Group 1

Assignment 5

Week 10

10.1.1*) Compile and run PrimeCountingPerf.java. Record the result in a text file.*

Sequential 5477216.1 ns 354045.52 64

IntStream 1.2 ns 0.05 268435456

Parallel 2.9 ns 0.17 134217728

ParallelStream 3.0 ns 0.23 134217728

10.1.2*) Fill in the Java code using a stream for counting the number of primes (in the range: 2..range). Record the result in a text file.*

9592

10.1.3*) Add code to the stream expression that prints all the primes in the range 2..range. To test this program reduce range to a small number e.g. 1000.*

see code.

10.1.4*) Fill in the Java code using the intermediate operation parallel for counting the number of primes (in the range: 2..range). Record the result in a text file.*

Sequential 5487057.0 ns 228964.71 64

IntStream 5594811.3 ns 280006.85 64

Parallel 1436794.3 ns 17953.56 256

10.1.5*) Add another prime counting method using a parallelStream for counting the number of primes (in the range: 2..range). Measure its performance using Mark7 in a way similar to how we measured the performance of the other three ways of counting primes.*

ParallelStream 1414756.3 ns 22077.23 256

Group 1

10.2.1*) Starting from the TestWordStream.java file, complete the readWords method and check that you can read the file as a stream and count the number of English words in it. For the english-words.txt file on the course homepage the result should be 235,886.*

see code

10.2.2*) Write a stream pipeline to print the first 100 words from the file.*

see code

10.2.3*) Write a stream pipeline to find and print all words that have at least 22 letters.*

see code

10.2.4*) Write a stream pipeline to find and print some word that has at least 22 letters.*

see code

10.2.5*) Write a method boolean isPalindrome(String s) that tests whether a word s is a palindrome: a word that is the same spelled forward and backward. Write a stream pipeline to find all palindromes and print them.*

see code

10.2.6*) Make a parallel version of the palindrome-printing stream pipeline. Is it possible to observe whether it is faster or slower than the sequential one?*

Yes, the parallel version is faster.

10.2.7*) Make a new version of the method readWordStream which can fetch the list of words from the internet. There is a (slightly modified) version of the word list at this URL: https://staunstrups.dk/jst/english-words.txt. Use this version of readWordStream to count the number of words (similarly to question 7.2.1). Note, the number of words is not the same in the two files !!*

see code

10.2.8*) Use a stream pipeline that turns the stream of words into a stream of their lengths, to find and print the minimal, maximal and average word lengths.*

see code

Min: 1

Man: 24

Avg: 9.569126612007494

10.3.1) *Redo the first example (running the code in Java8ParallelStreamMain) described in the article. Your solution should contain the output from doing this experiment and a short explanation of your output.*

Group 1

==========

Using Sequential Stream

==========

1 main

2 main

3 main

4 main

5 main

6 main

7 main

8 main

9 main

10 main

==========

Using Parallel Stream

==========

7 main

6 main

8 ForkJoinPool.commonPool-worker-2

1 ForkJoinPool.commonPool-worker-3

2 ForkJoinPool.commonPool-worker-2

10 ForkJoinPool.commonPool-worker-3

5 ForkJoinPool.commonPool-worker-2

3 ForkJoinPool.commonPool-worker-1

9 main

4 ForkJoinPool.commonPool-worker-4

We can observe that when using parallel streams, multiple threads are involved in the computation, leading to a improvement in speed.

Group 1

10.3.2) *Increase the size of the integer array (from the 10 in the article) and see if there is a relation between number of cores on your computer and the number of workes in the ForkJoin.*

Yes. As the size increases, the number of workers involved in the computation also increases. However, the number of workers is limited by the CPU count in the computer.

10.3.3) *Change the example by adding a time consuming task (e.g. counting primes in a limited range or the example in the artcle). Report what you see when running the example.*

It can be seen that using parallel streams results in a significantly faster speed compared to sequential streams.

10.4.1) *Describe what happens when this code runs:*

source().filter(w -> w.length() > 5).sink()

- *as a JavaStream (e.g. the source is a file)*
- *as a RxJava statement where the source could be an input field where a user types strings*

In Java Stream, it create a stream from a data source and filter out string elements with a length greater than 5. In RxJava, it create an observable object and similarly filter out strings with a length greater than 5.

