**CS7370 Project Report**

**Joseph Hendrix**

**Wright State University**

**Department of Computer Science**

**Table of Contents**

[Algorithm Description 2](#_Toc520053427)

[Implementation Design Description 3](#_Toc520053428)

[Testing 5](#_Toc520053429)

[Results 7](#_Toc520053430)

[Discussion 9](#_Toc520053431)

[Implementation Source Code 10](#_Toc520053432)

[ImageFileSearch.java 10](#_Toc520053433)

[ParallelImageFileSearch.java 12](#_Toc520053434)

[SerialImageFileSearch.java 13](#_Toc520053435)

[Implementation Documentation 14](#_Toc520053436)

[ImageFileSearch 14](#_Toc520053437)

[ParallelImageFileSearch 14](#_Toc520053438)

[SerialImageFileSearch 14](#_Toc520053439)

[Appendix: CS 7370 Programming Project Outline 15](#_Toc520053440)

# Algorithm Description

The algorithm that has been implemented for this project is a linear search algorithm that may return zero, one, or multiple items. The condition for the returned items was designed such that every item in the list must be examined, thereby forcing the best case, worst case, and average case to take as close to the same time as possible (namely, O(n)).

This project’s linear search algorithm looks over a series of directories, subdirectories, and files on a local file system in order to find any and all files that are images and are less than 3,000,000 bytes in size. A file is considered an image file if its extension is one of **jpg**, **jpeg**, **gif**, or **png**. Since it cannot be known if an arbitrary file meets these conditions, and all such files must be returns, every file must be searched before the algorithm can be finished.

The entire local file system is not searched; only a subset of the file system is searched. For the purpose of this project, three different directories and their subdirectories were searched. These three directories are **C:\Users\hendr\CEG7370\7**, **C:\Users\hendr\CEG7370\214**, and **C:\Users\hendr\CEG7370\1424**. The number of the left-most directory is named after the amount of files in that directory. Therefore, **C:\Users\hendr\CEG7370\7** has seven files, **C:\Users\hendr\CEG7370\214** has 214 files, and **C:\Users\hendr\CEG7370\1424** has 1,424 files.

# Implementation Design Description

The linear search algorithm was implemented using Java’s stream API. Java’s stream API was introduced with Java SE 8 in early 2014.

Java provides two types of streams: serial streams and parallel streams. Serial streams (which are just called *streams*) process data in a normal, sequential manner. Parallel streams process data concurrently, taking advantage of any multithreading capability of multicore computers. This project compares the difference in time between the two.

The file system is traversed by using the static **walk** method in the **java.nio.file.Files** class. While the **Files** class was introduced in 2011 with Java SE 7, the static **walk** method was introduced with Java SE 8. This method returns a path stream (**Stream<Path>** in the code) which is autoclosable.

Autoclosable, along with try-with-resources, was introduced with Java SE 7. It allows any IO object to be closed without explicitly calling the object’s close method. Before Java SE 7 and try-with-resources, outputting the first line in a file might appear as follows:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15** | BufferedReader br = **null**;  **try** {  br = **new** BufferedReader(**new** FileReader("path"));  System.***out***.println(br.readLine());  } **catch**(IOException e ) {  e.printStackTrace();  } **finally** {  **try** {  **if** (br != **null**) {  br.close();  }  } **catch**(IOException e ) {  e.printStackTrace();  }  } |

With try-with-resources implemented, the same functionality might appear as follows:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **try** (BufferedReader br = **new** BufferedReader(**new** FileReader("path"));) {  System.***out***.println(br.readLine());  } **catch**(IOException e ) {  e.printStackTrace();  } |

The search parameters are specified in the stream object’s **filter** method, which takes a method reference that returns a Boolean. Method references and lambdas were introduced in Java SE 8; method references follow the form **[object]::[method]** for instance methods and **[class]::[method]** for static methods. The **[object]** part of instance method references can either be a variable name or the keyword **this**.

The method passed into the steam’s **filter** method is also called **filter**. It uses basic Java String manipulation to determine if the file ends with a predetermined extension (as mentioned in **Algorithm Description** section, this is one of **jpg**, **jpeg**, **gif**, or **png**). It then extracts file size using the **BasicFileAttributes** class and compares the date using the date time API introduced in Java SE 8:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26** | **private** **static** **final** List<String> ***IMAGE\_EXTENSIONS*** =  Arrays.*asList*(".jpg",".jpeg",".gif",".png");  **public** **boolean** filter(Path path) {  **try** {  String pathString = path.toString();  **int** dot = pathString.lastIndexOf(".");    **if**(dot > 0) {  String extension = pathString  .substring(dot)  .toLowerCase();  **if**(***IMAGE\_EXTENSIONS***.contains(extension)) {  BasicFileAttributes attr = Files  .*readAttributes*(path, BasicFileAttributes.**class**);    **return** attr.size() < 3\_000\_000;  }  }  } **catch**(IOException e) {  e.printStackTrace();  }  **return** **false**;  } |

