PROCESSING BANK TRANSACTION RECORDS

62130500206 KAEWKET SAELEE

62130500226 WISARUT KITTICHAROENPHONNGAM

62130500254 KAVISARA SRISUWATCHAREE

THIS REPORT IS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR

CSC371 INTRODUCTION TO DISTRIBUTED SYSTEMS
AND PARALLEL COMPUTING

SCHOOL OF INFORMATION TECHNOLOGY

KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI

SEMESTER 1/2021

General Overview of platform

Operating System: macOS Monterey 12.0
Chip: Apple M1
Ram: 16 GB
Total number of Cores: 8 Cores

We get information about the operating system, chip, ram, and the total number of cores from the system report service in macOS.

Number of Cores in use: 4 Cores

We configure the number of cores in use using ForkJoinPool from library java.util.concurrent.ForkJoinPool

OpenJDK Version: 15.0.2

We retrieve the version of OpenJDK by running `java --version`

IDE: Visual Studio Code (v1.61.2)

Discussion on our strategy to time the tasks

We use System.currentTimeMillis() to get the current time in the millisecond and we convert it into a timestamp.

We use endTimestamp - startTimestamp to get the total time that is taken when performing the task as you can see in Figure 1.

```
Timestamp timestamp = new Timestamp(System.currentTimeMillis());
// Perform Task
Timestamp endTimestamp = new Timestamp(System.currentTimeMillis());
double time = (endTimestamp.getTime() - timestamp.getTime());
System.out.println("Time taken: " + time + " ms");
```

Figure 1 shows the implementation of timing the tasks.

Discussion of Manipulating the given data

We receive a CSV file that contains 5 columns which are Date, Description, Deposits, Withdrawals, Balance that are separated by a comma. There is also a comma in Deposits, Withdrawals, and Balance so we need to handle it properly.

We use Files.lines(Paths.get(filename)) to read the file as a stream and skip the first row which is the header row (show the column name) and we map it using our implemented CSVParser::Parse method that we create for parse the CSV that contains a comma in the column. Then we map it again to convert splitting data into Transaction Class as you can see in Figure 2.

```
Stream<String> files = Files.lines(Paths.get(filename));
files.skip(1)
     .map(CSVParser::parse)
     .map(line ->
        new Transaction(
            line[0].split("-"),
            line[1],
            line[2],
            line[3],
            line[4]))
```

Figure 2 shows the implementation of reading the file and parsing it to a list of Transaction.

```
public class CSVParser {
  public static String[] parse(String content) {
    String[] data = content.split(",(?=([^\"]*\"[^\"]*\")*[^\"]*$)");
    return data;
  }
}
```

Figure 3 shows CSVParser class and parse method.

In Transaction Class (Figure 4), we use Calendar (java.util.Calendar) to store the date, and since the month from CSV is MMM format so we need to convert it into the number in the range 0 - 11. We created a Helper class to handle it as you can see in Figure 5.

```java
import java.util.Calendar;
public class Transaction {
  private Calendar date;
  private String description;
  private String deposit;
  private String withdraw;
  private String balance;
  public Transaction(
    String[] date,
    String description,
    String deposit,
    String withdraw,
    String balance
  ) {
    this.date = Calendar.getInstance();
    this.date.set(Calendar.YEAR, Integer.parseInt(date[2]));
    this.date.set(Calendar.MONTH, Helper.monthToInt(date[1]));
    this.date.set(Calendar.DAY_OF_MONTH, Integer.parseInt(date[0]));
    this.date.set(Calendar.HOUR_OF_DAY, 0);
    this.date.set(Calendar.MINUTE, 0);
    this.date.set(Calendar.SECOND, 0);
    this.date.set(Calendar.MILLISECOND, 0);
    this.description = description;
    this.deposit = deposit.replace("\"", "");
    this.withdraw = withdraw.replace("\"", "");
    this.balance = balance.replace("\"", "");
 }
```

Figure 4 shows Transaction Class with its constructor.

```java
public class Helper {
  public static int monthToInt(String month) {
    switch (month) {
      case "Jan":return 0;
      case "Feb":return 1;
      case "Mar":return 2;
      case "Apr":return 3;
      case "May":return 4;
      case "Jun":return 5;
      case "Jul":return 6;
      case "Aug":return 7;
      case "Sep":return 8;
      case "Oct":return 9;
      case "Nov":return 10;
      case "Dec":return 11;
      default:return -1;
    }
  }
}
```

Figure 5 shows Helper Class with monthToInt method.

