 1291 and id <= 1524:



It can be observed that the Select Range Query using the B-Tree implementation performs approximately 2x faster than the corresponding sequential implementation.

- **UPDATE_ALL** Operation: This is used to update the values of the mentioned columns of all the rows to the values passed as part of the argument. This generates the following code in-place using macro programming:
  - for loop iterating through the vector of rows of the table to update the values of the mentioned columns to the values passed as part of the argument

  Macro Signature: UPDATE_ALL(name, upd_values)
  Example Usage: UPDATE_ALL(Person,
  ```
      ((id, -1))
      ((mob_no, "+91"))
  )
  ```

- **UPDATE_COND** Operation: This is used to update the values of the mentioned columns of the rows satisfying the given condition, to the values passed as part of the argument. This generates the following code in-place using macro programming:
  - for loop iterating through the vector of rows of the table to update the values of the mentioned columns of the rows satisfying the given condition, to the values passed as part of the argument

  Macro Signature: UPDATE_COND(name, upd_values, cond_lhs, cond_cmp, cond_rhs)
  Example Usage: UPDATE_COND(Person,
  ```
      ((id, 15))
      ((mob_no, "+91")),
  ```

        id, ==, 14
)

- **DELETE_ALL_ROWS** Operation: This is used to delete all the rows of the mentioned table. This generates the following code in-place using macro programming:
    - for loop iterating through the vector of rows to delete the memory allocated for these rows
    - calling the clear operation on the vector of rows

  Macro Signature: DELETE_ALL_ROWS(name)
  Example Usage: DELETE_ALL_ROWS(Person)

- **DELETE_COND** Operation: This is used to delete the rows of the mentioned table that satisfy the given condition. This generates the following code in-place using macro programming:
    - while loop iterating through the vector of rows to delete the memory allocated for the row and to erase it from the vector if they satisfy the given condition
    - above while loop ensures to safely handle the deletion of elements of a vector while iterating through it by updating the iterators appropriately

  Macro Signature: DELETE_COND(name, cond_lhs, cond_cmp, cond_rhs)
  Example Usage: DELETE_COND(Person, id, >=, 2)

- **ORDER_BY_ASC** Operation: This is used to order the table in ascending order of the mentioned columns and display the rows after ordering. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to display the field values for those rows

  Macro Signature: ORDER_BY_ASC(name, cols)
  Example Usage: ORDER_BY_ASC(Person,
        ((fname))
        ((lname))
        ((id))
)

  Code Snippet:

```
#define ORDER_BY_ASC(name, cols) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : sort_vec) \
    { \
        boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    } \
    std::cout << "\n"; \
}
```

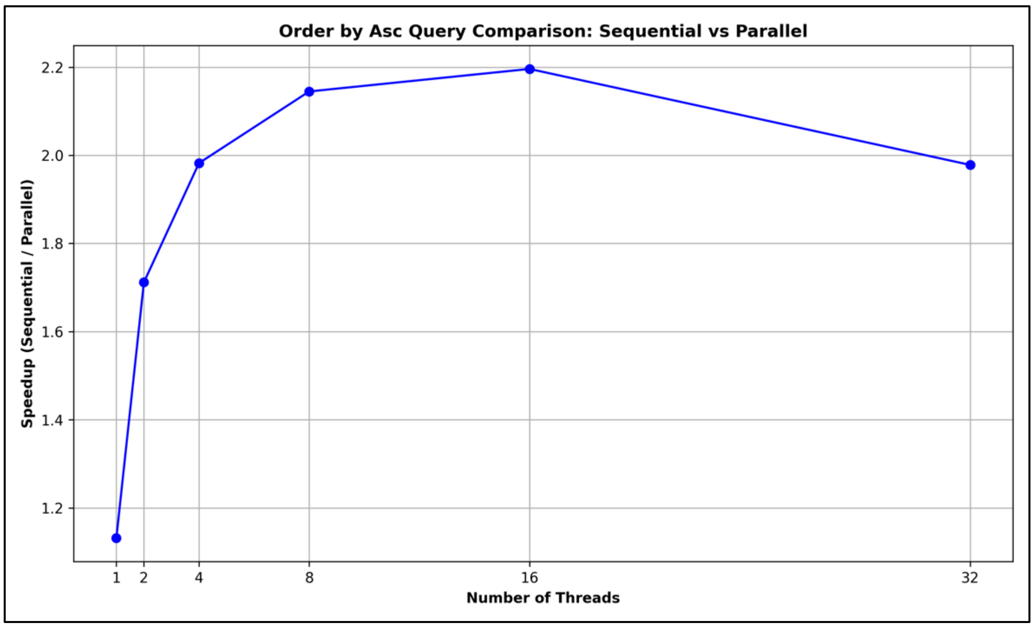
- **ORDER_BY_ASC_PAR** Operation: This is a parallel version of the above ORDER_BY_ASC Query and is used to order the table in ascending order of the mentioned columns and display the rows after ordering. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to display the field values for those rows

  Macro Signature: ORDER_BY_ASC_PAR(name, cols)
  Example Usage: ORDER_BY_ASC_PAR(Person,
  ((fname))
  ((lname))
  ((id))
)

  Code Snippet:

```
#define ORDER_BY_ASC_PAR(name, cols) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : sort_seq) \
    { \
        boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    } \
    std::cout << "\n"; \
}
```

The below graph summarizes the performance comparison between ORDER_BY_ASC and ORDER_BY_ASC_PAR for a query to order the rows of the Person relation (of 10000 rows) based on (fname, lname, id) fields:



It can be observed that the Parallel version achieved a maximum speedup of 2.2 for 16 threads, thereby denoting that there were some benefits due to parallelism.