The two different types of streams are implemented by creating an abstract class **ImageFileSearch** with one abstract method as well as the **filter** method described previously, and then extending that abstract class into two separate concrete classes **ParallelImageFileSearch** and **SerialImageFileSearch**. The abstract method is called **search**, which takes **String** argument representing a path, and returns a list of paths (**List<Path>** in the code). The implementation of this method is nearly identical in both concrete classes. The key difference is that in the implementation in the **ParallelImageFileSearch** class, the stream calls its **parallel** method before it calls its final method.

The final method called by the stream object in both **ParallelImageFileSearch** and **SerialImageFileSearch** is **collect**, which executes the stream and returns one of Java’s collection objects, such as a list or set. This method takes a **Collector** object that specifies the type of collection. In the case of this project, **Collector.toList()** was used.

Below is the **search** method implemented by **SerialImageFileSearch**:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11** | **public** List<Path> search(String pathname) {  **try** (Stream<Path> pathStream = Files.*walk*(Paths.*get*(pathname))) {  **return** pathStream  .filter(**this**::filter)  .collect(Collectors.*toList*());  } **catch** (IOException e) {  e.printStackTrace();  }  **return** **new** ArrayList<>();  } |

The following is the **search** method implemented by **ParallelImageFileSearch**, with the **parallel** method called on line 4:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **public** List<Path> search(String pathname) {  **try** (Stream<Path> pathStream = Files.*walk*(Paths.*get*(pathname))) {  **return** pathStream  .parallel()  .filter(**this**::filter)  .collect(Collectors.*toList*());  } **catch** (IOException e) {  e.printStackTrace();  }  **return** **new** ArrayList<>();  } |

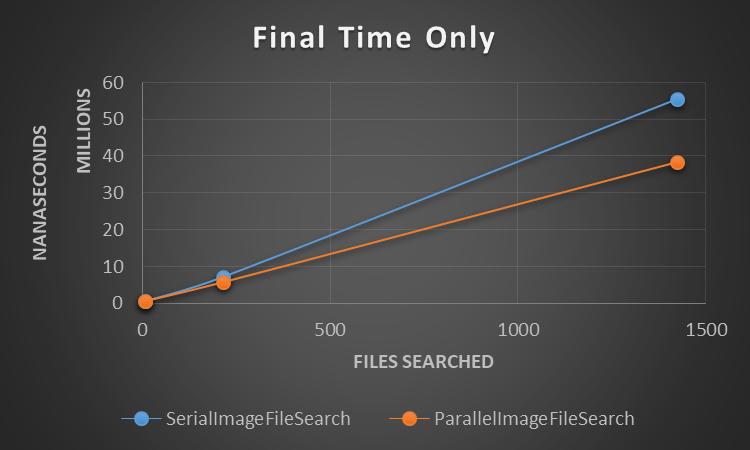
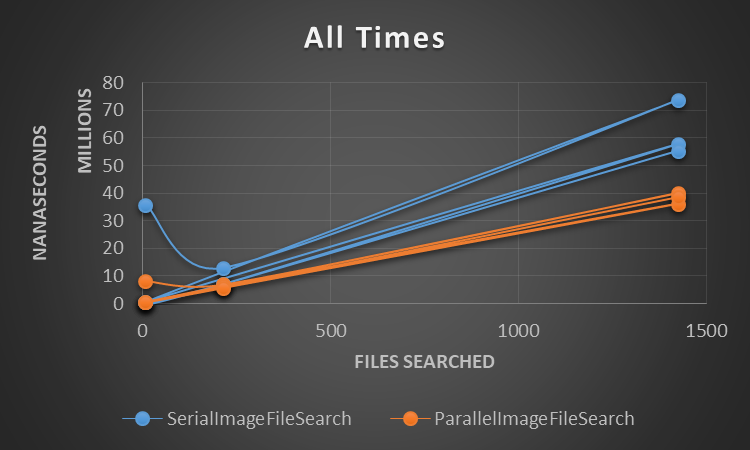
## Testing

Testing was done using Java’s standard **main** method. This **main** method was implemented in the **ImageSearch** class. Furthermore, the **ImageSearch** class contains a **test** instance method that measures the time in nanoseconds to execute the **search** method. Each individual call of the test instance method tests the search method for each of the test directories mentioned in the algorithm description section (namely, **C:\Users\hendr\CEG7370\7**, **C:\Users\hendr\CEG7370\214**, and **C:\Users\hendr\CEG7370\1424**). The test is then executed three times for each concrete class.