10.4.2) . *Describe what happens when this code runs:*

source().filter(w -> w.length() > 5).sink();

source().filter(w -> w.length() > 10).sink()

- *as a JavaStream (e.g. the source is a file)*
- *as a RxJava statement where the source could be an input field where a user types strings*

In the Java Stream environment, this code attempts to create two streams from the same data source and apply different filtering conditions to them. However, Java Stream's design does not allow for repeated operations on the same source, which can lead to exceptions.

In RxJava environment, it is possible to create multiple independent data streams from the same event source and apply different filtering logic to them (many subscribers). The sink above will receive

strings with a length greater than 5, while the one below will receive strings with a length greater than 10.

Week 11

11.1.1) . *Design and implement the guardian actor (in Guardian.java) and complete the Main.java class to start the system. The Main class must send a kick-off message to the guardian. For now, when the guardian receives the kick-off message, it should spawn an MobileApp actor. Finally, explain the design of the guardian, e.g., state (if any), purpose of messages, etc. Also, briefly explain the purpose of the program statements added to the Main.java to start off the actor system.*

The guardian actor serves as top-level supervisor of the other actors in the system. It does not have any state. in the initialization of the guardian, Behavior is set up. it uses the Guardian command interface for messages, with specific message type(s) implementing that interface. in this case we use the KickOff type, that is used for starting the actor system, specifically spawning a MobileApp actor.

Statements added to Main.java:

initialization of the guardian actor	- sets up the foundation of the actor based application.
Kickoff Message	- for triggering the guardian actor (spawning the mobile app).
Termination	- For ensuring that all actors shuts down.

11.1.2). *Design and implement the Account actor (see the file Account.java in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state, how it is initialized, purpose of messages, etc.*

See Account.java

For the account actor, we added a balance property for its state. It includes a message type Deposit, for handling money deposits, adding the amount in the given deposit message to the value of balance in the account.

11.1.3). *Design and implement the Bank actor (see the file Bank.java in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state, how it is initialized, purpose of messages, etc.*

The Bank actor includes a Map<String, ActorRef<Account.AccountCommand>> to manage accounts using unique identifiers. It uses a message type NewAccount designed to create new accounts using name and Actor reference. The Transaction message type handles transactions, by specifying source, destination, and the amount to be transfered. The Bank actor maintains its state using the map of accounts, initialized within the constructor.

Group 1

11.1.4). *Design and implement the Mobile App actor (see the file MobileApp.java in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state, how it is initialized, purpose of messages, etc.*

The MobileApp actor uses a Map<String, ActorRef<Bank.BankCommand>> to help keep an overview of all registered banks in the system. The state is initialized within the constructor. The MobileApp includes message types NewBank, for adding new banks to the state, StartTransaction, for initiating single transactions.

11.1.5). *Update the guardian so that it starts 2 mobile apps, 2 banks, and 2 accounts. The guardian must be able to send a message to mobile app actors to execute a set of payments between 2 accounts in a specified bank. Finally, the guardian must send two payment messages: 1) from a1 to a2 via b1 , and 2) from a2 to a1 via b2 . The amount in these payments is a constant of your choice.*

See Guardian.java

11.1.6). *Modify the mobile app actor so that, when it receives a "make payments" message from the guardian, it sends 100 transactions between the specified accounts and bank with a random amount. Hint: You may use Random::ints to generate a stream of random integers. The figure below illustrates the computation.*

See MobileApp.java + Guardian.java.

11.1.7). *Update the Account actor so that it can handle a message PrintBalance. The message handler should print the current balance in the target account. Also, update the guardian actor so that it sends PrintBalance messages to accounts 1 and 2 after sending the make payments messages in the previous item. What balance will be printed? The one before all payments are made, or the one after all payments are made, or anything in between, or anything else? Explain your answer.*

The possibilities include before all payments , after all payments, and anything in between. Because No assumptions should be made about the order of arrival of messages. It is not guaranteed that payment message arrives before print().

11.1.8). *Consider a case where two different bank actors send two deposit exactly at the same time to the same account actor. Can data races occur when updating the balance? Explain your answer.*

No. In Akka, each actor has a mailbox. When two bank actors send deposit messages *at the same time* to an account actor, these messages are placed in the account actor's mailbox in the order they arrive and processed sequentially. While multiple actors can execute concurrently, the message processing within each actor is single-threaded, handling one message at a time. This ensures there is no data races.