Protein Sequence Alignment

Mahsa Rezaei Firuzkuhi

Problem Description

The problem that we have addressed in this homework is called the protein sequence alignment. In this problem we will have a sequence of various amino acids and we need to identify which protein has that amino acid sequence. This problem is very challenging because a protein can consist of so many amino acids and on the other hand the list of the known proteins is very large. Since the sequences of the amino acids can be easily altered during the experiments or cannot be distinguished during the measurements, protein sequence alignment is a highly challenging task and is not a one to one matching task. We need to consider all variants of a protein sequence (amino acid) in order to find the best matching protein in the database.

Assume we have a protein sequence that we are analyzing and a sequence from the database that we are trying to compare to, the similarity between two sequence will be indicated by assigning a score value. If they are matching the assigned score is a positive value and if they are different the score is zero and if there is a gap in one of the amino acid alignment sequences, the assigned score is negative value. The final score would be sum of these values for any two sequences. In this problem we need to create all possible variants of the protein sequence that is going to be identified and compare those with every single database sequence. The big challenge here is that we have a large dataset and comparing the sequences in pairs have quadratic complexity and we need to create millions of subsequences of a sequence to identify the best match. The actual goal of this assignment is converting the sequential protein sequence alignment code which is finding 25 best matches of the protein sequence among a large dataset of protein sequences and developing the Pyspark version of that. So, we used Spark technology to be able do the operation in parallel form. In the following section we briefly introduce the spark technology and how we employ its features to speed up the protein sequence alignment problem.

Solution Strategy

Apache Spark is an open-source cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. Apache Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read only multi-set of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. Spark uses shared memory to facilitate communication between resources and this fact makes it faster compare to other technologies that do not use shared memory like Hadoop. In this section we provide a high-level description of how we parallelized the sequential code. There are multiple of aspect that can be exploit within a Spark version of code. For example, distributing the database sequences onto different executors and splitting up the iteration space onto different executors.

In this assignment, we have a sequential code that we need to develop a Spark version of that. The code takes the original sequence and generates the variants of the protein sequence, for example by introducing number of spaces at the beginning of the sequence and then compares the modified sequence with database sequence. Also, the code will make many variants of the protein sequence just by inserting a few gaps in any place of the sequence and this results the very high complexity of the problem. We are going to process two datasets of protein sequences: The first one contains only 20 sequences, and the second one is a larger database including 66 sequences. The solution for solving this problem is described in the following steps. We need to implement a Pyspark version of the code which is able to do the sequence alignment of the input protein sequence and all its variants with every single sequences of the database and then sort the matching candidates according to their similarity score and report the top 10 best matches of the input protein with the database proteins.

In the first step we read all the protein sequences in a data-set file by calling “textFile (dbfile)” which retrieves all the protein sequences in the given text file (dbfile) in the format of one column. Next, we need to modify the input protein sequence to find best alignment of that with every single sequence from the database. We made the functions provided in the initial code, return the similarity score of every single variants of the input protein sequence and every single protein sequence from the data-baes in a list of tuples instead of printing them (using the function “_GetScoresWithVariants”), and store all the collection of possible combinations in an RDD. We use flatmap transformation to parallelize the alignment. The result is a RDD that contains the following tuples (“score”, variant of input protein sequence, data-base sequence). Then, we have to use the function sortby() to sort the result RDD (which included the collection of tuples), according to the similarity score. In the sortBy() function arguments, the third argument is an integer value that set the number of partitions used in the parallel sort. We use the argument “numPartitions” to change the number of partitions. Then we need to display top N matches in an output file. We used a for loop and then write top N of the tuples in a file one by one. We used function writelines() and at the end of each tuple we added \n to change the line. We close the file once all top N tuples are written in the output file.

Description of how to run your code

To run the code named “proteinscore-with-text.py”, please use the following commands. To change the number of partitions you need to change the parameter “numPartitions” as an argument of the function sortBy() inside the code proteinscore-with-text.py . All other parameters can be set using the following two commands. For example, we must change the number of executors, it means changing the value of -N in the salloc command.

