**Programming Project: Best Matching Algorithm for Words**

Changes in Code:

package concurrent;  
  
  
import common.BestMatchingData;  
import distance.LevenshteinDistance;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class BestMatchingBasicConcurrentCalculation {  
  
 public static BestMatchingData getBestMatchingWords(String word, List<String> dictionary) throws InterruptedException, ExecutionException {  
  
 int numCores = Runtime.*getRuntime*().availableProcessors();  
// ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(numCores);  
 ExecutorService executor = Executors.*newFixedThreadPool*(numCores);  
  
 int size = dictionary.size();  
 int step = size / numCores;  
 int startIndex, endIndex;  
 List<Future<BestMatchingData>> results = new ArrayList<>();  
  
 for (int i = 0; i < numCores; i++) {  
 startIndex = i \* step;  
 if (i == numCores - 1) {  
 endIndex = dictionary.size();  
 } else {  
 endIndex = (i + 1) \* step;  
 }  
// BestMatchingBasicTask task = new BestMatchingBasicTask(startIndex, endIndex, dictionary, word);  
  
// Added Codes  
 int finalStartIndex = startIndex;  
 int finalEndIndex = endIndex;  
 Callable<BestMatchingData> task = () -> {  
 List<String> resultss=new ArrayList<String>();  
 int minDistance=Integer.*MAX\_VALUE*;  
 int distance;  
 for (int j = finalStartIndex; j< finalEndIndex; j++) {  
 distance= LevenshteinDistance.*calculate*(word,dictionary.get(j));  
 if (distance<minDistance) {  
 resultss.clear();  
 minDistance=distance;  
 resultss.add(dictionary.get(j));  
 } else if (distance==minDistance) {  
 resultss.add(dictionary.get(j));  
 }  
 }  
  
 BestMatchingData result=new BestMatchingData();  
 result.setWords(resultss);  
 result.setDistance(minDistance);  
 return result;  
 };  
  
 Future<BestMatchingData> future = executor.submit(task);  
 results.add(future);  
 }  
  
 executor.shutdown();  
  
 List<String> words = new ArrayList<String>();  
 int minDistance = Integer.*MAX\_VALUE*;  
 for (Future<BestMatchingData> future : results) {  
 BestMatchingData data = future.get();  
 if (data.getDistance() < minDistance) {  
 words.clear();  
 minDistance = data.getDistance();  
 words.addAll(data.getWords());  
 } else if (data.getDistance() == minDistance) {  
 words.addAll(data.getWords());  
 }  
  
 }  
  
 BestMatchingData result = new BestMatchingData();  
 result.setDistance(minDistance);  
 result.setWords(words);  
 return result;  
 }  
  
}

package concurrent;  
  
import common.BestMatchingData;  
import distance.LevenshteinDistance;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class BestMatchingAdvancedConcurrentCalculation {  
  
 public static BestMatchingData getBestMatchingWords(String word, List<String> dictionary) throws InterruptedException, ExecutionException {  
  
 int numCores = Runtime.*getRuntime*().availableProcessors();  
 ExecutorService executor = Executors.*newFixedThreadPool*(numCores);  
  
 int size = dictionary.size();  
 int step = size / numCores;  
 int startIndex, endIndex;  
 List<Future<BestMatchingData>> results;  
 List<Callable<BestMatchingData>> tasks = new ArrayList<>();  
  
 for (int i = 0; i < numCores; i++) {  
 startIndex = i \* step;  
 if (i == numCores - 1) {  
 endIndex = dictionary.size();  
 } else {  
 endIndex = (i + 1) \* step;  
 }  
// BestMatchingBasicTask task = new BestMatchingBasicTask(startIndex, endIndex, dictionary, word);  
  
 int finalStartIndex = startIndex;  
 int finalEndIndex = endIndex;  
 Callable<BestMatchingData> task = () -> {  
 List<String> resultss=new ArrayList<String>();  
 int minDistance=Integer.*MAX\_VALUE*;  
 int distance;  
 for (int j = finalStartIndex; j< finalEndIndex; j++) {  
 distance= LevenshteinDistance.*calculate*(word,dictionary.get(j));  
 if (distance<minDistance) {  
 resultss.clear();  
 minDistance=distance;  
 resultss.add(dictionary.get(j));  
 } else if (distance==minDistance) {  
 resultss.add(dictionary.get(j));  
 }  
 }  
  