- **ORDER_BY_DESC** Operation: This is used to order the table in descending order of the mentioned columns and display the rows after ordering. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to display the field values for those rows

  Macro Signature: ORDER_BY_DESC(name, cols)

Example Usage: ORDER_BY_DESC(Person,
    ((fname))
    ((lname))
    ((id))
)

Code Snippet:

```
#define ORDER_BY_DESC(name, cols) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_DESC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : sort_vec) \
    { \
        boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    } \
    std::cout << "\n"; \
}
```

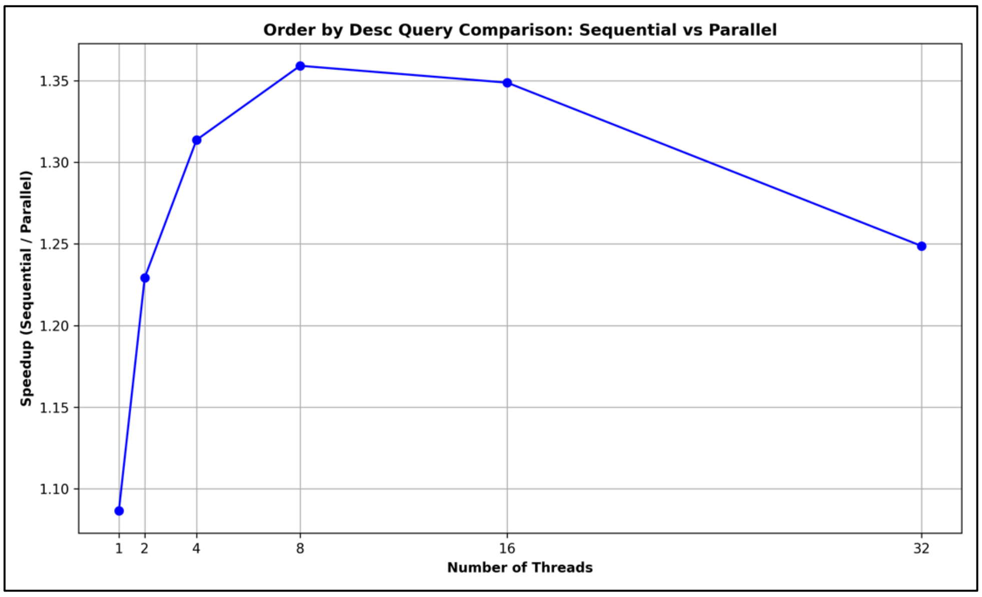
- **ORDER_BY_DESC_PAR** Operation: This is a parallel version of the above ORDER_BY_ DESC Query and is used to order the table in descending order of the mentioned columns and display the rows after ordering. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to display the field values for those rows

  Macro Signature: ORDER_BY_ DESC _PAR(name, cols)
  Example Usage: ORDER_BY_ DESC _PAR(Person,
      ((fname))
      ((lname))
      ((id)))

Code Snippet:

```
#define ORDER_BY_DESC_PAR(name, cols) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_DESC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    for(auto ele_attr : attr_##name) \
    { \
        std::cout << ele_attr << "; "; \
    } \
    std::cout << "\n"; \
    for(auto row_trav : sort_seq) \
    { \
        boost::pfr::for_each_field(*row_trav, [](const auto& field) { \
            std::cout << field << "; "; \
        }); \
        std::cout << "\n"; \
    } \
    std::cout << "\n"; \
}
```

The below graph summarizes the performance comparison between ORDER_BY_ DESC and ORDER_BY_ DESC _PAR for a query to order the rows of the Person relation (of 10000 rows) based on (salary, id) fields:



It can be observed that the Parallel version achieved a maximum speedup of 1.36 for 8 threads, thereby denoting that there were some benefits due to parallelism.

- **GROUP_BY_COUNT** Operation: This is used to group the table based on the mentioned columns and aggregate the counts of these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate their counts

  Macro Signature: GROUP_BY_COUNT(name, cols)

  Example Usage: GROUP_BY_COUNT(Person,
  ((lname))
)

  Code Snippet:

```
#define GROUP_BY_COUNT(name, cols) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_vec.size() != 0) \
    { \
        std::cout << "Count; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_vec[0]; \
        long long int curr_count = 1; \
        for(int i = 1; i<sort_vec.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_vec[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                ++curr_count; \
            } \
            else \
            { \
                std::cout << curr_count << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                curr_count = 1; \
                first_r = sort_vec[i]; \
            } \
        } \
        std::cout << curr_count << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_COUNT_PAR** Operation: This is parallel version 1 of the above GROUP_BY_COUNT Query and is used to group the table based on the mentioned columns and aggregate the counts of these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate their counts

  Macro Signature: GROUP_BY_COUNT_PAR(name, cols)
  Example Usage: GROUP_BY_COUNT_PAR(Person,
      ((lname))
)

  Code Snippet:

```
#define GROUP_BY_COUNT_PAR(name, cols) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Count; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_seq[0]; \
        long long int curr_count = 1; \
        for(int i = 1; i<sort_seq.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_seq[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                ++curr_count; \
            } \
            else \
            { \
                std::cout << curr_count << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                curr_count = 1; \
                first_r = sort_seq[i]; \
            } \
        } \
        std::cout << curr_count << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

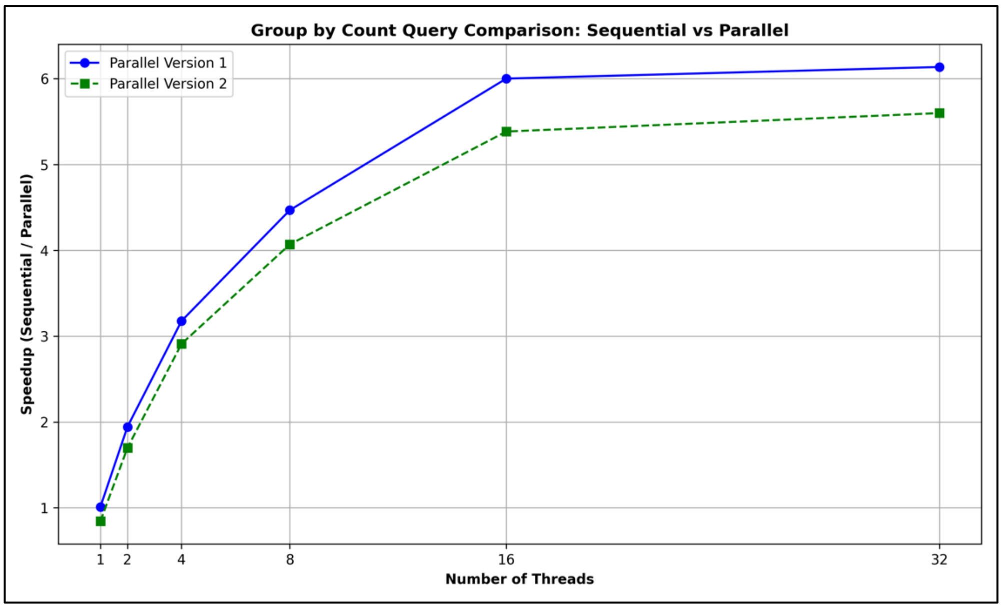
- **GROUP_BY_COUNT_PAR2** Operation: This is parallel version 2 of the above GROUP_BY_COUNT Query and is used to group the table based on the mentioned columns and aggregate the counts of these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - code to retrieve the indices of the rows of the table that have unique values for the mentioned fields using parallel tabulate and parallel filter
  - code to retrieve the counts of the groups corresponding to the above unique rows using parallel tabulate
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop to display groups and their aggregated counts