Discussion of Task 1

We need to show the first instances of each transaction's description category. So after we show the transaction record, we use a filter with Transaction::isClearBalance in order to check the balance is equal to 0. We collect it using groupingBy(Transaction:: getDescription) which will group the transaction by description. Then, we convert it as a stream using .entrySet().stream(), and then we map it with entry.getValue() to get only values of the map (ignore the key). After that, we sort the data by date and description in case that it has the same transaction date as you can see in figure ??. Then we collect it as a list and print the first instances of each transaction's description category as you can see in Figure 6.

```
.filter(Transaction::isClearBalance)
        .collect(
          groupingBy(
            Transaction::getDescription,
            collectingAndThen(toList(), t -> t.stream().findFirst().get())
          )
        )
        .entrySet()
        .stream()
        .map(entry -> entry.getValue())
        .sorted(
          (t1, t2) -> {
            int dateCompare = t1.getDate().compareTo(t2.getDate());
            if (dateCompare == 0) {
              return t1.getDescription().compareTo(t2.getDescription());
            }
            return dateCompare;
          }
        )
```

Figure 6 shows the implementation of Task 1.

```
public class Transaction {
  public boolean isClearBalance() {
    return balance.equals("00.00");
  }

  public String getDescription() {
    return description;
  }

  public Calendar getDate() {
    return date;
  }
}
```

Figure 7 shows methods for Task 1 in the Transaction class.

Task 1's result on 5000 BT Records.csv

```
==================== QUESTION 1 (5000 BT Records.csv) ====================
================================ SEQUENCE ================================
20/8/2020 Transfer 00.00 203,392.65 00.00
22/8/2020 ATM 00.00 529,646.03 00.00
22/8/2020 Miscellaneous 00.00 45,214.40 00.00
22/8/2020 RTGS 00.00 202,014.62 00.00
22/8/2020 Reversal 00.00 443,730.89 00.00
24/8/2020 Interest 00.00 2,185,930.03 00.00
25/8/2020 Bill 00.00 95,002.58 00.00
26/8/2020 Commission 00.00 25,903.94 00.00
31/8/2020 Purchase 00.00 882,493.33 00.00
31/8/2020 Tax 00.00 1,634,931.86 00.00
1/9/2020 NEFT 00.00 117,202.15 00.00
3/9/2020 Debit Card 00.00 225,249.13 00.00
5/9/2020 IMPS 00.00 833,070.75 00.00
16/9/2020 Cash 00.00 2,458,253.36 00.00
5/10/2020 Cheque 00.00 1,437,028.05 00.00
Time taken: 285.0 ms
================================ PARALLEL ================================
20/8/2020 Transfer 00.00 203,392.65 00.00
22/8/2020 ATM 00.00 529,646.03 00.00
22/8/2020 Miscellaneous 00.00 45,214.40 00.00
22/8/2020 RTGS 00.00 202,014.62 00.00
22/8/2020 Reversal 00.00 443,730.89 00.00
24/8/2020 Interest 00.00 2,185,930.03 00.00
25/8/2020 Bill 00.00 95,002.58 00.00
26/8/2020 Commission 00.00 25,903.94 00.00
31/8/2020 Purchase 00.00 882,493.33 00.00
31/8/2020 Tax 00.00 1,634,931.86 00.00
1/9/2020 NEFT 00.00 117,202.15 00.00
3/9/2020 Debit Card 00.00 225,249.13 00.00
5/9/2020 IMPS 00.00 833,070.75 00.00
16/9/2020 Cash 00.00 2,458,253.36 00.00
5/10/2020 Cheque 00.00 1,437,028.05 00.00
Time taken: 113.0 ms
Sequence: 285.0 ms
Parallel: 113.0 ms
Speed Up: 2.52212389380531 times faster
Efficiency: 63.05309734513275 %
==========================================================================
```

Figure 8 shows the result of task 1 on 5000 BT Records.csv.

