Bank Transactions

Runtime platform overview

The program is run on the virtual machine on Microsoft Azure. We decided to use VM on Azure because we can control how many vCPUs are available to Java runtime. The VM has the following specification.

- VM Sizes: Standard F4s_v2
- vCPUs: 4
- Memory: 8GB
- OS: Windows 10 Pro
- Java version: 17.0.1
- IDE: IntelliJ IDEA

Task Running

To run the task, we create the following helper class and functions and to avoid duplicate codes.

Transaction

Transaction class is created to represent the data on each row of transactions in a CSV file. It contains the following attributes and some other getter, setter, and printing methods.

- date
- description
- deposit
- withdrawal
- balance

Task

This is a functional interface for our task which is written in a different method. We can easily use this interface for running the task in `runTask` function.

readFromCSV

We created this function to receive the filename and read the CSV file. We used an external library calle `OpenCSV` for parsing each row in CSV into the array of strings. Then the data is used to create a Transaction object and added to ArrayList of Transaction. After that, the ArrayList of Transaction object is returned.

runTask

This function is used to run the task by passing ArrayList of Transaction, boolean whether to print the result and the task as function interface. It is responsible for timing the task using `LocalDateTime.now()` function and returning the duration between start time and finish time as `long` in a nanosecond.

printSpeedupAndEfficiency

This function is responsible for calculate the speedup and efficiency with given inputs of time in serial and time in parallel. The output is printed to the console.

roundToTwoDecimal

This function is used to round the value in `double` to two decimal places.

printMachineInfo

This function is used to print the available processor and memory of the running machine. The information is obtain from the `Runtime` class

Main

Main function is the entry point of the application. First, it called `readFromCSV` function to get ArrayList of Transaction. Then it used `runTask` function to run the specific task. After that, it will use `printSpeedupAndEfficiency` function to print speedup and efficiency to the console. These steps are repeated for different tasks and data.

Task 1

First, we transform the input data source into stream by using `.stream()` and `parallelStream()`. Then, we used `.filter((Transaction t) -> t.getBalance() == 0)` to filter only transaction that made account balance equal to 0. After that, the results are collected with `.collect()` and grouped by their description. At this point, we have Map of String, the description, and List of Transaction objects. Therefore, we transform the values of Map (List of Transaction objects) into the stream again. Then, we used `.map()` to get only the first Transaction from each description. Since the ArrayList has encountered order, the results that we get is the first transaction of each description that made the balance equal to 0. In the end, we used `.forEach()` to print out the result to the console.

Result

> Task 1: Serial 15.62 ms, Parallel 15.63 ms
> Speedup: 1.0 Efficiency: 0.25
>
> Large Task 1: Serial 124.99 ms, Parallel 62.51 ms
> Speedup: 2.0 Efficiency: 0.5

Task 2

Task 2.1

First, we transform the input data source into stream by using `.stream()` and `.parallelStream()`. Then, we used `.collect(Collectors.groupingBy(Transaction::getMonthYear))` to group the transactions by month and year (transactions in the group will have the same month and year). After that, we use `.forEach()` to iterate all group of transactions. After that, we calculate the balance of transaction in the

group by tranform ArrayList of Transaction to be stream by using `.stream()` and `parallelStream()`. Then use `.map()` to get the balance in that transation (deposit - withdrawl). Then we reduce it to be the balance by using `.reduce(0f, Float::sum)`. In the end, we print out the balance in that month to the console.

Result of Task 2.1

> Task 2.1: Serial 31.25 ms, Parallel 15.62 ms
> Speedup: 2.0 Efficiency: 0.5
>
> Large Task 2.1: Serial 2453.11 ms, Parallel 1093.76 ms
> Speedup: 2.24 Efficiency: 0.56

Task 2.2

This task is similar to task2.1, but in order to calculate the balance we need to use the balance from the given CSV file. That means we need to extract the first balance of that month. After we create the stream and group by month and year and iterate all group by using `.forEach()`, we use `.get(0)` to retreive the first transaction. After that the initial balance will be balance of first transaction + withdrawl of first transaction - deposit of first transaction (since balance in this row is balance after including withdrawl and deposit). After that we use `.map()` and `.reduce()` to calculate the balance in that month. In the end, we print out the balance in that month to the console.

Result of Task 2.2

> Task 2.2: Serial 15.62 ms, Parallel 15.62 ms
> Speedup: 1.0 Efficiency: 0.25
>
> Large Task 2.2: Serial 2000.05 ms, Parallel 1078.12 ms
> Speedup: 1.86 Efficiency: 0.47

Conclusion

From the result of running tasks 1 and 2, we can clearly see the difference in speedup and efficiency between the two sizes of the dataset for using parallel stream. The speedup and efficiency increase when we change data from 5,000 records to 5,000,000 records. It follows the NQ model such that the larger the product of N (number of data) * Q (Amount of operation per element), the more likely that we will benefit from parallel computing. Therefore, increasing out N or number of data from 5,000 to 5,000,000 made the problem large enough to overcome the overhead cost of parallel computing.

Team Members

- 62130500209 Thanakorn Aungunchuchod
- 62130500212 Thanaphon Sombunkaeo
- 62130500230 Sethanant Pipatpakorn