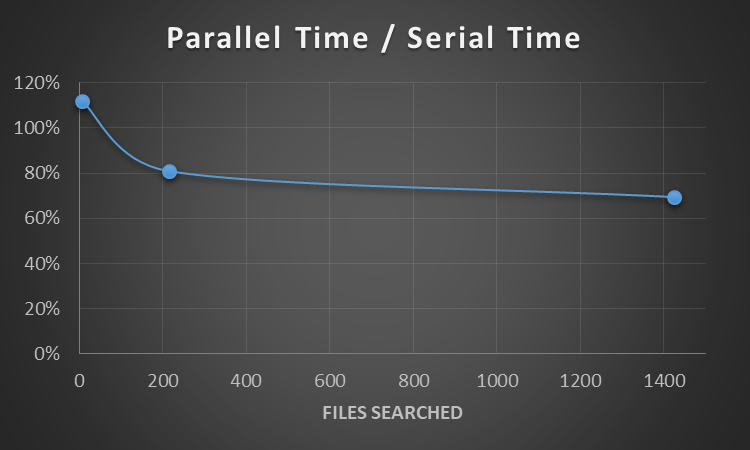
|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28** | **public** **static** **final** String[] ***PATHS*** = {"C:\\Users\\hendr\\CEG7370\\7",  "C:\\Users\\hendr\\CEG7370\\214",  "C:\\Users\\hendr\\CEG7370\\1424"};    **public** **static** **void** main(String[] args) **throws** IOException {  ImageFileSearch[] tests = {**new** SerialImageFileSearch(),  **new** ParallelImageFileSearch()};  System.***out***.println("directory\tclass\t# images\tnanoseconds;");  **for**(ImageFileSearch test : tests) {  **for**(**int** i = 0; i < 3; i++) {  test.test();  }  }  }    **public** **void** test() {  **for**(String path : ***PATHS***) {  **long** start = System.*nanoTime*();  **long** count = search(path).size();  **long** end = System.*nanoTime*();  **long** time = end - start;    System.***out***.printf("%s\t%s\t%d\t%,d%n",  path, **this**.getClass().getSimpleName(), count, time);  }  } |

# Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | files searched | images found | SerialImageFileSearch | ParallelImageFileSearch |
| 1st iteration | 7 | 6 | 35,826,276 ns | 8,272,604 ns |
| 214 | 160 | 12,966,600 ns | 6,912,994 ns |
| 1424 | 793 | 73,740,479 ns | 39,905,107 ns |
| 2nd iteration | 7 | 6 | 590,589 ns | 596,221 ns |
| 214 | 160 | 7,060,705 ns | 6,414,308 ns |
| 1424 | 793 | 57,785,349 ns | 36,082,019 ns |
| 3rd iteration | 7 | 6 | 498,174 ns | 558,078 ns |
| 214 | 160 | 7,056,865 ns | 5,719,015 ns |
| 1424 | 793 | 55,346,703 ns | 38,445,400 ns |

****

|  |  |
| --- | --- |
| files searched | Parallel time / serial time |
| 7 | 112% |
| 214 | 81% |
| 1424 | 69% |

****

# Discussion

The first time **search** is ran takes exceedingly longer than any other time **search** is ran. This is true regardless if **search** is called first via **SerialImageFileSearch** or **ParallelImageFileSearch**, or the amount of files to be searched. This is most likely due to caching and Java loading the class.

**ParallelImageFileSearch** performed better when searching 1,424 files and 214 files, whereas **SerialImageFileSearch** performed better when searching only 7 files. This is most likely due to any overhead incurred by parallel streams. It is notable that searching 1,424 files via a parallel stream took approximately 69% of the time it took to search via a serial steam, whereas searching 214 files via a parallel stream took approximately 81% of the time it took to search via a serial steam. This improved performance over a greater number of files indicates that any overhead with parallel streams does not increase as much when searching a greater number of files – it may even remain constant.

