Design Lab Project Report
on

Parallel Sequence Comparison using Apache Spark

by
Sunandita Patra
09CS3037
5th year Dual Degree
COMPUTER SCIENCE AND ENGINEERING
IIT KHARAGPUR
Autumn 2013-14

Introduction

String matching is an important algorithm in the field of bio-informatics. It has its application in many problems like nucleotide/protein/DNA sequence matching. Nucleotides and amino acids have certain properties determined by biological scientists, for example, a primer is a conserved DNA sequence used in the Polymerase Chain Reaction (PCR) to identify the location of the DNA sequence that will be amplified (amplification starts at the location immediately following the 3 end primer, known as the forward primer). Finding if a DNA sequence contains a specific (candidate) primer is therefore paramount to the ability to run correct PCR. A conserved DNA sequence is a sequence of nucleotides in DNA, which is found in the DNA of multiple species and/ormultiple strains (for bacteria/prokaryotes), or, in general, in all/almost all sequences in a specific collection. Some sequences are conserved precisely.

In some cases, the sequences are conserved with some modifications. Finding such modified strings is an important process for mapping DNA of a new organism, based on the known DNA of a related organism.

Literature Survey

Many sequential algorithms exist for the string matching problem and are widely used in practice. The better known are those of Knuth Morris and Pratt and Boyer and Moore. These algorithms achieve $O(n + m)$ time which is the best possible in the worse case and the latter algorithm performs even better on average. Another well known algorithm which was discovered by Aho and Corasik searches for multiple patterns over a fixed alphabet. Many variations on these algorithms exist and an excellent survey paper by Aho covers most of the techniques used. All these algorithms use an $O(m)$ auxiliary space. Figure 1 shows the development of non-parallel string matching algorithms over the years.

Parallel string matching algorithms are mostly developed on parallel random access machine (PRAM) computation model. This model consists of some processors with access to a shared memory. There are several versions of this model which differ in their simultaneous access to a memory location. The weakest is the exclusive-read exclusive-write EREW-PRAM model where at each step simultaneous read operation and write operations at the same memory location are not allowed. A more powerful model is the concurrent read exclusive write CREW-PRAM model where only simultaneous read operations are allowed. The most powerful model is the concurrent-read concurrent-write CRCW-PRAM model where read and write operations can be simultaneously executed.

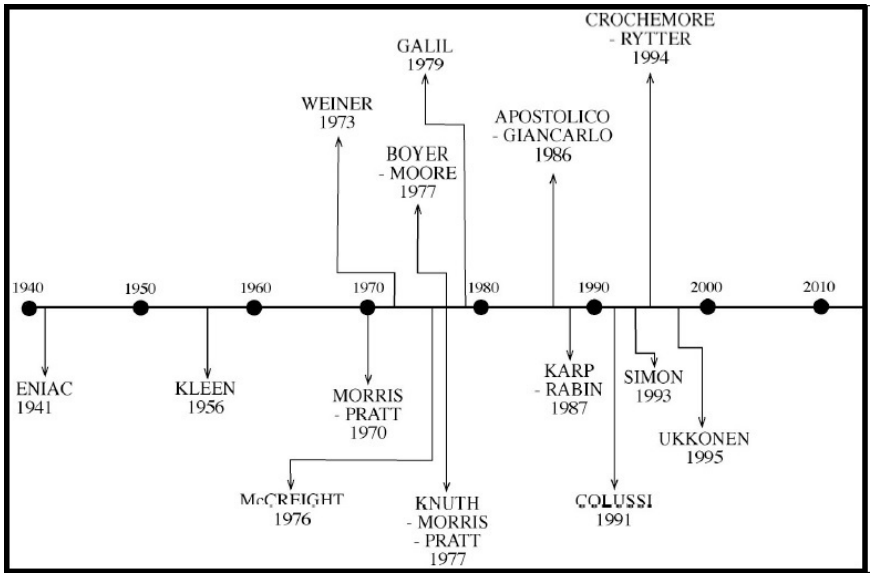

Figure 1: The figure shows the history of development of sequential string matching algorithms.

Parallel Sequence Comparison using Spark

We are aiming to develop a parallel sequence comparison algorithm which is scalable, i.e., it runs at a reasonable speed for large sequences and the performance is dependent on the number of computing units available. With increase in the amount of resources available, the time required by our algorithm should decrease. Spark is a MapReduce-like cluster computing framework designed for low-latency iterative jobs. It provides us with some basic elements of parallel computation and our algorithm tries to make use of these elements.

