Shawn Jones
CSCI 322
Final Project

Executive Summary

The Rabin-Karp string matching algorithm computes a rolling hash by moving from the beginning of the text being searched to the end. By breaking the text up into equal-sized chunks and assigning each chunk to a thread, the processing time of Rabin-Karp can be improved. Preliminary output suggests that this strategy does indeed work up to one thread per physical core.

Statement of Need & Motivation

String matching algorithms allow us to find occurrences of a pattern within a given text. The naive method is to simply iterate through each position in the text, and look for the substring there. This results in a time complexity of $O(n*m)$, where n is the size of the text, and m is the size of the pattern. Although this is closer to $O(n)$ when searching texts based on human languages, this isn't the case when searching through text with a relatively small-sized alphabet. An example of this is searching for a text pattern in the human genome.

The Rabin-Karp algorithm achieves a time complexity of $O(n)$ regardless of the size of the alphabet or type of text. It performs m steps to compute the hash value of the substring at index zero. Each successive hash value is calculated based off of the previous index's hash value. In this manner, it only has a constant amount of work to do at each index after the first. Can we use concurrency to help the runtime performance of Rabin-Karp? It seems that by breaking up a large piece of text into several smaller but equal-sized pieces of text, we could easily give a smaller-sized version of the initial problem to each core we have available.

Program Description

The code attached to the appendix of this document does just that in an attempt to determine the benefit of using concurrency with Rabin-Karp. Most Rabin-Karp pseudocode starts by calculating the hash value of the pattern, then looking for matches as it calculates the hash values of each substring of size *m* in the text. When a matching hash value is found, a character-by-character comparison is used to determine whether this is a true match or just a hash collision. In the implementation used here, the hash value of each substring of length *m* is computed first, and only the calculation of substring hash values for the text is timed. This removes the time variance associated with checking each possible match, which has a frequency that is dependent on the constants chosen for the hash function and the specific pattern being searched for.

A small amount of extra work is needed when breaking this algorithm up into smaller chunks. Although it is a small price to pay, it is worth mentioning that each thread must do its own initial work to calculate the first hash value for its portion of the text. This is very small and only adds *m-1* initial steps. There is also a bit of overlap into the beginning of the next thread's chunk when one thread is calculating hash values for substrings near the end of its chunk. This is due to the fact that each thread is responsible for calculating hash values for substrings of length *m* that have starting indexes in its assigned chunk. This also meant that care had to be taken to make sure the thread with the last chunk adjusted its ending index so that its last substring calculated ended at the last character of text.

Preliminary Results

The implementation was timed on a machine with an eight-core hyperthreading processor. Execution times are shown for one thread, two threads, four threads, and eight threads. Each of those thread pool sizes was tested against text of sizes ~11KB, ~22.5KB, ~45KB, and ~90KB. The text consisted of “Lorem Ipsum” dummy text, generated by the website www.lipsum.com. Hash values were calculated for substrings of length five. Each runtime reported is the average value of four executions.

	Average	**runtime**	**(seconds)**	
Filesize (bytes)	1 thread	2 threads	4 threads	8 threads
11,170	0.0144344593	0.0127203160	0.0110969135	0.0119267053
22,349	0.0204686053	0.0176120413	0.0144127513	0.0149385588
44,884	0.0330266518	0.0254898768	0.0212246260	0.0201231253
89,545	0.0558002340	0.0400904690	0.0289156165	0.0270207600