  Macro Signature: GROUP_BY_COUNT_PAR2(name, cols)
  Example Usage: GROUP_BY_COUNT_PAR2(Person,
      ((lname))
)

  Code Snippet:

```
#define GROUP_BY_COUNT_PAR2(name, cols) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    auto ind_seq = parlay::tabulate(sort_seq.size(), [](int i) -> int { return i; }); \
    auto boundaries_seq = parlay::filter(ind_seq, [&](auto ele) { return (ele == 0
BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY_PAR2, (sort_seq[ele], sort_seq[ele-1]),
BOOST_PP_VARIADIC_TO_SEQ cols)); }); \
    auto group_count_seq = parlay::tabulate(boundaries_seq.size(), [&](int i) -> int { \
        int st_ind = boundaries_seq[i]; \
        int en_ind = -1; \
        if(i + 1 == boundaries_seq.size()) \
        { \
            en_ind = sort_seq.size(); \
        } \
        else \
        { \
            en_ind = boundaries_seq[i+1]; \
        } \
        auto count_v = en_ind - st_ind; \
        return count_v; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Count; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        for(int i = 0; i<boundaries_seq.size(); ++i) \
        { \
            std::cout << group_count_seq[i] << "; "; \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, sort_seq[boundaries_seq[i]],
BOOST_PP_VARIADIC_TO_SEQ cols) \
            std::cout << "\n"; \
        } \
        std::cout << "\n\n"; \
    } \
}
```

The below graph summarizes the performance comparison between GROUP_BY_COUNT, GROUP_BY_COUNT_PAR, and GROUP_BY_COUNT_PAR2 for a query to group the rows of the Person relation (of 10000 rows) based on (lname) and retrieve their counts:



It can be observed that the Parallel Version 1 performs slightly better than the Parallel Version 2. This could be because the overheads introduced by version 2's algorithm which requires the creation of additional data structures using tabulate and filter is quite considerable that it is outweighing any benefits.

- **GROUP_BY_MIN** Operation: This is used to group the table based on the mentioned columns and aggregate the minimum of the mentioned column for these groups. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the minimum of the mentioned column

  Macro Signature: GROUP_BY_MIN(name, cols, min_col)
  Example Usage: GROUP_BY_MIN(Person,
      ((country)), salary
)

  Code Snippet:

```
#define GROUP_BY_MIN(name, cols, min_col) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_vec.size() != 0) \
    { \
        std::cout << "Min_" << TO_STRING_EXPAND(min_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_vec[0]; \
        auto min_val = sort_vec[0]->min_col; \
        for(int i = 1; i<sort_vec.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_vec[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                if(sort_vec[i]->min_col < min_val) \
                { \
                    min_val = sort_vec[i]->min_col; \
                } \
            } \
            else \
            { \
                std::cout << min_val << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                min_val = sort_vec[i]->min_col; \
                first_r = sort_vec[i]; \
            } \
        } \
        std::cout << min_val << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_MIN_PAR** Operation: This is parallel version 1 of the above GROUP_BY_MIN Query and is used to group the table based on the mentioned columns and aggregate the minimum of the mentioned column for these groups. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the minimum of the mentioned column

  Macro Signature: GROUP_BY_MIN_PAR(name, cols, min_col)
  Example Usage: GROUP_BY_MIN_PAR(Person,
      ((country)), salary
)

  Code Snippet:

```
#define GROUP_BY_MIN_PAR(name, cols, min_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Min_" << TO_STRING_EXPAND(min_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_seq[0]; \
        auto min_val = sort_seq[0]->min_col; \
        for(int i = 1; i<sort_seq.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_seq[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                if(sort_seq[i]->min_col < min_val) \
                { \
                    min_val = sort_seq[i]->min_col; \
                } \
            } \
            else \
            { \
                std::cout << min_val << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                min_val = sort_seq[i]->min_col; \
                first_r = sort_seq[i]; \
            } \
        } \
        std::cout << min_val << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

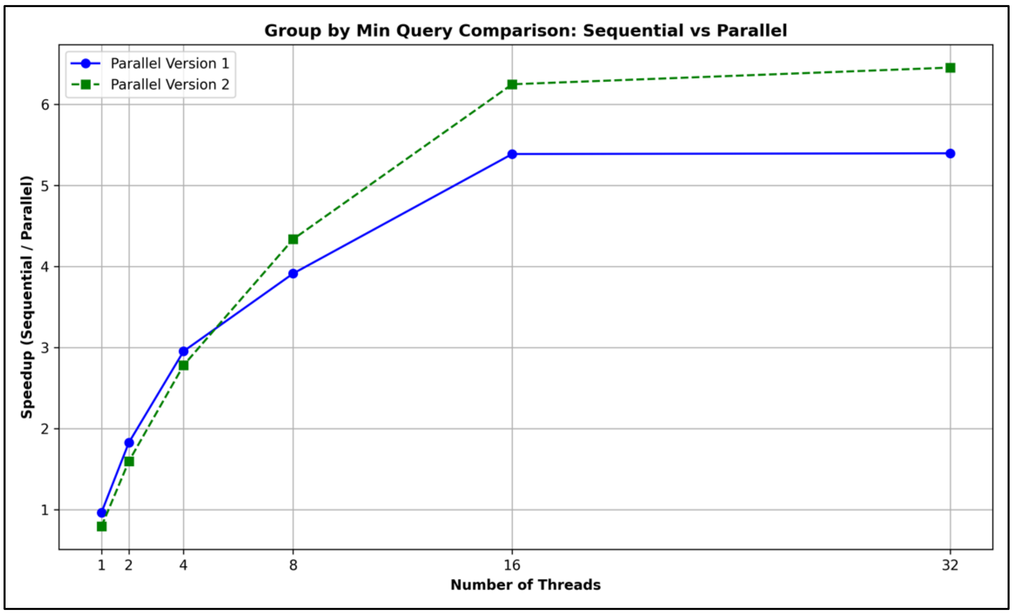
- **GROUP_BY_MIN_PAR2** Operation: This is parallel version 2 of the above GROUP_BY_MIN Query and is used to group the table based on the mentioned columns and aggregate the minimum of the mentioned column for these groups. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - code to retrieve the indices of the rows of the table that have unique values for the mentioned fields using parallel tabulate and parallel filter
    - code to retrieve the minimum of the mentioned column for the groups corresponding to the above unique rows using parallel tabulate and parallel reduce
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop to display groups and their aggregated minimum