Our algorithm is designed as follows:
Input:A, B, the two sequences to be matched
Output: $List[(i_A, j_A, i_B, j_B)]$ which denotes that $A[i_A...j_A]$ matches with $B[i_B...j_B]$

When the length of A or B is 1, we can simply search the single character in the other string, and return the list of indices found.

When length of A or B is not 1, we identify four parallel sub-problems as being :

1. $List0$: Matching between $A[i_A...mid_A]$ and $B[i_B...mid_B]$
2. $List1$: Matching between $A[i_A...mid_A]$ and $B[mid_B + 1...j_B]$
3. $List2$: Matching between $A[mid_A + 1...j_A]$ and $B[i_B...mid_B]$
4. $List3$: Matching between $A[mid_A + 1...j_A]$ and $B[mid_B + 1...j_B]$

The above sub-problems are in parallel because they are independent of each other.

After this step, we need to merge the results obtained in the previous step. We observe that elements of $List0$ and $List1$ can be merged to give a single match if the ending point (j_A, j_B) of any element of $List0$ just precedes the starting point (i_A, i_B) of $List1$. So, we map elements of $List0$ to their endpoints $((j_A+1, j_B+1))$ and elements of $List1$ to their beginning points, (i_A, i_B). Similar mapping approach can be followed for merging $List2$ and $List3$.

Once the 4 lists are mapped, the ReduceByKey operation is be done on the mapped lists. Elements with the same key are reduced to form a single match. For example, if the key of an element of $List0$ matches with the key of an element of $List1$, it means they are sub-parts of a bigger match. As a result, they can be combined to form a match of larger size.

Thus, the elements of $List0$ and $List1$ are merged to form $List01$, and the elements of $List2$ and $List3$ are merged to form $List23$. Now, the elements of $List01$

are mapped to their lower end points and elements of $List23$ are mapped to their upper endpoints and then reduction is done accordingly using the ReduceByKey operation.

The code of the algorithm is attached at the end of the report.

Experimental Results

Our algorithm was implemented in Scala using the Spark API and deployed in standalone mode. The standalone cluster consisted of 6 working cores. Random sequences upto length 100 were compared and the average time required to compute the matching for 10 sequences of each length. Figure 2 shows the average time required to compute matching sequences of length 50 with number of cores varying from 1 to 6. Figure 3 shows the same for sequences of length 100.

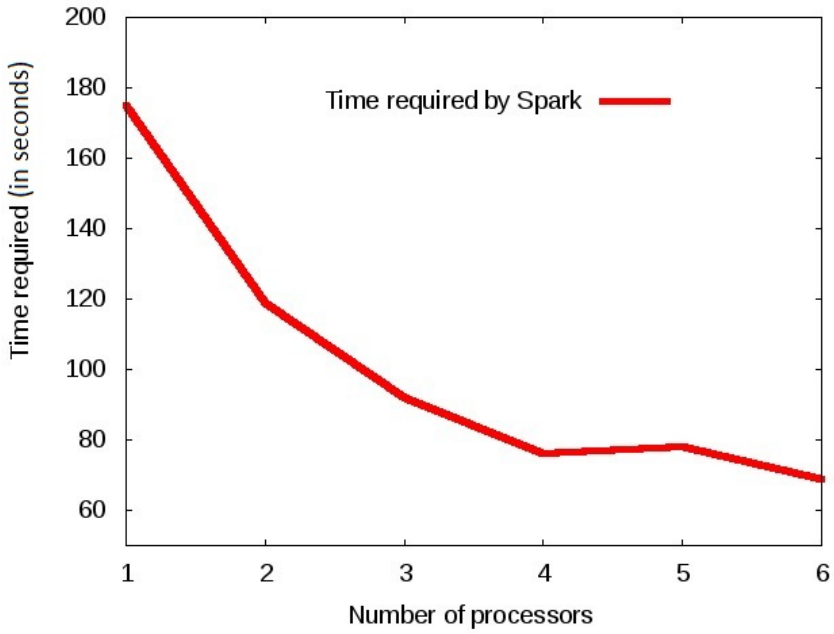

Figure 2: The figure shows the variation of the average time required for sequence comparison DNA sequences of length 50 with the number of processors available for computation.

Figure 4 compares the time required by sequences of length 50 and 100 when run on k processors, where k varies from 1 to 6.