Figure 1

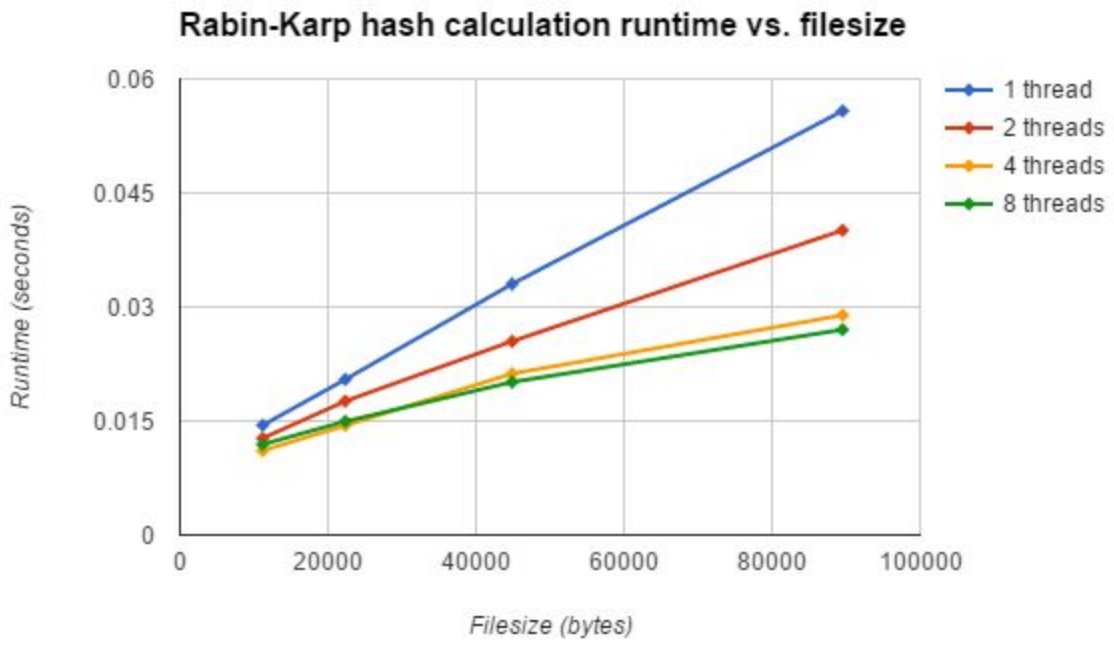

Figure 2

Conclusions

The benefit of concurrency seems to be more favorable as text size increases, likely because the threading overhead has been justified by the increased data size. Surprisingly, the preliminary data only shows decreasing runtimes for up to four threads on the test machine. Since the test machine is using a processor with four physical cores, but eight logical cores (Intel hyperthreading), it seems safe to conclude that hyperthreading was simply providing no benefit in this application.

In hyperthreading processors, two logical cores share one physical core's cache and other resources. It is likely that one thread is using each physical core's shared resources intensely enough that the second thread running on that core is starved for those resources. A low cache-miss rate can cause this. Nevertheless, the potential for further decrease in runtimes via execution in an environment with more physical cores is obvious.

References

Cormen, Thomas H. "The Rabin-Karp Algorithm." *Introduction to Algorithms*. Cambridge, MA: MIT, 2009. 990-94. Print.

Karp, R. M., and M. O. Rabin. "Efficient Randomized Pattern-matching Algorithms." *IBM Journal of Research and Development* 31.2 (1987): 249-60. Web.

"Introduction to String Searching Algorithms." *Topcoder*. N.p., n.d. Web. 11 Mar. 2016. <https://www.topcoder.com/community/data-science/data-science-tutorials/introduction-to-string-searching-algorithms/>.

Appendix

Source code:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class rabinKarpThreaded {

   private final char[] text;
   private final int textSize;
   private int subStringLength;
   private int base;
   private int bigPrime;
   private ConcurrentMap<Integer, List<Integer>> hashes;

   public rabinKarpThreaded(String file, int subStringLength, int base, int bigPrime)
     throws IOException {
       this.text = readFile(file);
       this.textSize = this.text.length;
       this.subStringLength = subStringLength;
       this.base = base;
       this.bigPrime = bigPrime;
       this.hashes = new ConcurrentHashMap<>();
   }

   // getters
   public char[] getText() {
       return text;
   }
   public int getTextSize() {
       return textSize;
   }
   public int getSubStringLength() {
       return subStringLength;
   }
   public int getBase() {
       return base;
   }
   public int getBigPrime() {
       return bigPrime;
   }
   public ConcurrentMap<Integer, List<Integer>> getHashes() {
       return hashes;
   }

   // converts input file to String, replacing line separators with spaces
   private char[] readFile( String file ) throws IOException {
       BufferedReader reader = new BufferedReader( new FileReader(file));
       String line = null;
       StringBuilder stringBuilder = new StringBuilder();
```

```
        try {
            if ( ( line = reader.readLine() ) != null ) {
                stringBuilder.append( line );
            }
            while( ( line = reader.readLine() ) != null ) {
                stringBuilder.append( " " );
                stringBuilder.append( line );
            }

            return stringBuilder.toString().toCharArray();
        } finally {
            reader.close();
        }
    }

    // calculates a modulus b (not a remainder b)
    //  result will be positive, even if a < 0
    public int modulus(long a, long b)
    {
        return (int)((a % b + b) % b);
    }

    // code to calculate hash of a pattern
    public int computePatternHash(String patternString) {
        char[] pattern = patternString.toCharArray();

        int hashValue = 0;
        for (int i=0; i<pattern.length; i++) {
            hashValue = modulus(hashValue * base + pattern[i], bigPrime);
        }

        return hashValue;
    }

    // thread worker code
    class computeTextHashes implements Runnable {
        private int indexStart;
        private int indexEnd;
        private int textSize;
        private int subStringLength;
        private int base;
        private int bigPrime;
        private long E;

        public computeTextHashes(int start, int end) {
            this.indexStart = start;
            this.indexEnd = end;
            this.textSize = getTextSize();
            this.subStringLength = getSubStringLength();
            this.base = getBase();
            this.bigPrime = getBigPrime();
            this.E = (long)Math.pow(getBase(), getSubStringLength()-1);
        }

        public void run() {

            // check for erroneous parameters
            if (this.indexStart < 0) {
                throw new IllegalArgumentException("negative indexStart (" + this.indexStart +
                        ") found by thread " + Thread.currentThread().getId());
```