Sequential Stream Processing takes 285.0 ms
Parallel Stream Processing takes 113.0 ms
It speeds up 2.52 times faster which is 63.05 % Efficiency (core = 4)

Task 1's result on 5000000 BT Records.csv

```
==================== QUESTION 1 (5000000 BT Records.csv) ==================
================================ SEQUENCE ================================
21/8/2020 Cheque 00.00 1,046,983.40 00.00
21/8/2020 Commission 00.00 106,538.38 00.00
21/8/2020 Debit Card 00.00 1,294,787.92 00.00
21/8/2020 NEFT 00.00 2,028,759.54 00.00
21/8/2020 Reversal 00.00 476,630.35 00.00
21/8/2020 Tax 00.00 23,332.54 00.00
22/8/2020 ATM 00.00 463.11 00.00
22/8/2020 IMPS 00.00 30,785.36 00.00
22/8/2020 RTGS 00.00 161,386.03 00.00
25/8/2020 Bill 00.00 826,992.00 00.00
25/8/2020 Miscellaneous 00.00 614,076.46 00.00
25/8/2020 Purchase 00.00 1,808,969.00 00.00
25/8/2020 Transfer 00.00 14,398.08 00.00
27/8/2020 Interest 00.00 396,681.97 00.00
6/9/2020 Cash 00.00 81,018.22 00.00
Time taken: 20910.0 ms
================================ PARALLEL ================================
21/8/2020 Cheque 00.00 1,046,983.40 00.00
21/8/2020 Commission 00.00 106,538.38 00.00
21/8/2020 Debit Card 00.00 1,294,787.92 00.00
21/8/2020 NEFT 00.00 2,028,759.54 00.00
21/8/2020 Reversal 00.00 476,630.35 00.00
21/8/2020 Tax 00.00 23,332.54 00.00
22/8/2020 ATM 00.00 463.11 00.00
22/8/2020 IMPS 00.00 30,785.36 00.00
22/8/2020 RTGS 00.00 161,386.03 00.00
25/8/2020 Bill 00.00 826,992.00 00.00
25/8/2020 Miscellaneous 00.00 614,076.46 00.00
25/8/2020 Purchase 00.00 1,808,969.00 00.00
25/8/2020 Transfer 00.00 14,398.08 00.00
27/8/2020 Interest 00.00 396,681.97 00.00
6/9/2020 Cash 00.00 81,018.22 00.00
Time taken: 6333.0 ms
Sequence: 20910.0 ms
Parallel: 6333.0 ms
Speed Up: 3.30175272382757 times faster
Efficiency: 82.54381809568925 %
==========================================================================
```

Figure 9 shows the result of task 1 on 5000000 BT Records.csv.

Sequential Stream Processing takes 20910.0 ms
Parallel Stream Processing takes 6333.0 ms
It speeds up 3.30 times faster which is 82.54 % Efficiency (core = 4)

Discussion of Task 2 in general

We need to show summary details in each month. So after we finish manipulating the data.