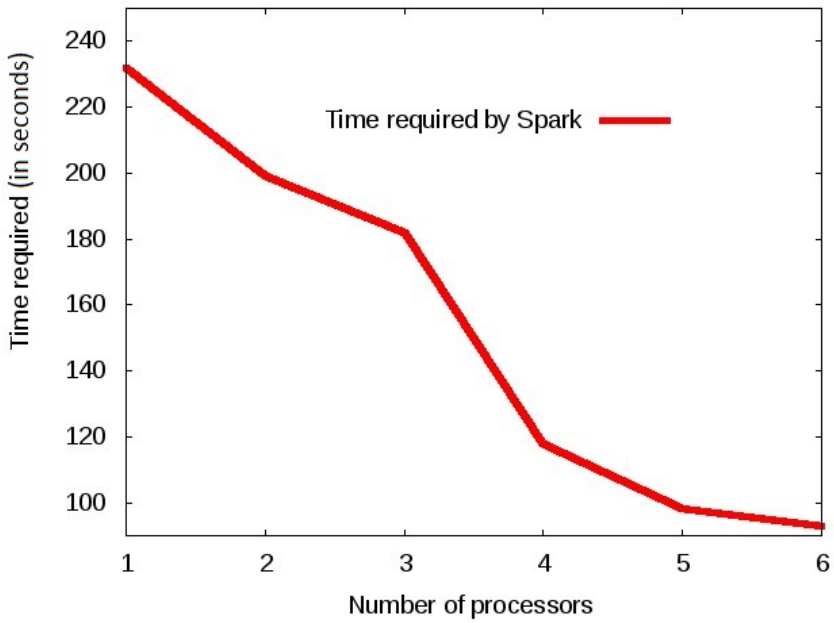

Figure 3: The figure shows the variation of the average time required for sequence comparison DNA sequences of length 100 with the number of processors available for computation.

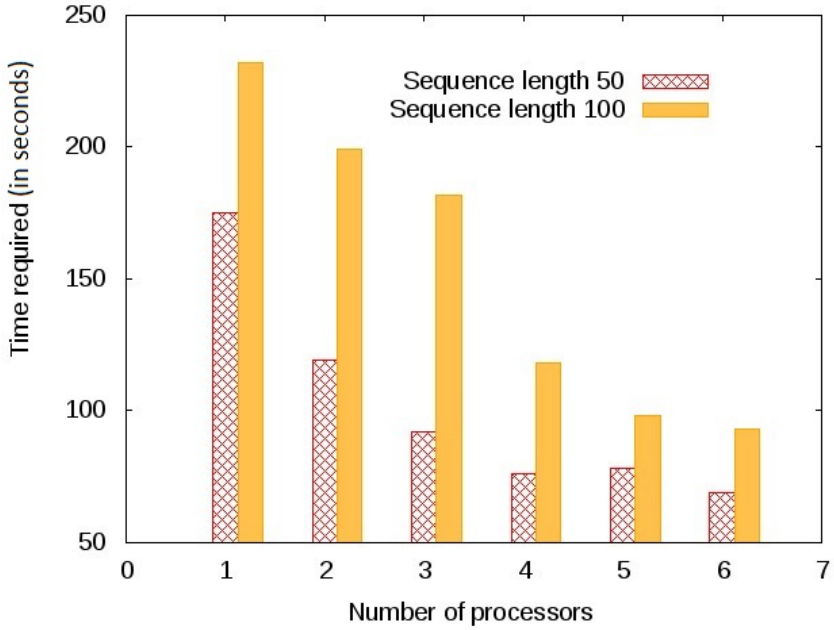

Figure 4: The figure compares the average time required by spark to compare sequences of lengths 50 and 100 using 1 to 6 processors.

Conclusion and Future Scope

We conclude that the performance our algorithm improves greatly with increase in the number of processors available for computation. This shows how our algorithm can take advantage of the parallel processing power of a cluster computing framework. Further, compatibility of spark with Hadoop file system ensures that large DNA sequences can be compared. In the future, analysis for such large sequences can be done to study the performance of our algorithm. Attempts can be made to do approximate string matching by extending the algorithm to include a tunable parameter which denotes the degree to which some part of the two strings being compared differ from each other.

References

1. Breslauer, Dany, and Zvi Galil. "An optimal time parallel string matching algorithm." SIAM Journal on Computing 19.6 (1990): 1051-1058.

2. Evans, D. J., and S. Ghanemi. "Parallel string matching algorithms." Kybernetes 17.3 (1988): 32-44.

3. Dan Gusfield (1997), Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology, Cambridge University Press, 1997.

4. Donald E. Knuth, James H. Morris, Vaughan R. Pratt, (1977), Fast Pattern Matching in Strings, SIAM Jorunal on Computing, Volume 6, pp. 323-350, 1977.