```
        }
        if (this.indexEnd >= getTextSize()) {
            throw new IllegalArgumentException("indexEnd (" + this.indexEnd +
                    ") out of range found by thread " + Thread.currentThread().getId());
        }
        if (this.subStringLength > this.textSize) {
            throw new IllegalArgumentException("subStringLength (" + this.subStringLength +
                    ") is larger than textSize (" + this.textSize + ")");
        }

        // calculate hash value of substring text[0..subStringLength-1]
        int hashValue = 0;
        for (int i=this.indexStart; i<(this.indexStart + this.subStringLength); i++) {
            hashValue = modulus(hashValue * base + text[i], bigPrime);
        }
        // add hash value of first substring to hash map
        List<Integer> valueList = Collections.synchronizedList(new ArrayList<>());
        valueList.add(this.indexStart);
        if (hashes.putIfAbsent(hashValue, valueList) == null) {
            // value added successfully
        } else {
            // there is already a mapping, add to the existing value list
            hashes.get(hashValue).add(this.indexStart);
        }


        // does this worker have the chunk of text at the end?
        if (this.indexEnd >= (getTextSize() - (this.subStringLength-1))) {
            // yes
            this.indexEnd = (getTextSize()-1)-(this.subStringLength-1);
        } else {
            // no -- leave indexEnd as is
        }


        // calculate the hash values for all substrings of size subStringLength
        //  and starting index i
        for (int i=this.indexStart+1; i<=this.indexEnd; i++) {
            valueList = Collections.synchronizedList(new ArrayList<>());
            valueList.add(i);

            hashValue = modulus(hashValue - modulus(text[i-1] * E, bigPrime), bigPrime);
            hashValue = modulus(hashValue * base, bigPrime);
            hashValue = modulus(hashValue + text[i+(subStringLength-1)], bigPrime);

            if (hashes.putIfAbsent(hashValue, valueList) == null) {
                // value added successfully
            } else {
                // there is already a mapping, add to the existing value
                hashes.get(hashValue).add(i);
            }
        }
    }
}


public static void main(String[] args) throws IOException {
   int numThreads = 8;
    String file = "lorem90k.txt";
    String pattern = "ipsum";
```

```
        int subStringLength = pattern.length();
        int base = 457;
        int bigPrime = 6131;
        rabinKarpThreaded myRK = new rabinKarpThreaded(file, subStringLength, base, bigPrime);
        int chunkSize = myRK.getTextSize() / numThreads;

        // get start time
        double startTime = System.nanoTime();

        // create threads
        try {
            ExecutorService ex = Executors.newFixedThreadPool(numThreads);
            for (int i=0; i<numThreads; i++) {
                ex.execute(myRK.new computeTextHashes(i*chunkSize, (i+1)*chunkSize - 1));
            }
            ex.shutdown();
            ex.awaitTermination(1, java.util.concurrent.TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        // get end time
        double endTime = System.nanoTime();

        // print out time and thread information
        System.out.println(numThreads + " threads took "
                + ((endTime-startTime)/1E9) + " sec to calculate hashes.");

        // print out input size
        System.out.println("Text size: " + myRK.getTextSize() + " bytes.");

        // get hash of pattern
        int patternHash = myRK.computePatternHash(pattern);

        // print out pattern
        System.out.println("Pattern: " + pattern);

        int matches = 0;
        int spuriousHits = 0;
        String possibleMatch;
        List<Integer> matchList = new ArrayList<>();

        // iterate through matching hashes
        for (int offset : myRK.getHashes().get(patternHash)) {
            possibleMatch = String.valueOf(myRK.getText(), offset, 5);
            if (possibleMatch.equals(pattern)) {
                matches++;
                matchList.add(offset);
            } else {
                spuriousHits++;
            }
        }

        Collections.sort(matchList);
        System.out.println("Matches at indexes: " + matchList.toString());
        System.out.println("Matches: " + matches);
        System.out.println("Spurious hits: " + spuriousHits);

    }
}
```

Sample output:

```
8 threads took 0.024564088 sec to calculate hashes.
Text size: 89545 bytes.
Pattern: ipsum
Matches at indexes: [6, 276, 551, 2861, 3220, 3631, 5640, 5975, 6735, 6817, 9295, 10114, 11067, 11515,
11557, 11939, 13140, 13724, 14556, 15190, 15704, 17509, 19224, 19403, 20763, 22294, 23151, 23361,
23976, 24291, 24439, 24690, 25876, 26415, 27418, 27454, 28189, 30003, 31162, 32508, 33000, 33317,
33367, 33831, 34009, 34700, 35173, 37633, 38200, 40477, 41775, 42850, 43282, 44859, 51095, 51958,
52146, 52510, 52975, 53462, 54608, 55052, 55850, 56334, 57646, 57808, 59280, 60070, 61306, 62215,
62644, 63627, 64412, 64476, 65346, 65820, 65938, 66404, 67473, 68772, 68958, 69421, 69835, 71364,
71504, 72565, 74005, 74235, 75179, 75420, 76265, 76652, 76732, 78553, 79623, 79791, 81325, 81900,
82027, 83128, 83861, 84892, 85087, 87061, 87807, 88482, 88536]
Matches: 107
Spurious hits: 3
```