We collect it using groupingBy(Transaction::getMonthYear)) which will group the transaction by month and year. And we convert it as a stream using .entrySet().stream() and then we map to run the particular task depending on the requirement which will return MonthTransactionClass. After that, we sort the data by month and year and collect it into the list and print the details of each particular task.

```
filesAfterManipulate
.collect(groupingBy(Transaction::getMonthYear))
.entrySet()
.stream()
.map( PERFORM_SPECIFIC_TASK )
.sorted((t1, t2) -> t1.getMonthYear().compareTo(t2.getMonthYear()))
.collect(toList())
.forEach( PRINT_SPECIFIC_DETAIL );
```

Figure 10 shows the brief implementation of Task 2.

```
public class Transaction {
    public String getMonthYear() {
        return ((date.get(Calendar.MONTH) + 1) + "/" + date.get(Calendar.YEAR));
    }
}
```

Figure 11 shows the getMonthYear() method in Transaction class.

```
public class MonthTransaction {
  private Calendar monthYear;
  private double deposit;
  private double withdraw;
  private double balance;
  public Calendar getMonthYear() {
    return monthYear;
  }
}
```

Figure 12 shows MonthTransaction class with getMonthYear() method.

Discussion of Task 2.1

We compute the monthly deposits and monthly withdrawals by making the entry transaction as stream and map to get deposits or withdraw and use reduce to sum it. We use BigDecimal Class for computing decimal numbers to prevent errors that an error when calculating the floating-point number as you can see in Figure 13.

```
entry -> {
    double totalDeposit = entry
                    .getValue()
                    .stream()
                    .map(d -> d.getDeposit())
                    .reduce(BigDecimal.ZERO, BigDecimal::add)
                    .setScale(2, RoundingMode.HALF_UP)
                    .doubleValue();

    double totalWithdraw = entry
                    .getValue()
                    .stream()
                    .map(d -> d.getWithdraw())
                    .reduce(BigDecimal.ZERO, BigDecimal::add)
                    .setScale(2, RoundingMode.HALF_UP)
                    .doubleValue();

    return new MonthTransaction(
                    entry.getKey(),
                    totalDeposit,
                    totalWithdraw,
                    0
                  );
}
```

Figure 13 shows the implementation of Task 2.1.

For this task, we will show the deposit and withdraw in each month using toStringWithDetail in MonthTransaction Class.

```
.forEach(v -> System.out.println(v.toStringWithDetail()));

public class MonthTransaction {
    public String toStringWithDetail() {
        return (
          (monthYear.get(Calendar.MONTH) + 1) +
          "/" +
          monthYear.get(Calendar.YEAR) +
          ": " +
          " Deposit: " +
          String.format("%.2f", deposit) +
          " Withdraw: " +
          String.format("%.2f", withdraw)
        );
    }
}
```

Figure 14 shows the print method of Task2.1 and its detail.

Task 1's result on 5000 BT Records.csv

```
================================ TASK 1 ================================
==================== QUESTION 2 (5000 BT Records.csv) ====================
================================ SEQUENCE ================================
8/2020:  Deposit: 34776892.97 Withdraw: 34664601.66
9/2020:  Deposit: 88195424.71 Withdraw: 88379796.55
10/2020:  Deposit: 58346285.49 Withdraw: 56249329.16
11/2020:  Deposit: 70017776.84 Withdraw: 71145538.59
12/2020:  Deposit: 59403190.49 Withdraw: 59900428.75
1/2021:  Deposit: 80935484.27 Withdraw: 80687706.96
2/2021:  Deposit: 42083017.52 Withdraw: 42531607.04
Time taken: 424.0 ms
================================ PARALLEL ================================
8/2020:  Deposit: 34776892.97 Withdraw: 34664601.66
9/2020:  Deposit: 88195424.71 Withdraw: 88379796.55
10/2020:  Deposit: 58346285.49 Withdraw: 56249329.16
11/2020:  Deposit: 70017776.84 Withdraw: 71145538.59
12/2020:  Deposit: 59403190.49 Withdraw: 59900428.75
1/2021:  Deposit: 80935484.27 Withdraw: 80687706.96
2/2021:  Deposit: 42083017.52 Withdraw: 42531607.04
Time taken: 218.0 ms
Sequence: 424.0 ms
Parallel: 218.0 ms
Speed Up: 1.944954128440367 times faster
Efficiency: 48.62385321100918 %
==========================================================================
```

Figure 15 shows the result of task 2.1 on 5000 BT Records.csv.