  Macro Signature: GROUP_BY_MIN_PAR2(name, cols, min_col)
  Example Usage: GROUP_BY_MIN_PAR2(Person,
  ((country)), salary)

Code Snippet:

```
#define GROUP_BY_MIN_PAR2(name, cols, min_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    auto ind_seq = parlay::tabulate(sort_seq.size(), [](int i) -> int { return i; }); \
    auto min_col_seq = parlay::tabulate(sort_seq.size(), [&](int i) { return sort_seq[i]-
>min_col; }); \
    auto boundaries_seq = parlay::filter(ind_seq, [&](auto ele) { return (ele == 0
BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY_PAR2, (sort_seq[ele], sort_seq[ele-1]),
BOOST_PP_VARIADIC_TO_SEQ cols)); }); \
    auto group_min_seq = parlay::tabulate(boundaries_seq.size(), [&](int i) -> int { \
        int st_ind = boundaries_seq[i]; \
        int en_ind = -1; \
        if(i + 1 == boundaries_seq.size()) \
        { \
            en_ind = sort_seq.size(); \
        } \
        else \
        { \
            en_ind = boundaries_seq[i+1]; \
        } \
        auto m = parlay::make_monoid([](long long int left, long long int right) { \
            if(left <= right) \
            { \
                return left; \
            } \
            return right; \
        } ,LLONG_MAX); \
        auto min_v = parlay::reduce(min_col_seq.cut(st_ind, en_ind), m); \
        return min_v; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Min_" << TO_STRING_EXPAND(min_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        for(int i = 0; i<boundaries_seq.size(); ++i) \
        { \
            std::cout << group_min_seq[i] << "; "; \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, sort_seq[boundaries_seq[i]],
BOOST_PP_VARIADIC_TO_SEQ cols) \
            std::cout << "\n"; \
        } \
        std::cout << "\n\n"; \
    } \
}
```

The below graph summarizes the performance comparison between GROUP_BY_MIN, GROUP_BY_MIN_PAR2, and GROUP_BY_MIN_PAR2 for a query to group the rows of the Person relation (of 10000 rows) based on (country) and retrieve the minimum salary for these groups:



It can observed that the Parallel Version 2 performs better than the Parallel Version 1 with the increase in number of threads. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **GROUP_BY_MAX** Operation: This is used to group the table based on the mentioned columns and aggregate the maximum of the mentioned column for these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the maximum of the mentioned column

  Macro Signature: GROUP_BY_MAX(name, cols, max_col)
  Example Usage: GROUP_BY_MAX(Person,
  ((country)), age
)

Code Snippet:

```
#define GROUP_BY_MAX(name, cols, max_col) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_vec.size() != 0) \
    { \
        std::cout << "Max_" << TO_STRING_EXPAND(max_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_vec[0]; \
        auto max_val = sort_vec[0]->max_col; \
        for(int i = 1; i<sort_vec.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_vec[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                if(sort_vec[i]->max_col > max_val) \
                { \
                    max_val = sort_vec[i]->max_col; \
                } \
            } \
            else \
            { \
                std::cout << max_val << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                max_val = sort_vec[i]->max_col; \
                first_r = sort_vec[i]; \
            } \
        } \
        std::cout << max_val << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_MAX_PAR** Operation: This is parallel version 1 of the above GROUP_BY_MAX Query and is used to group the table based on the mentioned columns and aggregate the maximum of the mentioned column for these groups. This generates the following code in-place using macro programming:

- code to create a copy of the vector of rows of the table
- code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
- for loop iterating through the vector of attribute strings to display the column names for the table
- for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the maximum of the mentioned column

Macro Signature: GROUP_BY_MAX_PAR(name, cols, max_col)
Example Usage: GROUP_BY_MAX_PAR(Person,
    ((country)), age
)

Code Snippet:

```
#define GROUP_BY_MAX_PAR(name, cols, max_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Max_" << TO_STRING_EXPAND(max_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_seq[0]; \
        auto max_val = sort_seq[0]->max_col; \
        for(int i = 1; i<sort_seq.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_seq[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                if(sort_seq[i]->max_col > max_val) \
                { \
                    max_val = sort_seq[i]->max_col; \
                } \
            } \
            else \
            { \
                std::cout << max_val << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                max_val = sort_seq[i]->max_col; \
                first_r = sort_seq[i]; \
            } \
        } \
        std::cout << max_val << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

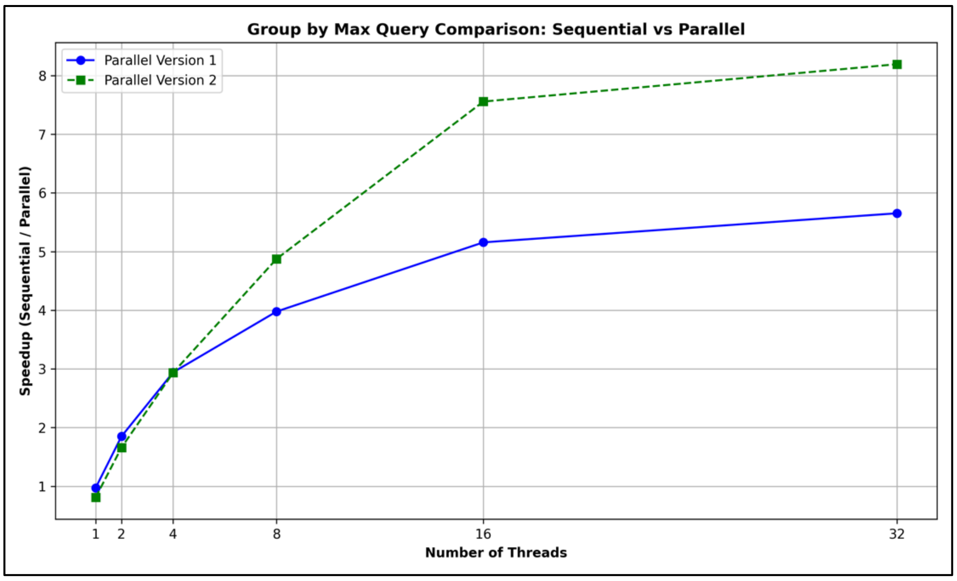
- **GROUP_BY_MAX_PAR2** Operation: This is parallel version 2 of the above GROUP_BY_MAX Query and is used to group the table based on the mentioned columns and aggregate the maximum of the mentioned column for these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - code to retrieve the indices of the rows of the table that have unique values for the mentioned fields using parallel tabulate and parallel filter
  - code to retrieve the maximum of the mentioned column for the groups corresponding to the above unique rows using parallel tabulate and parallel reduce
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop to display groups and their aggregated maximum

