CloudBlast: Fast Sequence Alignment through Hadoop

Sam Wischnewsky
Williams College

1 Introduction

The field of Computational biology has ballooned in the last few decades. A fundamental problem of bioinformatics research is sequence alignment. This is simply a more complex version of string matching in which we allow insertion and deletion events in the query and search strings in an attempt to model biological DNA mutation events. The cutting edge algorithm for sequence alignment is BLAST. This allows a user to enter a query sequence and search it against a large database very quickly because of the faster runtime of BLAST compared to other string matching techniques. In this project, I sought to develop a Hadoop based implementation of BLAST. The implementation works by mapping the key steps of BLAST into map and reduction operations to allow for parallel computation. I take advantage of optimizations in Hadoop like the distributed cache to minimize run time by removing repeated work across nodes as much as possible. This implementation of CloudBlast aims for run time speed at the expense of increased bandwidth use in the system. Further details are discussed in the architecture section below. This implementation was tested on the application of Alu searching within the human genome. The results of my testing indicate that the implementation allows for much more efficient searching through large genome databases than can be achieved using a single node architecture. This open source CloudBlast implementation could easily be adapted to use proprietary sequence databases which could otherwise not be hosted on NCBI's implementation of BLAST.

2 Architecture

2.1 BLAST and Sequence Alignment

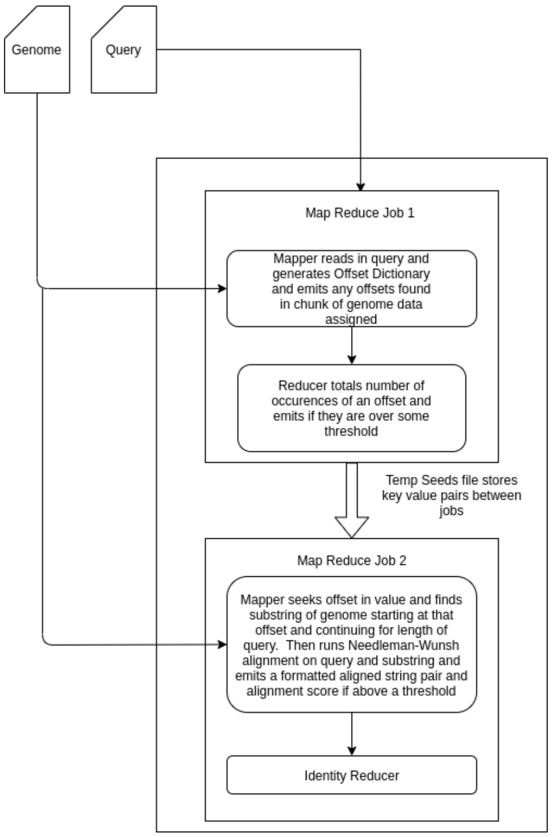
BLAST is an algorithm designed for rapid sequence alignment across an extremely large search space. Original algorithms for sequence alignment include Needleman-Wunsh and Smith-Waterman among others. These techniques generally depend on dynamic programming and use O(mn) space where m is the length of the query string and n is the length of the searched string. If we want to search for a small subsequence across a human chromosome then applying dynamic programming to it would take an exorbitant amount of space. These algorithms are still useful for doing sequence alignment but only when we have two strings of about our query length. That is where BLAST comes into the picture. BLAST involves first searching across the searchable string for good seed sites where we can then run a more expensive dynamic programming alignment. The identification of seeds is done by just looking for matching substrings in the query and search string and finding the offset between them. For example, if we were searching for ”Hello there” in the text of a novel we might search for everywhere we found ”there” in the book. If we find ”there” at character index 550 then it is offset from its position in the query string by 544 characters. Now if we also found ”Hello” at position 540 then it would be offset by 540. If we looked at a histogram of our offsets we would expect to see peaks around places where lots of substrings are close. For the application of nucleotide alignment the standard length of a subword is 6. So in this case we search for substrings of length 6 that appear in the query and then calculate their offsets in the searched string. At peaks in offsets we then can run a dynamic programming alignment algorithm to check how good of a match we have found.

2.2 BLAST through MapReduce

A BLAST job lends itself towards two map reduce jobs. The first job is focused on finding good seed sights and in the implementation is referred to as the ”seed generation job”. The second job uses these seed sites as the basis for running a dynamic programming string matching algorithm. I implemented these two jobs using the Apache MapReduce library in Java. The two jobs are connected through a temporary folder called temp_seeds. Each job is timed and analysis of their relative run times follows in the Evaluation section.