Sequential Stream Processing takes 424.0 ms
Parallel Stream Processing takes 218.0 ms
It speeds up 1.94 times faster which is 48.62 % Efficiency (core = 4)

Task's 1 result on 5000000 BT Records.csv

```
================================== TASK 1 ==================================
==================== QUESTION 2 (5000000 BT Records.csv) ===================
================================= SEQUENCE =================================
8/2020:  Deposit: 108810108.64 Withdraw: 108165645.28
9/2020:  Deposit: 247968720.02 Withdraw: 248607891.43
10/2020:  Deposit: 214481950.63 Withdraw: 213114321.69
11/2020:  Deposit: 283774415.91 Withdraw: 285207959.27
12/2020:  Deposit: 257631102.73 Withdraw: 256496178.05
1/2021:  Deposit: 265999378.94 Withdraw: 266974115.06
2/2021:  Deposit: 244908763.69 Withdraw: 244946268.40
3/2021:  Deposit: 303545632.13 Withdraw: 303346767.23
.
.
.
4/2155:  Deposit: 259427006.49 Withdraw: 260431700.45
5/2155:  Deposit: 249672562.05 Withdraw: 248481809.22
6/2155:  Deposit: 262500922.76 Withdraw: 262266612.46
7/2155:  Deposit: 251133889.15 Withdraw: 252424215.24
8/2155:  Deposit: 294445949.48 Withdraw: 294476483.70
9/2155:  Deposit: 302395312.83 Withdraw: 302491866.89
10/2155:  Deposit: 257978888.70 Withdraw: 256915042.94
11/2155:  Deposit: 175948074.92 Withdraw: 176893822.58
Time taken: 14687.0 ms
Sequence: 34648.0 ms
Parallel: 14687.0 ms
Speed Up: 2.3590930755089534 times faster
Efficiency: 58.977326887723834 %
============================================================================
```

Figure 16 shows the result of task 2.1 on 5000000 BT Records.csv.

Sequential Stream Processing takes 34648.0 ms
Parallel Stream Processing takes 14687.0 ms
It speeds up 2.36 times faster which is 58.98 % Efficiency (core = 4)

Discussion of Task 2.2

We want to compute the monthly balance using entry balance - total transaction in the month. We calculate the entry balance by calling the function getEntryBalance() that will use the actual balance add the withdrawal amount and subtract the deposit amount to get the entry balance before it passes this transaction. And we calculate the total transaction in the month by sum of deposits amount subtract by withdrawal amount and we add it to the entry balance to get the monthly balance as you can see in Figure 17.

```
entry -> {
            Transaction firstTransaction = entry.getValue().get(0);
            BigDecimal initialBalance = firstTransaction.getEntryBalance();
            BigDecimal totalTransaction = entry
              .getValue()
              .stream()
              .map(d -> d.getTransactionAmount())
              .reduce(BigDecimal.ZERO, BigDecimal::add);
            double monthlyBalance = initialBalance
              .add(totalTransaction)
              .setScale(2, RoundingMode.HALF_UP)
              .doubleValue();
            return new MonthTransaction(entry.getKey(), 0, 0, monthlyBalance);
```

Figure 17 shows the implementation of Task 2.2.

```
public class Transaction {
  public BigDecimal getBalance() {
    return new BigDecimal(balance.replace(",", ""));
  }

  public BigDecimal getDeposit() {
    return new BigDecimal(deposit.replace(",", ""));
  }

  public BigDecimal getWithdraw() {
    return new BigDecimal(withdraw.replace(",", ""));
  }

  public BigDecimal getTransactionAmount() {
    return getDeposit().subtract(getWithdraw());
  }

  public BigDecimal getEntryBalance() {
    return getBalance().subtract(getTransactionAmount());
  }
}
```

Figure 18 shows methods for Task 2.2 in the Transaction class.

```
.forEach(v -> System.out.println(v.toStringWithDetail()));

public class MonthTransaction {
  public String toString() {
    return (
      (monthYear.get(Calendar.MONTH) + 1) +
      "/" +
      monthYear.get(Calendar.YEAR) +
      ": " +
      String.format("%.2f", balance)
    );
  }
}
```

Figure 19 shows the print method of Task2.2 and its detail.