# Implementation Source Code

## ImageFileSearch.java

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69** | **package** hendrix11.search.image;  **import** java.io.IOException;  **import** java.nio.file.Files;  **import** java.nio.file.Path;  **import** java.nio.file.attribute.BasicFileAttributes;  **import** java.util.Arrays;  **import** java.util.List;  **public** **abstract** **class** ImageFileSearch {    **private** **static** **final** List<String> ***IMAGE\_EXTENSIONS*** =  Arrays.*asList*(".jpg",".jpeg",".gif",".png");  **private** **static** **final** String[] ***PATHS*** = {"C:\\Users\\hendr\\CEG7370\\7",  "C:\\Users\\hendr\\CEG7370\\214",  "C:\\Users\\hendr\\CEG7370\\1424"};  **public** **static** **void** main(String[] args) **throws** IOException {  ImageFileSearch[] tests = {**new** SerialImageFileSearch(),  **new** ParallelImageFileSearch()};  System.***out***.println("directory\tclass\t# images\tnanoseconds;");    **for**(ImageFileSearch test : tests) {  **for**(**int** i = 0; i < 3; i++) {  test.test();  }  }  }  **public** **void** test() {  **for**(String path : ***PATHS***) {  **long** start = System.*nanoTime*();  **long** count = search(path).size();  **long** end = System.*nanoTime*();  **long** time = end - start;  System.***out***.printf("%s\t%s\t%d\t%,d%n",  path, **this**.getClass().getSimpleName(), count, time);  }  }  **public** **abstract** List<Path> search(String pathname);  **public** **boolean** filter(Path path) {  **try** {  String pathString = path.toString();  **int** dot = pathString.lastIndexOf(".");    **if**(dot > 0) {  String extension = pathString  .substring(dot)  .toLowerCase();  **if**(***IMAGE\_EXTENSIONS***.contains(extension)) {  BasicFileAttributes attr = Files  .*readAttributes*(path, BasicFileAttributes.**class**);  **return** attr.size() < 3\_000\_000;  }  }  } **catch**(IOException e) {  e.printStackTrace();  }  **return** **false**;  }  } |

## ParallelImageFileSearch.java

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27** | **package** hendrix11.search.image;  **import** java.io.IOException;  **import** java.nio.file.Files;  **import** java.nio.file.Path;  **import** java.nio.file.Paths;  **import** java.util.ArrayList;  **import** java.util.List;  **import** java.util.stream.Collectors;  **import** java.util.stream.Stream;  **public** **class** ParallelImageFileSearch **extends** ImageFileSearch {  @Override  **public** List<Path> search(String pathname) {  **try** (Stream<Path> pathStream = Files.*walk*(Paths.*get*(pathname))) {  **return** pathStream  .parallel()  .filter(**this**::filter)  .collect(Collectors.*toList*());  } **catch** (IOException e) {  e.printStackTrace();  }  **return** **new** ArrayList<>();  }  } |

## SerialImageFileSearch.java

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23** | **package** hendrix11.search.image;  **import** java.io.IOException;  **import** java.nio.file.Files;  **import** java.nio.file.Path;  **import** java.nio.file.Paths;  **import** java.util.ArrayList;  **import** java.util.List;  **import** java.util.stream.Collectors;  **import** java.util.stream.Stream;  **public** **class** SerialImageFileSearch **extends** ImageFileSearch {  @Override  **public** List<Path> search(String pathname) {  **try** (Stream<Path> pathStream = Files.*walk*(Paths.*get*(pathname))) {  **return** pathStream  .filter(**this**::filter)  .collect(Collectors.*toList*());  } **catch** (IOException e) {  e.printStackTrace();  }  **return** **new** ArrayList<>();  }  } |

# Implementation Documentation

## ImageFileSearch

**private** **static** **final** List<String> ***IMAGE\_EXTENSIONS***

A list of image file extensions in lowercase and including the dot (.).

**private** **static** **final** String[] ***PATHS***

An array of the path to the directories to search for each test.

**public** **static** **void** main(String[] args)

The main entry point to the program. Any input arguments are ignored and not used for this program. This method runs the tests as well.

**public** **void** test()

Runs a single test for the current instance and outputs the path name, class name, number of files found, and the amount of time taken in nanoseconds.

**public** **abstract** List<Path> search(String pathname)

Abstract method that must be implemented by any concrete classes that extend this class. Takes a path name as a **String** and returns a list containing any and all paths that return true when passed to the filter method.

**public** **boolean** filter(Path path)

Takes a **Path** object and returns **true** if its **String** representative ends with one of the extensions in **IMAGE\_EXTENSIONS** and the associated file is less than three million bytes in size. It returns false otherwise.

## ParallelImageFileSearch

This class extends **ImageFileSearch** and overrides the abstract method **search** in a parallel manner.

## SerialImageFileSearch

This class extends **ImageFileSearch** and overrides the abstract method **search** in a serial manner.

# Appendix: CS 7370 Programming Project Outline

* Requirements
  + The summer 2018 version of CS 7370 requires a programming project. The requirement of this programming project is to implement a simple parallel/distributed algorithm which involves multiple processes that communicate.
* Language to Be Used
  + Java
* Algorithm to Be Implemented
  + Parallel search
* Report Sections
  + Algorithm Description
  + Implementation Design Description
  + Results
  + Discussion
  + Implementation Source Code
  + Implementation Documentation
* Due Dates
  + Project Outline (this paper): June 18, 2018
  + Presentation: TDB, but sometime in class from July 16 through July 23, 2018
  + Report and other deliverables: July 23, 2018