For 1 executor we insert -N 2

For 2 executor we insert -N 3

For 4 executor we insert -N 5

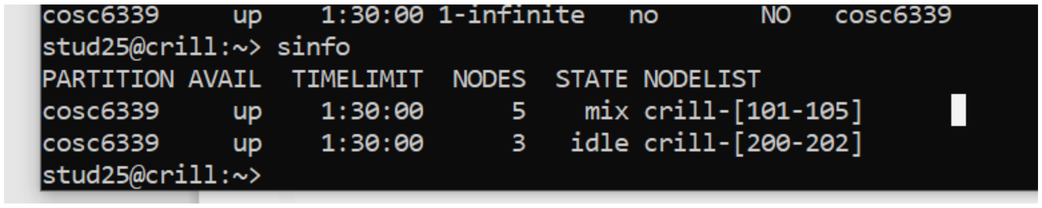
```
salloc -p cosc6339 -N [Number of executors+1] --cpus-per-task=2

crill-spark-submit -a 2 proteinscore-with-text.py protein-1.txt [Data-base name] [Maximum num of GAPs]
```

Results

In this section, we present and discuss the results of all Spark jobs

Description of resources used

We have utilized 8 compute nodes of the “Crill” cluster for our project.The properties of these nodes are listed below:



```
cosc6339     up    1:30:00 1-infinite   no       NO   cosc6339
stud25@crill:~> sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cosc6339     up    1:30:00      5    mix crill-[101-105]
cosc6339     up    1:30:00      3   idle crill-[200-202]
stud25@crill:~>
```

2 Appro 1326G4 nodes (crill-101 - crill-102), each node with

- two 2.2 GHz 12-core AMD Opteron processor (24 cores total)
- 32 GB main memory
- four NVIDIA Tesla M2050 GPUs (448 cores each) (not operational as of today)
- 4xQDR InfiniBand HCA

3 Sun Microsystems X2200 nodes (crill-103 - crill-105), each node with

- two 2.2 GHz 4-core AMD Opteron processor (8 cores total)
- 32 GB main memory
- 4xQDR InfiniBand HCA

3 HP DL 160 Gen 8 nodes (crill-200 - crill-202), each node with

- two 2.4 GHz quad-core Intel Xeon E5-2665 processor (12 cores/24 threads total)
- 16 GB main memory
- 4xQDR InfiniBand HCA

Network Interconnect

- 288 port 4xInfiniBand QDR InfiniPath Switch (donation by TOTAL)
- 24 port 4xInfiniBand SDR switch (I/O switch to the SSD storage)
- 48 port Netgear GE switch

Storage

- 4 TB NSF /home directory
- 18 TB distributed storage operated as OrangeFS (/pvfs2) or BeeGFS (/beegfs)
- 2 TB RamSan 620 SSD storage (/pvfs2-ssd) (not operational)