  Macro Signature: GROUP_BY_MAX_PAR2(name, cols, max_col)
  Example Usage: GROUP_BY_MAX_PAR2(Person,
      ((country)), age
)

  Code Snippet:

```
#define GROUP_BY_MAX_PAR2(name, cols, max_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    auto ind_seq = parlay::tabulate(sort_seq.size(), [](int i) -> int { return i; }); \
    auto max_col_seq = parlay::tabulate(sort_seq.size(), [&](int i) { return sort_seq[i]-
>max_col; }); \
    auto boundaries_seq = parlay::filter(ind_seq, [&](auto ele) { return (ele == 0
BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY_PAR2, (sort_seq[ele], sort_seq[ele-1]),
BOOST_PP_VARIADIC_TO_SEQ cols)); }); \
    auto group_max_seq = parlay::tabulate(boundaries_seq.size(), [&](int i) -> int { \
        int st_ind = boundaries_seq[i]; \
        int en_ind = -1; \
        if(i + 1 == boundaries_seq.size()) \
        { \
            en_ind = sort_seq.size(); \
        } \
        else \
        { \
            en_ind = boundaries_seq[i+1]; \
        } \
        auto m = parlay::make_monoid([](long long int left, long long int right) { \
            if(left >= right) \
            { \
                return left; \
            } \
            return right; \
        } ,LLONG_MIN); \
        auto max_v = parlay::reduce(max_col_seq.cut(st_ind, en_ind), m); \
        return max_v; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Max_" << TO_STRING_EXPAND(max_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        for(int i = 0; i<boundaries_seq.size(); ++i) \
        { \
            std::cout << group_max_seq[i] << "; "; \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, sort_seq[boundaries_seq[i]],
BOOST_PP_VARIADIC_TO_SEQ cols) \
            std::cout << "\n"; \
        } \
        std::cout << "\n\n"; \
    } \
}
```

The below graph summarizes the performance comparison between GROUP_BY_MAX, GROUP_BY_MAX_PAR, and GROUP_BY_MAX_PAR2 for a query to group the rows of the Person relation (of 10000 rows) based on (country) and retrieve the maximum age for these groups:



It can be observed that the Parallel Version 2 performs better than the Parallel Version 1 with the increase in number of threads. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **GROUP_BY_SUM** Operation: This is used to group the table based on the mentioned columns and aggregate the sum of the mentioned column for these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the sum of the mentioned column

  Macro Signature: GROUP_BY_SUM(name, cols, sum_col)
  Example Usage: GROUP_BY_SUM(Person,
  ((country)), salary
)

  Code Snippet:

```
#define GROUP_BY_SUM(name, cols, sum_col) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_vec.size() != 0) \
    { \
        std::cout << "Sum_" << TO_STRING_EXPAND(sum_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_vec[0]; \
        long long int curr_sum = sort_vec[0]->sum_col; \
        for(int i = 1; i<sort_vec.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_vec[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                curr_sum += sort_vec[i]->sum_col; \
            } \
            else \
            { \
                std::cout << curr_sum << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                curr_sum = sort_vec[i]->sum_col; \
                first_r = sort_vec[i]; \
            } \
        } \
        std::cout << curr_sum << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_SUM_PAR** Operation: This is parallel version 1 of the above GROUP_BY_SUM Query and is used to group the table based on the mentioned columns and aggregate the sum of the mentioned column for these groups. This generates the following code in-place using macro programming:
    - code to create a copy of the vector of rows of the table
    - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
    - for loop iterating through the vector of attribute strings to display the column names for the table
    - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the sum of the mentioned column

  Macro Signature: GROUP_BY_SUM_PAR(name, cols, sum_col)

Example Usage: GROUP_BY_SUM_PAR(Person,
    ((country)), salary
)

Code Snippet:

```
#define GROUP_BY_SUM_PAR(name, cols, sum_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Sum_" << TO_STRING_EXPAND(sum_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_seq[0]; \
        long long int curr_sum = sort_seq[0]->sum_col; \
        for(int i = 1; i<sort_seq.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_seq[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                curr_sum += sort_seq[i]->sum_col; \
            } \
            else \
            { \
                std::cout << curr_sum << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \

                std::cout << "\n"; \
                curr_sum = sort_seq[i]->sum_col; \
                first_r = sort_seq[i]; \
            } \
        } \
        std::cout << curr_sum << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_SUM_PAR2** Operation: This is parallel version 2 of the above GROUP_BY_SUM Query and is used to group the table based on the mentioned columns and aggregate the sum of the mentioned column for these groups. This generates the following code in-place using macro programming:

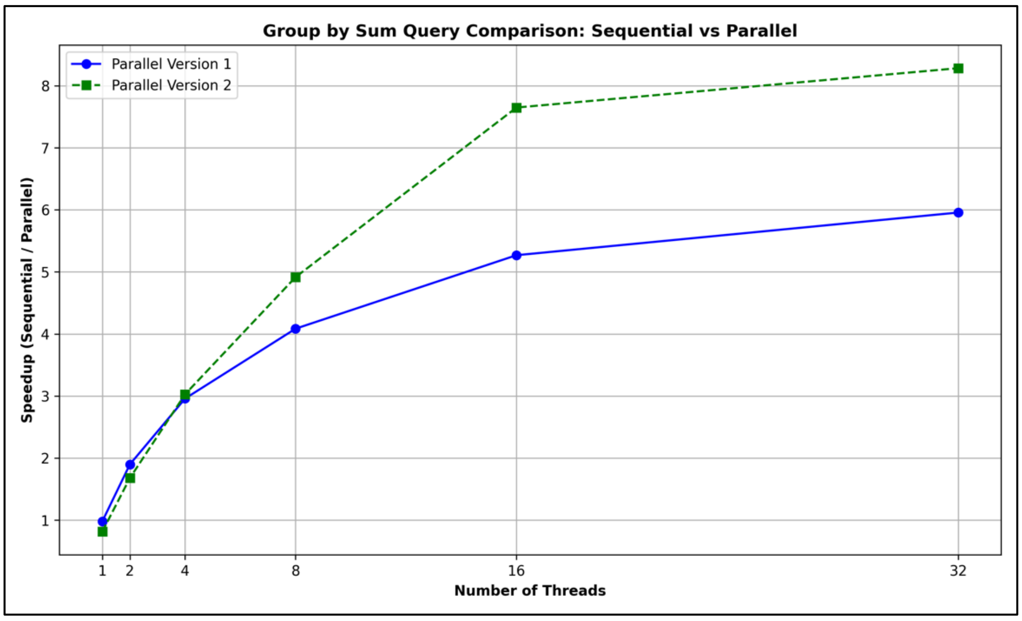
- code to create a copy of the vector of rows of the table
- code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
- code to retrieve the indices of the rows of the table that have unique values for the mentioned fields using parallel tabulate and parallel filter
- code to retrieve the sum of the mentioned column for the groups corresponding to the above unique rows using parallel tabulate and parallel reduce
- for loop iterating through the vector of attribute strings to display the column names for the table
- for loop to display groups and their aggregated sum

Macro Signature: GROUP_BY_SUM_PAR2(name, cols, sum_col)
Example Usage: GROUP_BY_SUM_PAR2(Person,
    ((country)), salary
)

Code Snippet:

```
#define GROUP_BY_SUM_PAR2(name, cols, sum_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    auto ind_seq = parlay::tabulate(sort_seq.size(), [](int i) -> int { return i; }); \
    auto sum_col_seq = parlay::tabulate(sort_seq.size(), [&](int i) { return sort_seq[i]-
>sum_col; }); \
    auto boundaries_seq = parlay::filter(ind_seq, [&](auto ele) { return (ele == 0
BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY_PAR2, (sort_seq[ele], sort_seq[ele-1]),
BOOST_PP_VARIADIC_TO_SEQ cols)); }); \
    auto group_count_seq = parlay::tabulate(boundaries_seq.size(), [&](int i) -> int { \
        int st_ind = boundaries_seq[i]; \
        int en_ind = -1; \
        if(i + 1 == boundaries_seq.size()) \
        { \
            en_ind = sort_seq.size(); \
        } \
        else \
        { \
            en_ind = boundaries_seq[i+1]; \
        } \
        auto m = parlay::make_monoid([](long long int left, long long int right) {return
(left + right);} ,0); \
        auto sum_v = parlay::reduce(sum_col_seq.cut(st_ind, en_ind), m); \
        return sum_v; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Sum_" << TO_STRING_EXPAND(sum_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        for(int i = 0; i<boundaries_seq.size(); ++i) \
        { \
            std::cout << group_count_seq[i] << "; "; \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, sort_seq[boundaries_seq[i]],
BOOST_PP_VARIADIC_TO_SEQ cols) \
            std::cout << "\n"; \
        } \
        std::cout << "\n\n"; \
    } \
}
```

The below graph summarizes the performance comparison between GROUP_BY_SUM , GROUP_BY_SUM_PAR, and GROUP_BY_SUM_PAR2 for a query to group the rows of the Person relation (of 10000 rows) based on (country) and retrieve the sum of the salary for these groups:



It can be observed that the Parallel Version 2 performs better than the Parallel Version 1 with the increase in number of threads. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

- **GROUP_BY_AVG** Operation: This is used to group the table based on the mentioned columns and aggregate the average of the mentioned column for these groups. This generates the following code in-place using macro programming:

- code to create a copy of the vector of rows of the table
- code to call the sequential sort function on the above copy vector with a comparator generated for comparing the mentioned columns
- for loop iterating through the vector of attribute strings to display the column names for the table
- for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the average of the mentioned column

Macro Signature: GROUP_BY_AVG(name, cols, avg_col)
Example Usage: GROUP_BY_AVG(Person,
    ((country)), salary
)

Code Snippet:

```
#define GROUP_BY_AVG(name, cols, avg_col) \
{ \
    std::vector<row_##name*> sort_vec(name.begin(), name.end()); \
    std::sort(sort_vec.begin(), sort_vec.end(), [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        return false; \
    }); \
    if(sort_vec.size() != 0) \
    { \
        std::cout << "Avg_" << TO_STRING_EXPAND(avg_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_vec[0]; \
        double curr_sum = sort_vec[0]->avg_col; \
        long long int curr_count = 1; \
        for(int i = 1; i<sort_vec.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_vec[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                curr_sum += sort_vec[i]->avg_col; \
                ++curr_count; \
            } \
            else \
            { \
                std::cout << (curr_sum/curr_count) << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
                std::cout << "\n"; \
                curr_sum = sort_vec[i]->avg_col; \
                curr_count = 1; \
                first_r = sort_vec[i]; \
            } \
        } \
        std::cout << (curr_sum/curr_count) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

- **GROUP_BY_AVG_PAR** Operation: This is parallel version 1 of the above GROUP_BY_AVG Query and is used to group the table based on the mentioned columns and aggregate the average of the mentioned column for these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop iterating through the sorted vector of rows of the table to identify groups and aggregate the average of the mentioned column

  Macro Signature: GROUP_BY_AVG_PAR(name, cols, avg_col)
  Example Usage: GROUP_BY_AVG_PAR(Person,
  ((country)), salary
)

  Code Snippet:

```
#define GROUP_BY_AVG_PAR(name, cols, avg_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Avg_" << TO_STRING_EXPAND(avg_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        row_##name* first_r = sort_seq[0]; \
        double curr_sum = sort_seq[0]->avg_col; \
        long long int curr_count = 1; \
        for(int i = 1; i<sort_seq.size(); ++i) \
        { \
            if(1 BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY, sort_seq[i],
BOOST_PP_VARIADIC_TO_SEQ cols)) \
            { \
                curr_sum += sort_seq[i]->avg_col; \
                ++curr_count; \
            } \
            else \
            { \
                std::cout << (curr_sum/curr_count) << "; "; \
                BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ
cols) \
                std::cout << "\n"; \
                curr_sum = sort_seq[i]->avg_col; \
                curr_count = 1; \
                first_r = sort_seq[i]; \
            } \
        } \
        std::cout << (curr_sum/curr_count) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, first_r, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n\n"; \
    } \
}
```

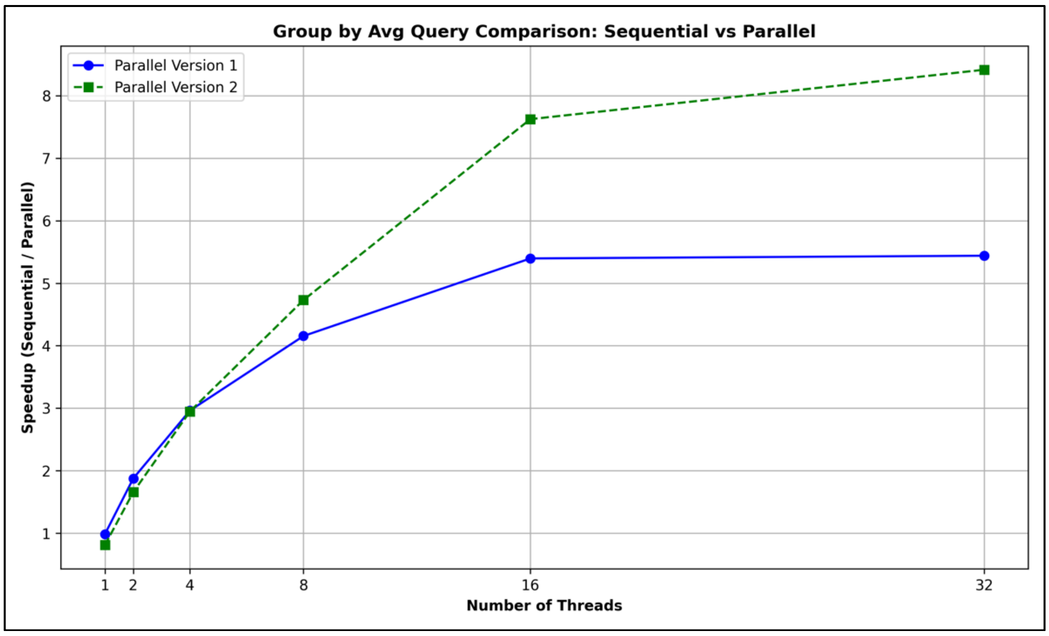
- **GROUP_BY_AVG_PAR2** Operation: This is parallel version 2 of the above GROUP_BY_AVG Query and is used to group the table based on the mentioned columns and aggregate the average of the mentioned column for these groups. This generates the following code in-place using macro programming:
  - code to create a copy of the vector of rows of the table
  - code to call the parallel merge sort function on the above copy vector with a comparator generated for comparing the mentioned columns
  - code to retrieve the indices of the rows of the table that have unique values for the mentioned fields using parallel tabulate and parallel filter
  - code to retrieve the average of the mentioned column for the groups corresponding to the above unique rows using parallel tabulate and parallel reduce
  - for loop iterating through the vector of attribute strings to display the column names for the table
  - for loop to display groups and their aggregated average

  Macro Signature: GROUP_BY_AVG_PAR2(name, cols, avg_col)
  Example Usage: GROUP_BY_AVG_PAR2(Person,
      ((country)), salary
)

  Code Snippet:

```
#define GROUP_BY_AVG_PAR2(name, cols, avg_col) \
{ \
    parlay::sequence<row_##name*> sort_seq(name.begin(), name.end()); \
    merge_sort(sort_seq, [](row_##name* left, row_##name* right){ \
        BOOST_PP_SEQ_FOR_EACH(EXPAND_FIELD_COMPARISON_ASC, _, BOOST_PP_VARIADIC_TO_SEQ cols)
\
        return false; \
    }); \
    auto ind_seq = parlay::tabulate(sort_seq.size(), [](int i) -> int { return i; }); \
    auto avg_col_seq = parlay::tabulate(sort_seq.size(), [&](int i) { return sort_seq[i]-
>avg_col; }); \
    auto boundaries_seq = parlay::filter(ind_seq, [&](auto ele) { return (ele == 0
BOOST_PP_SEQ_FOR_EACH(GENERATE_EQUALITY_CMP_GROUP_BY_PAR2, (sort_seq[ele], sort_seq[ele-1]),
BOOST_PP_VARIADIC_TO_SEQ cols)); }); \
    auto group_avg_seq = parlay::tabulate(boundaries_seq.size(), [&](int i) -> int { \
        int st_ind = boundaries_seq[i]; \
        int en_ind = -1; \
        if(i + 1 == boundaries_seq.size()) \
        { \
            en_ind = sort_seq.size(); \
        } \
        else \
        { \
            en_ind = boundaries_seq[i+1]; \
        } \
        auto m = parlay::make_monoid([](long long int left, long long int right) {return
(left + right);} ,0); \
        double sum_v = parlay::reduce(avg_col_seq.cut(st_ind, en_ind), m); \
        auto count_v = en_ind - st_ind; \
        double avg_v = sum_v/count_v; \
        return avg_v; \
    }); \
    if(sort_seq.size() != 0) \
    { \
        std::cout << "Avg_" << TO_STRING_EXPAND(avg_col) << "; "; \
        BOOST_PP_SEQ_FOR_EACH(DISPLAY_COL_NAMES, _, BOOST_PP_VARIADIC_TO_SEQ cols) \
        std::cout << "\n"; \
        for(int i = 0; i<boundaries_seq.size(); ++i) \
        { \
            std::cout << group_avg_seq[i] << "; "; \
            BOOST_PP_SEQ_FOR_EACH(DISPLAY_VALS_COLS, sort_seq[boundaries_seq[i]],
BOOST_PP_VARIADIC_TO_SEQ cols) \
            std::cout << "\n"; \
        } \
        std::cout << "\n\n"; \
    } \
}
```

The below graph summarizes the performance comparison between GROUP_BY_AVG, GROUP_BY_AVG_PAR, and GROUP_BY_AVG_PAR2 for a query to group the rows of the Person relation (of 10000 rows) based on (country) and retrieve the average of the salary for these groups:



It can be observed that the Parallel Version 2 performs better than the Parallel Version 1 with the increase in number of threads. This could be because the parallel version 2 algorithm using primitives like tabulate, filter and reduce inherently contains more scope for parallelism as compared to parallel version 1 which is essentially sequential after the initial parallel sort.

2. Auxiliary Data Structures and Functions:

- **B-Tree:** A B-Tree data structure was created to use as an indexing data structure for the table based on its primary key fields
    - **BTreeNode Class:** Represents a node of the B-Tree
        - It contains the following fields: an array of keys, the minimum degree, an array of child pointers, current number of keys and Boolean to denote a leaf
        - If ‘t’ is the minimum degree, each node contains between t and 2*t-1 keys and 2*t child pointers
        - It includes a constructor, destructor, function to traverse through the B-Tree nodes, function to search for a key based on equality condition, function to insert into a non-full node, function to split a child node and a function to traverse through the B-Tree nodes to retrieve the keys falling within a certain range

- Code Snippet for search based on equality condition:

```
template <typename T>
std::pair<BTreeNode<T>*, T> BTreeNode<T>::search_eq(const T &k, std::function<bool(const T&,
const T&)> comp)
{
    int i = 0;
    while (i < n && comp(keys[i], k)) i++;

    if (i < n && !comp(k, keys[i]) && !comp(keys[i], k))
        return {this, keys[i]};

    if (leaf)
        return {nullptr, T()};

    return C[i]->search_eq(k, comp);
}
```

- Code Snippet for traversal through the B-Tree nodes to retrieve the keys falling within a certain range:

```
template <typename T>
void BTreeNode<T>::range_traverse(const T &ge_val, const T &le_val, std::function<bool(const
T&, const T&)> comp)
{
    int i = 0;

    while (i < n && comp(keys[i], ge_val))
    {
        i++;
    }

    while (i < n && !comp(le_val, keys[i]))
    {
        if (!leaf)
        {
            C[i]->range_traverse(ge_val, le_val, comp);
        }
        // std::cout << " " << keys[i];
        boost::pfr::for_each_field(*(keys[i]), [](const auto& field) {
            std::cout << field << "; ";
        });
        std::cout << "\n";
        ++i;
    }

    if (!leaf)
    {
        C[i]->range_traverse(ge_val, le_val, comp);
    }
}
```

  - **BTree Class:** Represents the B-Tree
    - It contains the following fields: BTreeNode pointing to root and the degree of the B-Tree (3 is the degree used in the queries above)
    - It includes a constructor, destructor, function to traverse the tree, insert into the tree, and a function to perform a range traversal

- **Parallel Merge Sort:**

  - A Parallel merge sort was implemented and is in turn used by many parallel queries that require sorting
  - It switches to a sequential sort when the block size is below 100
  - It includes a parallel merge that uses the median of the larger sorted sub-block to perform binary search on the smaller sorted sub-block and recursively merges the resultant splits accordingly
  - It switches to a sequential merge when the size of the 2 sub-blocks under consideration is below 1000 cumulatively

3. Functional Testing

- Functional Testing was performed to verify the behaviour of the different functions implemented and ensure that the sequential and parallel versions of a particular function are equivalent in terms of their correctness criterion
- A Person Relation was created consisting of the fields: *id, first name, last name, age, country, salary*
- 100 rows were inserted into the above relation
- The sequential and parallel versions (including version 1 and 2 for the group by aggregate queries) of different functions were invoked on the above relation
- The generated outputs for the sequential and parallel versions were compared after sorting
- It was verified that the outputs for the sequential and parallel versions are equivalent, hence denoting their correctness

4. Performance Comparison

- Performance Testing was done to compare the performance of the sequential and parallel versions of the different implemented functions
- The performance results were generated on the *crunchy5.cims.nyu.edu* machine with the following configuration:
  - Sockets: 4
  - Cores per socket: 8
  - Threads per core: 2
  - CPUs: 64 (4*8*2)
  - Architecture: x86_64

- A Person Relation was created consisting of the fields: *id, first name, last name, age, country, salary*
- 10000 rows were inserted into the above relation
- The sequential and parallel versions (including version 1 and 2 for the group by aggregate queries) of different functions were invoked on the above relation
- Instrumentation code was added to record the start and end timestamps of the functions
- The parallel versions were executed for varying number of threads, namely, 1,2,4,8,16,32 threads
- The elapsed duration was computed and used to generate speedup graphs

Challenges Encountered

- The key challenge that was encountered was related to providing a working interface to the different CRUD operations which accurately generates the required code while also accounting for the different cases. In this regard, various solutions and workarounds were implemented in order to effectively handle this such as using macro programming and pre-processing directives provided by the C++ Boost library.
- It was attempted to implement the range query (>=x and <=y) on a primary key using 2 split operations on the B-Tree such that the first split generates a new tree with values >= x. This new tree is split again to generate values <=y. It was possible to get a working and correct implementation in this direction. However, while executing for larger number of records, it triggered a few edge cases that could not be resolved in time. Therefore, this implementation has not been considered for the performance comparison (the implemented functions are still retained in the submitted source code: range_query, split_ge and split_le functions, but they are not used). An alternate range traversal function was implemented and used for the performance comparison.

Source Code

- The Source Code for this project can be found on the following GitHub Repository:
https://github.com/darshand15/PPA_Project

- The implementation of the parallel relational database is provided as a header file that can just be included in the client to avail the different functionalities. This header file, named *"header_par_db.h"* is placed inside the *./include* directory
- The *./include* directory also contains *parlaylib*, which is used to avail the parallel fork-join interface in C++

- **Functional Testing**
  - Functional Testing can be performed by running the script, *./run_script.sh* within the *./functional_testing* directory
  - If the *diff* output from the script shows that the generated sequential and parallel outputs were identical, it denotes that the Functional Testing was successful

    - The recorded outputs are contained in the generated *./functional_testing/output* directory
    - It has to be noted that Functional Testing uses a slightly different version of the *"header_par_db.h"*, named *"header_par_db_ft.h"* contained in the *./functional_testing* directory. This is because the parallel sort in *"header_par_db_ft.h"* switches to a sequential merge and sequential sort for smaller block sizes as compared to the header in *./include*, which is optimized for performance. This ensures that parallel behaviour is invoked and verified despite the functional tests having fewer records.

- **Performance Testing**
    - Performance Testing can be performed by running the script, *./run_script.sh* within the *./performance_testing* directory
    - The recorded timing measurements are contained in the generated *./performance_testing/timing_measurements* directory

- *generate_insert.py* inside *./generate_data* directory can be used to generate input data
- The *./generate_graphs* directory contains the code to generate graphs
- The generated graphs are contained in the *./generate_graphs/graphs* directory