5. Design and evaluation of parallel string matching algorithms for network intrusion detection systems, 2007, v. 4672 LNCS, p. 344-353

6. Parallel Processing of Biological Sequence Comparison Algorithms, Elizabeth W. Edmiston, Nolan G. Core, Joel H. Saltz, and Roger M. Smith, International Journal of Parallel Programming, Vol. 17, No. 3, 1988

7. A Space-Efficient Parallel Sequence Comparison Algorithm for a Message-Passing Multiprocessor, Xiaoqiu Huang, International Journal of Parallel Programming, Vol. 18, No. 3, 1989

8. A Multi-threaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison, W.S. Martins, Pacific Symposium on Biocomputing 6:311-322 (2001)

```
/*** SequenceComparison.scala ***/

import scala.math.random
import spark._
import spark.SparkContext
import spark.util.Vector
import spark.SparkContext._
import scala.collection.mutable.HashMap
import scala.collection.mutable.HashSet

object SequenceComparison {

    val spark = new SparkContext("spark://MASTER", "SequenceComparison",
     "./", List("target/scala-2.9.3/sequencecomparison_2.9.3-1.0.jar"));

  def parseDNA(line: String):(Int, String, String, Int, Int, Int, Int) =
  {
     val seq = line.split(' ');
     val m = seq(0).length();
     val n = seq(1).length();
     println("Sequence 1 : " + seq(0) + " Sequence 2 : "+seq(1) );
     return(0, seq(0), seq(1), 0, m-1, 0, n-1);
  }

  def createFormat(line1: String, line2: String):String = {
      return (line1 + "_" + line2);
  }

  def createFormat2(line: String):
           (Int, String, String, Int, Int, Int, Int) = {
      val seq = line.split('_');
      val m = seq(0).length();
      val n = seq(1).length();
      return(0, seq(0), seq(1), 0, m-1, 0, n-1);
  }

  def mapper1(input:(Int, Int, Int, Int, Int)):
                      ((Int, Int, Int), (Int, Int, Int, Int)) = {
    println("In mapper 1");
    val index = input._1;
    val iA = input._2;
    val jA = input._3;
    val iB = input._4;
    val jB = input._5;

    if(index == 0 || index == 1) {
      val newindex = 0;
      if(index == 0) {
        val newi = jA + 1;
        val newj = jB + 1;
        return ((newindex, newi, newj), (iA, jA, iB, jB));
      } else {
        val newi = iA;
        val newj = iB;
```

```
      return ((newindex, newi, newj), (iA, jA, iB, jB));
    }
  }
  else
  {
    val newindex = 1;
    if(index == 2) {
      val newi = jA + 1;
    val newj = jB + 1;
    return ((newindex, newi, newj), (iA, jA, iB, jB));
    } else {
    val newi = iA;
    val newj = iB;
    return ((newindex, newi, newj), (iA, jA, iB, jB));
    }
  }
}

def reducer1(input1:((Int, Int, Int), (Int, Int, Int, Int)),
               input2:((Int, Int, Int), (Int, Int, Int, Int))):
                     ((Int, Int, Int), (Int, Int, Int, Int)) = {

  val index = input1._1._1;
  val iA1 = input1._2._1;
  val jA1 = input1._2._2;
  val iB1 = input1._2._3;
  val jB1 = input1._2._4;

  val iA2 = input2._2._1;
  val jA2 = input2._2._2;
  val iB2 = input2._2._3;
  val jB2 = input2._2._4;

  if(iA1 < iA2) {
    val newiA = iA1;
    val newjA = jA2;
    val newiB = iB1;
    val newjB = jB2;
    val newkeyi = newjA + 1;
    val newkeyj = newjB + 1;
    return((index, newkeyi, newkeyj), (newiA, newjA, newiB, newjB));
  } else {
    val newiA = iA2;
    val newjA = jA1;
    val newiB = iB2;
    val newjB = jB1;
    val newkeyi = newjA + 1;
    val newkeyj = newjB + 1;
    return((index, newkeyi, newkeyj), (newiA, newjA, newiB, newjB));
  }
}
```

```
def reducer2(input1:(Int, Int, Int, Int),
               input2:(Int, Int, Int, Int)):(Int, Int, Int, Int) = {
 println("in reducer 2");
  val iA1 = input1._1;
  val jA1 = input1._2;
  val iB1 = input1._3;
  val jB1 = input1._4;

  val iA2 = input2._1;
  val jA2 = input2._2;
  val iB2 = input2._3;
  val jB2 = input2._4;