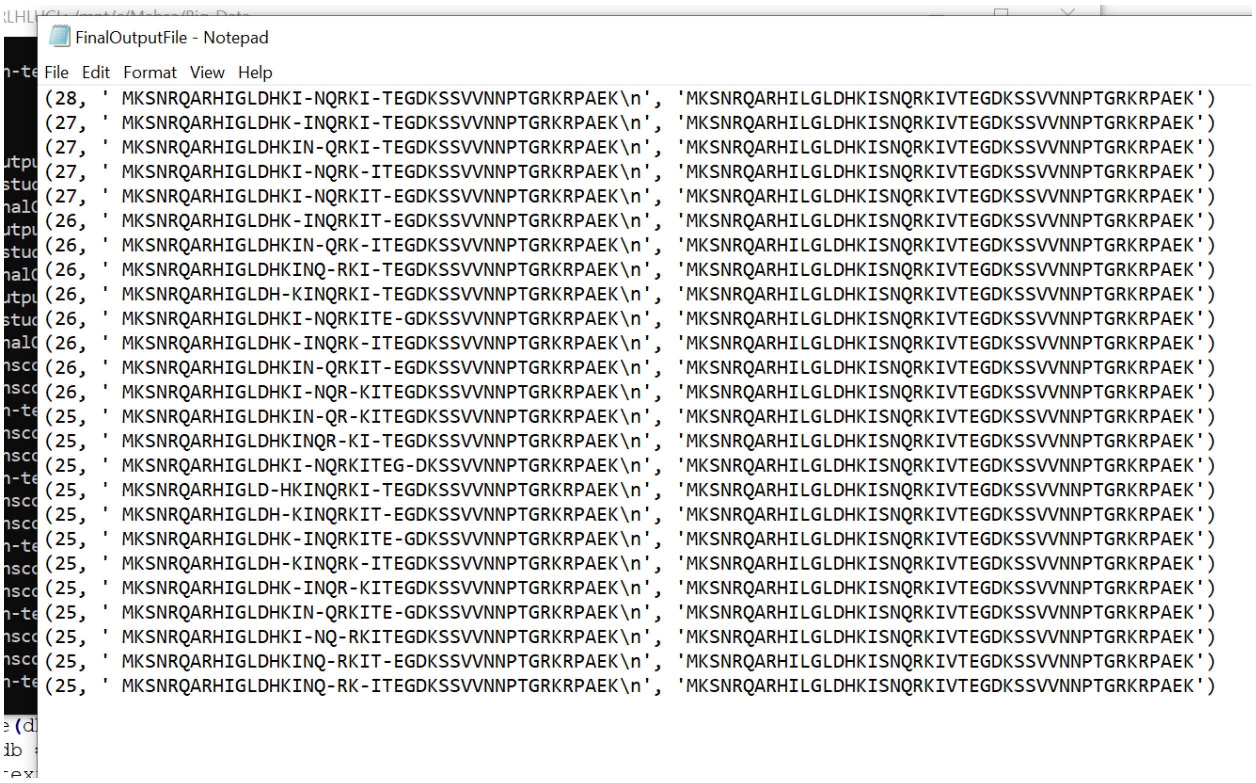
Description of measurements performed

We imported the time library and used time.time() to clock the execution time. In the following three section we will describe the details of the measurements.

Results (graphs/tables + findings)

Part1:

We developed a Pyspark version of the code that parallelizes the sequence alignment operation. The following figure shows the output file containing the top 25 matches with score and sequences used (both protein and database sequence).



```
(28, ' MKSNRQARHIGLDHKI-NQRKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(27, ' MKSNRQARHIGLDHK-INQRKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(27, ' MKSNRQARHIGLDHKIN-QRKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(27, ' MKSNRQARHIGLDHKI-NQRK-ITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(27, ' MKSNRQARHIGLDHKI-NQRKIT-EGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHK-INQRKIT-EGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHKIN-QRK-ITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHKINQ-RKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDH-KINQRKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHKI-NQRKITE-GDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHK-INQRK-ITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHKIN-QRKIT-EGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(26, ' MKSNRQARHIGLDHKI-NQR-KITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKIN-QR-KITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKINQR-KI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKI-NQRKITEG-DKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLD-HKINQRKI-TEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDH-KINQRKIT-EGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHK-INQRKITE-GDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDH-KINQRK-ITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHK-INQR-KITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKIN-QRKITE-GDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKI-NQ-RKITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKINQ-RKIT-EGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
(25, ' MKSNRQARHIGLDHKINQ-RK-ITEGDKSSVVNNPTGRKRPAEK\n', 'MKSNRQARHILGLDHKISNQRKIVTEGDKSSVVNNPTGRKRPAEK')
```

Figure 1: top 25 matches of the input protein sequence in the database

We present in the following tables the average of the three different running of the code. The sections two, three and four of the assignment is about doing the performance evaluation and changing the various parameters of the code

Part 2:

We need to change the number of executors that we are using for measurements. We consider both small and large sequence database files and running the Spark version of the code from part one, with one, two and four number of executors and evaluate how does the performance will change. We expect that the more Spark executors we are using, the smaller the execution time becomes. This performance improvement is dependent on the extra hardware employment for execution. We have measured the execution time and provide the results in the following table for different number of executors and set the “max-gap” parameter to two. We have to change the number of executors, it means changing the value of -N in the salloc command. And for each number of executors, we ran the code several times and provide the average execution time.

Number of Executors	1	2	4
Execution Time (Small-DB)	502	555	612
Execution Time(Large-DB)	1272	1517	1567
max-gap = 2			
Default partition			

Table 1: Average execution time for different number of executors for part 2 with both large and small data-sets

Finding:

Result shows that, increasing the number of executors is increasing the execution time, but its effect is too small for switching from 4 executors to 2 and one. We need to consider that increasing number of executors reduces amount of time spent on each node but, it will increase shuffling overhead so increasing number of executors is not helpful in all the cases. One important way to increase parallelism of spark processing is to increase the number of executors on the cluster. However, knowing how the data should be distributed, so that the cluster can process data efficiently is extremely important. Apache Spark manages data through RDDs using partitions which help parallelize distributed data processing with negligible network traffic for sending data between executors. A spark program can control RDD partitioning to reduce communications. In this part of assignment, the RDDs are automatically partitioned and we did not set the parameter “numPartitions” in sortBy() function. As a result, we had the default number of partitions for all the experiments. Partitioning in Spark might not be helpful for all applications, for instance, if a RDD is scanned only once, then partioning data within the RDD might not be helpful but if a dataset is reused multiple times in various key oriented operations, then partitioning data will be helpful. As we can observe from the execution time while varying the number of executors from 1 to 2 and 4, the execution time is not deceased. It has been increased slightly for both the small database (dbase-smaller) and large database (db-small). While observing the same execution time for larger databases, shows that increasing number of executors was not very effective in this application. Another reason can be number of total jobs on the cluster. Tasks run on shared university cluster. The fact observed during experiments is that based to the total job load on the cluster during the job submission, time changes significantly. Based on these observations, increasing number of executors is not helpful in all the cases and depends to the application. In our experiment, the size of the input database should be much bigger to utilize extra computing power available with higher number of executors.

Part 3:

In the second set of measurements we have changed the "Partition" argument of the Spark. The RDDs are automatically partitioned in spark. however, it is possible to change the partitioning scheme by changing the size of the partitions and number of partitions. In this experiment we changed the number of partitions using the argument "numPartitions" which is an input argument of the actions like, sortby(), sortbykey(), groupbyKey() and etc.

Ideally, if we increase the number of partitions, the level of data parallelism increases, so we expect some performance improvement. We need to find out what fraction of the execution time is dependent on the operations that are affected by the partition parameter. We used four executors and set the "max-gap" parameter to two for the evaluations reported in the following table.

Number of Partitions	1	2	3	4
Average Execution Time	954	1473	1596	1492
Large-DB				
max-gap = 2				
Num of Executors = 4				

Finding:

As mentioned in the part 2 finding, partitioning in Spark might not be helpful for all applications, for instance, if a RDD is scanned only once, then portioning data within the RDD might not be helpful but if a dataset is reused multiple times in various key oriented operations, then partitioning data will be helpful. As we can observe from the execution time while varying the number of executors from 1 to 2 and 4, the execution time is not necessarily deceased. We observe that, changing the number of partitions from 1 to 2 or 3 will increase the execution time slightly. The experiments of the execution time for large database shows that increasing number of partitions from 3 to 4, decrease the execution time. Based on these observations, increasing number of partitions is not helpful in all the cases and depends to the application and range of the change (for example if we increase the number of partitions from 10 to 15 most probably we will see the expected decrease in the execution time, while with changing number of partitions in the range of 1-4 does not shows the performance improvement and maximum resource utilization). In our example, the amount of input data should be much bigger to utilize extra computing power available with higher number of partitions. The number of partitions should be selected based on the cluster configuration and requirements of the application. We have to consider that increasing the number of partitions will make each partition have less data or no data at all and since our utilized database in these experiments are very small, we cannot see the expected performance improvment. The best way to decide on the number of partitions in an RDD is to make the number of partitions equal to the number of cores in the cluster so that all the partitions will process in parallel and the resources will be utilized in an optimal way.

Part 4:

In this section of evaluation we change the “max-gap” parameter of the algorithm between 1 and 3. We are using the small database and in case of “max-gap” equal to 3, we will have a very large variant dataset of the protein sequence and have to compare millions of possible combinations, while with value one for “max-gap” will result in a very faster sequence alignment as a result of a very small collection of possible combination. We expect that in case of the ‘max-gap” equal to one, we do not observe the benefits of the parallel resources but for the values of two and three we expect performance improvement and variations since we will have a very large dataset of all possible variants of the input protein sequence. In this set of measurements, we set numPartitions value equal to number of executors in each run.

max-gap	1	2	3
Average Execution Time	42	592	5400
Small-DB			
Num of Executors = 4			
Num of Partitions = 4			

Finding:

First, we were not able to finish the run in the case of maximum number of gaps to be 3 in the time limit of 90 minutes that we had on the shared cluster for each job. Second, as we expected, in case of the ‘max-gap” equal to one, we do not observe the benefits of the parallel resources but for the values of two and three we expect performance improvement. The code takes the original sequence and generates the variants of the protein sequence, for example by inserting a few gaps in any place of the sequence and then compares the modified sequence with database sequence. This data manipulation results in the very high complexity of the problem. As mentioned in the part 2 and 3, in our application, the amount of input data should be much bigger to utilize extra computing power available with higher number of partitions or executors and increasing the number of gaps will provide that large collection of all combinations that should be applied to the code to guarantee the performance utilization of the increasing the number of executors or number of partitions.