 BestMatchingData result=new BestMatchingData();  
 result.setWords(resultss);  
 result.setDistance(minDistance);  
 return result;  
 };  
  
 tasks.add(task);  
 }  
  
 results = executor.invokeAll(tasks);  
 executor.shutdown();  
  
 List<String> words = new ArrayList<String>();  
 int minDistance = Integer.*MAX\_VALUE*;  
 for (Future<BestMatchingData> future : results) {  
 BestMatchingData data = future.get();  
 if (data.getDistance() < minDistance) {  
 words.clear();  
 minDistance = data.getDistance();  
 words.addAll(data.getWords());  
 } else if (data.getDistance() == minDistance) {  
 words.addAll(data.getWords());  
 }  
 }  
  
 BestMatchingData result = new BestMatchingData();  
 result.setDistance(minDistance);  
 result.setWords(words);  
 return result;  
  
 }  
  
}

package concurrent;  
  
import distance.LevenshteinDistance;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.NoSuchElementException;  
import java.util.concurrent.\*;  
  
public class ExistBasicConcurrentCalculation {  
  
 public static boolean existWord(String word, List<String> dictionary) throws InterruptedException, ExecutionException {  
 int numCores = Runtime.*getRuntime*().availableProcessors();  
// ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(numCores);  
 ExecutorService executor = Executors.*newFixedThreadPool*(numCores);  
  
 int size = dictionary.size();  
 int step = size / numCores;  
 int startIndex, endIndex;  
 List<Callable<Boolean>> tasks = new ArrayList<>();  
  
 for (int i = 0; i < numCores; i++) {  
 startIndex = i \* step;  
 if (i == numCores - 1) {  
 endIndex = dictionary.size();  
 } else {  
 endIndex = (i + 1) \* step;  
 }  
// ExistBasicTask task = new ExistBasicTask(startIndex, endIndex, dictionary, word);  
  
 int finalEndIndex = endIndex;  
 int finalStartIndex = startIndex;  
 Callable<Boolean> task = () -> {  
 for (int j = finalStartIndex; j< finalEndIndex; j++) {  
 if (LevenshteinDistance.*calculate*(word, dictionary.get(j))==0) {  
 return true;  
 }  
  
 if (Thread.*interrupted*()) {  
 return false;  
 }  
 }  
 throw new Exception("The word "+word+" doesn't exists.");  
 };  
  
 tasks.add(task);  
 }  
 try {  
 Boolean result = executor.invokeAny(tasks);  
 return result;  
 } catch (ExecutionException e) {  
 if (e.getCause() instanceof NoSuchElementException)  
 return false;  
 throw e;  
 } finally {  
 executor.shutdown();  
 }  
 }  
}

**Serial Version Execution Time:**

Dictionary Size: 250353

Word: stitter

Minimun distance: 1

List of best matching words: 9

sitter

skitter

slitter

spitter

stilter

stinter

stotter

stutter

titter

Execution Time: 444

Dictionary Size: 250353

Word: stitter

Exists: false

Execution Time: 466

**Parallel Version Execution Time:**

Run program on Intel core i7 with 8GB RAM

|  |  |
| --- | --- |
| No of Threads | Execution Time |
| 1 | 515 |
| 2 | 336 |
| 3 | 274 |
| 4 | 248 |

Dictionary Size: 250353

Word: stitter

Minimun distance: 1

List of best matching words: 9

sitter

skitter

slitter

spitter

stilter

stinter

stotter

stutter

titter

Execution Time: 291

Dictionary Size: 250353

Word: spitter

Exist: true

Execution Time: 123