Discussion of a transforming sequential to parallel stream processing

We have about 3 points to change from sequential into parallel stream processing.

First of all, when reading the file we add .parallel() into our stream process to make it parallel stream processing.

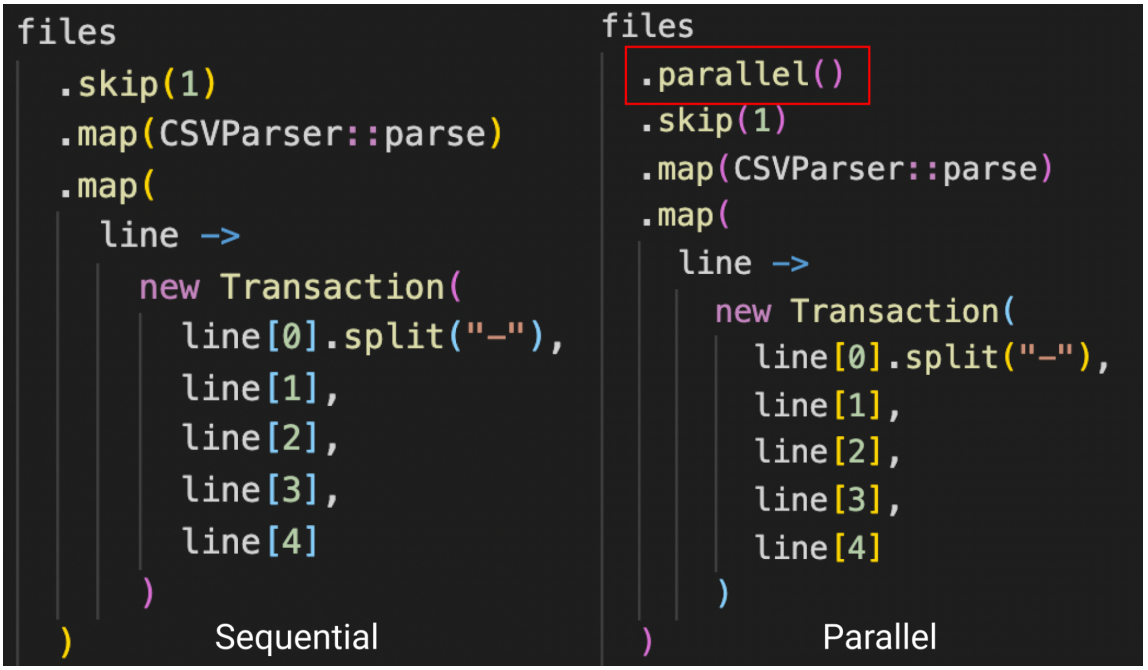

Figure 20 shows the first difference between sequential and parallel.

Second, we change from .stream() to be .parallelStream() to make it parallel stream processing.

Sequential

```
t -> t.stream().findFirst().get()
```

Parallel

```
t -> t.parallelStream().findFirst().get()
```

Figure 21 shows the second difference between sequential and parallel.

Last, we want to configure the number of cores that we want to use so we need to modify some code by adding ForkJoinPool and submitting our stream processing into ForkJoinPool before showing the result.