2.2.1 Seed Generation Job

The seed generation job takes the genome files as input. Thus the MapReduce framework deals with the necessary details of splitting up the input files for different map tasks. This job then calls the OffSetMapper class. The OffsetMapper has an overridden setup function which is run one time per node. This means that we can do work that is common to all lines from the genome file in this function and save time. So hear we read in the query file and parse it into substrings of length 6 and generate an offset dictionary for the query string. Then we use this offset dictionary in the mapper to generate real offsets for observed substrings in the genome. These are emitted as their location rounded down to the nearest multiple of 50. This rounding technique is used to amalgamate common keys to the same reduction functions. The reduction functions then just total the number of substrings near offsets and emit that total if it is over some threshold value. In my testing I used a threshold of 10. I chose this because you generally want to search sights that have longer continuous substrings. These are more likely to translate into high scores. For example an identical substring between the query and genome of length 10 will cause 5 intermediate key value pairs. If we had two substrings of length 6 each that don’t overlap then we are using 12 characters to only get a bump in the score of 2. So this heavy weighting towards continuous substrings meant that we saved time. The fundamental trade off here is between sensitivity and speed. To achieve a satisfyingly fast run time you set a fairly high threshold and to achieve the most accurate search sensitivity you set the threshold lower. In my case, I chose a higher threshold to facilitate testing.


Genome
Query
Map Reduce Job 1
Mapper reads in query and generates Offset Dictionary and emits any offsets found in chunk of genome data assigned
Reducer totals number of occurences of an offset and emits if they are over some threshold
Temp Seeds file stores key value pairs between jobs
Map Reduce Job 2
Mapper seeks offset in value and finds substring of genome starting at that offset and continuing for length of query. Then runs Needleman-Wunsh alignment on query and substring and emits a formatted aligned string pair and alignment score if above a threshold
Identity Reducer

Architecture Diagram

2.2.2 Dynamic Programming Job

This job is meant to take seeds and determine whether or not they are appropraite matches for the query. Luckily this technique has been developed by computational biologists years ago. I implemented an adapted Needleman-Wunsch algorithm for my string matching. In my implementation, I ran the algorithm using the query and the seeded substring plus the surrounding 20 characters. This gave me a larger window to search for the query and I hoped would raise the scores I saw. Then based on the scoring algorithm for Needleman-Wunsch I outputed those query and genome substrings that matched well. These were dumped with a string of each including insertion events along with the scoring for matching them.

2.3 Distributed Data Sharing

My architecture required that more data was available to Mappers than simply the key value pairs coming as inputs. This was generally handled in setup methods to try to avoid adding higher overhead to map functions. But it involved the passing of the genome and query text data to mappers so they could use it either to generate offest dictionaries or to perform string matching on substrings of the genome data. In both cases I passed the file paths through the job contexts. Each MapReduce job has a context variable that stores lots of information from the hadoop configuration files. It also contains a user area where programmers can store additional key value pairs called parameters. I used the API for contexts to set parameters in the main function and access them from within the Mapper classes. This worked when running locally because all of the paths were just to local files on my computer. It failed once I tried to run this job in a distributed environment. The failures manifested themselves as

```
Error: org.apache.hadoop.hdfs.
BlockMissingException: Could not
obtain block: BP-2017514945-
172.31.51.239-1588712990160
at Blast$GlobalAlignmentMapper.map
```

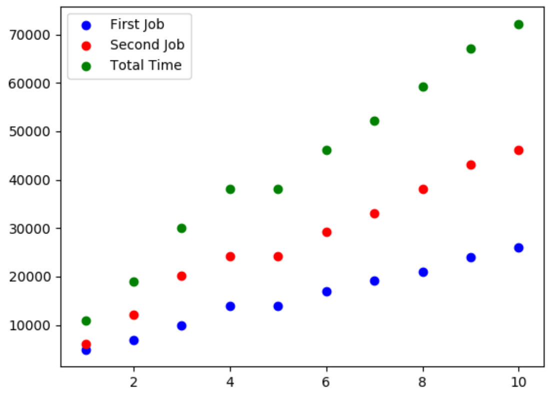
In these cases it appears that the data node was searching for a block when trying to perform the second mapper task. In this case I was trying to use the seek method to randomly access bytes within the inputted genome file. This behavior is apparently supported by hadoop although I could not get a cluster to function with this method call.