  if(iA1 < iA2) {
    val newiA = iA1;
    val newjA = jA2;
    val newiB = iB1;
    val newjB = jB2;
    val newkeyi = newjA + 1;
    val newkeyj = newjB + 1;
    return(newiA, newjA, newiB, newjB);
  } else {
    val newiA = iA2;
    val newjA = jA1;
    val newiB = iB2;
    val newjB = jB1;
    val newkeyi = newjA + 1;
    val newkeyj = newjB + 1;
    return(newiA, newjA, newiB, newjB);
  }
}

def mapper2(input:((Int, Int, Int), (Int, Int, Int, Int))):
               ((Int, Int), (Int, Int, Int, Int)) = {

  println("in mapper 2");
  val index = input._1._1;
  val iA = input._2._1;
  val jA = input._2._2;
  val iB = input._2._3;
  val jB = input._2._4;

  if(index == 0) {
    val newkeyi = jA + 1;
    val newkeyj = jB + 1;
    return((newkeyi, newkeyj),(iA, jA, iB, jB));
 } else {
    val newkeyi = iA;
    val newkeyj = iB;
    return((newkeyi, newkeyj),(iA, jA, iB, jB));
 }
}
```

```
def reducer3(input1:(Int, Int, Int, Int),
               input2:(Int, Int, Int, Int)):(Int, Int, Int, Int) = {

  println(" in reducer 3");
  val iA1 = input1._1;
  val jA1 = input1._2;
  val iB1 = input1._3;
  val jB1 = input1._4;

  val iA2 = input2._1;
  val jA2 = input2._2;
  val iB2 = input2._3;
  val jB2 = input2._4;

  if(iA1 < iA2) {
    val newiA = iA1;
    val newjA = jA2;
    val newiB = iB1;
    val newjB = jB2;
    return(newiA, newjA, newiB, newjB);
  } else {
    val newiA = iA2;
    val newjA = jA1;
    val newiB = iB2;
    val newjB = jB1;
    return(newiA, newjA, newiB, newjB);
  }

}

def calcMatch(input:(Int, String, String, Int, Int, Int, Int)):
               (Seq[(Int, Int, Int, Int, Int)]) = {

  println(" in calcMatch");
  val index = input._1;
  println(index);
  val A = input._2;
  val B = input._3;
  val iA = input._4;
  val jA = input._5;
  val iB = input._6;
  val jB = input._7;

  println("A = "+A);
  println("B = "+B);
  println("iA = "+iA);
  println("jA = "+jA);
  println("iB = "+iB);
  println("jB = "+jB);
```

```
  if(iA == jA || iB == jB) {
    if(A(0) == B(0)) {
    val iRC = Seq((index, iA, jA, iB, jB));
    return(iRC);
   }
    else
    return Seq();
  }
  else
  {
    val midA = (iA + jA) / 2;
    val midB = (iB + jB) / 2;
    val A1 = A.substring(iA - iA, midA + 1 - iA);
    val B1 = B.substring(iB - iB, midB + 1 - iB);
    val A2 = A.substring(midA + 1 - iA, jA + 1 - iA);
    val B2 = B.substring(midB + 1 - iB, jB + 1 - iB);

    val p0 = (0, A1, B1, iA, midA, iB, midB);
    val p1 = (1, A2, B1, midA + 1, jA, iB, midB);
    val p2 = (2, A1, B2, iA, midA, midB + 1, jB);
    val p3 = (3, A2, B2, midA + 1, jA, midB + 1, jB);

    val rec = Seq(p0, p1, p2, p3);
    println("recursive sequence created");
    val recinput = spark.parallelize(rec);
    println("parallelized");
    val recoutput = recinput.flatMap(calcMatch);
    println("recursive calls has returned");
    val maxMatch =
    recoutput.map(mapper1).reduceByKey(reducer2).map(mapper2)
                                        .reduceByKey(reducer3);
    println("reduce step over");

    val indexedResult = maxMatch.map(x => (index, x._2._1, x._2._2,
    x._2._3, x._2._4));
    println("Mapped to index");
    println("returning");
    val iRC = indexedResult.collect();

    var i = 0;
    for(i <- 0 to iRC.size - 1)  {
    val x = iRC(i);
    println(" returning "+x._1+ " : "+x._2+ ", "+x._3+", " + x._4 + ",
    "+x._5);
    }
    return(iRC);
  }
  return Seq();
}

def main(args: Array[String]) {


  val distFile = spark.textFile("Encodings.txt");
```

```
    val cands = distFile.map(parseDNA);

    val common = cands.flatMap(calcMatch);
    val iRC = common.collect();


      var i = 0;
       for(i <- 0 to iRC.size - 1)  {
      val x = iRC(i);
      println(" finally "+x._1+ " : "+x._2+ ", "+x._3+", " + x._4 + ",
"+x._5);
       }
         System.exit(0)
  }
}
```