Sequential

```
files
  .skip(1)
  .map(CSVParser::parse)
  .map(
    line ->
      new Transaction(
        line[0].split("-"),
        line[1],
        line[2],
        line[3],
        line[4]
      )
  )
  .filter(Transaction::isClearBalance)
  .collect(
    groupingBy(
      Transaction::getDescription,
      collectingAndThen(toList(), t -> t.stream().findFirst().get())
    )
  )
  .entrySet()
  .stream()
  .map(entry -> entry.getValue())
  .sorted(
    (t1, t2) -> {
      int dateCompare = t1.getDate().compareTo(t2.getDate());
      if (dateCompare == 0) {
        return t1.getDescription().compareTo(t2.getDescription());
      }
      return dateCompare;
    }
  )
  .collect(toList())
  .forEach(System.out::println);
```

Parallel

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
forkJoinPool
  .submit(
    () ->
      files
        .parallel()
        .skip(1)
        .map(CSVParser::parse)
        .map(
          line ->
            new Transaction(
              line[0].split("-"),
              line[1],
              line[2],
              line[3],
              line[4]
            )
        )
        .filter(Transaction::isClearBalance)
        .collect(
          groupingBy(
            Transaction::getDescription,
            collectingAndThen(toList(), t -> t.parallelStream().findFirst().get())
          )
        )
        .entrySet()
        .parallelStream()
        .map(entry -> entry.getValue())
        .sorted(
          (t1, t2) -> {
            int dateCompare = t1.getDate().compareTo(t2.getDate());
            if (dateCompare == 0) {
              return t1.getDescription().compareTo(t2.getDescription());
            }
            return dateCompare;
          }
        )
        .collect(toList())
  )
  .get()
  .forEach(System.out::println);
```

Figure 22 shows the third difference between sequential and parallel.

Task 2's result on 5000 BT Records.csv

```
================================== TASK 2 ==================================
==================== QUESTION 2 (5000 BT Records.csv) ====================
================================ SEQUENCE ================================
8/2020: 185078.48
9/2020: 706.77
10/2020: 2097663.36
11/2020: 969901.78
12/2020: 472663.74
1/2021: 720441.23
2/2021: 271851.86
Time taken: 430.0 ms
================================ PARALLEL ================================
8/2020: 185078.48
9/2020: 706.77
10/2020: 2097663.36
11/2020: 969901.78
12/2020: 472663.74
1/2021: 720441.23
2/2021: 271851.86
Time taken: 158.0 ms
Sequence: 430.0 ms
Parallel: 158.0 ms
Speed Up: 2.721518987341772 times faster
Efficiency: 68.0379746835443 %
==========================================================================
```

Figure 23 shows the result of task 2.2 on 5000 BT Records.csv.
Sequential Stream Processing takes 430.0 ms
Parallel Stream Processing takes 158.0 ms
It speeds up 2.72 times faster which is 68.04 % Efficiency (core = 4)

Task 2's result on 5000000 BT Records.csv

```
================================== TASK 2 ==================================
==================== QUESTION 2 (5000000 BT Records.csv) ===================
================================== SEQUENCE ================================
8/2020: 708139.19
9/2020: 68968.16
10/2020: 1436597.62
11/2020: 3054.82
12/2020: 1137980.45
1/2021: 163245.07
2/2021: 125741.08
3/2021: 324606.89
.
.
.
1/2155: 515092.28
2/2155: 1356.20
3/2155: 1004692.38
4/2155: -0.72
5/2155: 1190752.83
6/2155: 1425063.91
7/2155: 134738.63
8/2155: 104205.14
9/2155: 7652.06
10/2155: 1071498.68
11/2155: 125751.78
Time taken: 14329.0 ms
Sequence: 33162.0 ms
Parallel: 14329.0 ms
Speed Up: 2.314327587410147 times faster
Efficiency: 57.85818968525368 %
============================================================================
```

Figure 24 shows the result of task 2.2 on 5000000 BT Records.csv.
Sequential Stream Processing takes 33162.0 ms
Parallel Stream Processing takes 14329.0 ms
It speeds up 2.31 times faster which is 57.86 % Efficiency (core = 4)

Conclusion

From the experiment, we found that on both 5000 BT records and 5000000 BT records the parallel stream takes less time for processing the data than the sequential stream. Since the size of the datasets is 1000 times so we cannot measure the performance of parallel stream processing compared to sequential stream processing properly.