2.3.1 Attempted Technologies

I attempted file sharing between the main class and map reduce jobs using a variety of interfaces suffering from various degrees of deprecation. I think the original solution supported by Hadoop 2.x with x less than 8 was a class called DistributedCache. The distributed cache was meant to be a way to easily share small files and metadata across nodes without having to use the overhead of the hadoop distributed file system. But this class was deprecated. The next iteration apparently was putting the distributed cache into the context variable and providing an interface of ”addCacheFile” and ”getAllCacheFiles”. Thus a programmer could theoretically add a cache file in the main method and then iterate through the list of Cache file URIs to find the one they wanted in a Mapper class. Sadly this also got deprecated. The method I finally tried was just putting the file paths directly into the context variable with get and set but sadly this didn't work out either. One method I discuss in Future work is different hadoop run time environements. I was running the normal hadoop framework but there is a more recent prototyping specific version called hadoop streaming that seems suitable to data caching.

3 Evaluation

Evaluation was performed on a single node set up. This was done because of issues running in a distributed environment as discussed previously. In a single node environment we sadly don't get to study the differences in overhead for access of larger distributed files because all files are stored on the same physical disk. The testing method was fairly simple. I searched for a repeated section of DNA known as Alu. The Alu sequence is a roughly 300 base pair sequence that encodes the the information to make a DNA splicing complex that reinserts Alu somewhere else in the DNA sequence. So Alu is almost like a genetic virus that propogates itself throughout the human genome. It is not known to encode anything useful to humans and is potentially the result of a benign genetic infection some early homo sapiens caught tens of thousands of years ago. Alu is a good test sequence because it is fairly regularly dispersed throughout the human genome so as we test larger and larger subsets we expect to be finding a comparable proportion of Alus. The genetic sequences I tested with were all subsets of human chromosome 19. I made files including the first 10 percent, 20 percent, and so on up to the entire chromosome. Then I ran the Blast job on each subset using the Alu sequence. The results of that testing are shown below.



Experimental Data

For each of the genome subsets I recorded the run time of the first, second, and sum of all jobs. As can be seen all increase roughly linearly. This result supports the assumption that Alus are roughly evenly distributed throughout this chromosome. One area where all three curves flatten is between 40 percent and 50 percent which might indiciate that there were very few Alus there. Otherwise this performance is awesome. The job is effectively O(n) in practice. One aspect of the run time I found interesting was that the second job was only about twice as slow as the first job. The second job involved a much more difficult computation but I guess it also applied to a much smaller subset of the genome space. So in the end the smaller search space compensated for the much longer computational time. Overall I'm really satisfied with this performance on one node though. Even with fairly large input files it still worked quickly with effective score cutoffs for each step.

4 Future Work

This project still has large areas for improvement. One area that could be valuable is the string matching algorithm. At this point the algorithm is penalizing indels regardless as to whether they are trailing, middle or ending. A leading or trailing indel should probably not be penalized because we are using two strings of different lengths in matching so we are gauranteed to require indels in the query string. Another area of potential improvement would be the actual run time environment. In testing and development I ran hadoop through its normal executable. But there has recently been a newly developed version called Hadoop streaming. Based on my preliminary research it seems like Hadoop streaming supports file path caching. There is a -files flag that allows you to cache files and specify a shorter path that will be available in mapper nodes by putting another name after a . This filename is just a symlink to the original file but can make accessing files from paths much easier.

5 Conclusions

The experience of developing this was frustrating at times but overall quite rewarding. The hardest part was definitely attempting the file sharing between nodes which I ended up not being able to accomplish. Getting the two jobs linked and actually outputting reasonable matches to alu sequences was really cool though. I had implemented a python program to try to iteratively do blast on a sequence and it was orders of magnitude slower than even the single node version of hadoop. This project really highlighted how versatile the MapReduce framework can be even with applications as distant as biology. It surprises me sometimes the percentage of major discoveries that come from just realizing that two known topics can be effectively connected to create a better whole. Blast and Hadoop are two powerful frameworks and combining them actually makes it super fast to search through a genome database for a test sequence. Granted combining known ideas isn't Nobel level work but it still can have some wild implications for research efficiency. Completing this project was cool and honestly taking Distributed Systems and Computational Biology at the same time has been pretty mind opening. I kind of wish I had taken Computational Biology earlier because I might have tried to get a job doing that instead of a plain Software Engineering job. But maybe I can pivot once the recession ends.