The University of Pavia

Department of Electrical, Computer, and Biomedical Engineering

A project report on

Substring matching with application to genomics

By:

Saurav Anand

Content

Introduction:

Our program functions in the context of looking for genes within a species' genome. The collection of all an organism's genes is known as its genome. The majority of organisms' DNA (Deoxyribonucleic Acid) is a very big molecule that resembles a long, twisted ladder. A depiction of the well-known DNA double helix can be seen here.

It is an polymer (Organic) that holds the genetic information of the organism. It is made of four kinds of nucleotides: **Adenine, Thymine, Cytosine**, and **Guanine**, which is basically represented as A, T, C, G respectively.

This code's letter placement is important since it affects how DNA behaves. Genes, which are shorter segments (substrings) of DNA that code for particular features, are found inside the DNA. Some genes do not code for specific proteins.

There are two types of cells came in picture here i.e. Eukaryotes and Prokaryotes.

Eukaryotes vs Prokaryotes:

The difference between prokaryotic and eukaryotic cells is significant. In contrast to more sophisticated living beings, prokaryotes are single-celled organisms with a simple cellular structure and simplified critical functions. Human DNAs are examples of prokaryotes. They typically have one circular chromosome (DNA). In Eukaryotic cells, a prime example of this is the human genome, which is more complicated and has a DNA made up of one or more linear DNA chromosomes.

As for our project, we have chosen the genomes of human which contain **3267134034** nitrogenous base pairs.

The genome in Fasta files/ Genome sequence (GRCh38.p13)/ALL which we have chosen from the GENCODE website.

https://www.gencodegenes.org/human/

Analysis of the Serial Algorithm

String Matching

The algorithms in search of the substrings which is called pattern-string (length=M), in a larger string text-string (length=N) are of basic importance, and it is used in different fields, including Genomics and Proteomics.

The Rabin-Karp Algorithm

The Rabin-Karp Method is the string-matching algorithm we settled on our project. The Rabin-Karp Algorithm uses a rolling-hash function to probabilistically reach performance in the range of O (M + N). Since hashing is used to look for substrings, this method of finding substrings is entirely different from brute-force.

If we want to conclude this operation in short, it can be summarized as follows:

1. For a pattern P, compute the hash function h(P)
2. Then, with the help of the same hash function for each possible substring of characters M of the text, search for a match.
3. Compare the individual characters in the substring if you discover one that has the same hash value as the pattern (Note: a matching hash does NOT imply a matching string).

Role of the Rolling-Hash Function:

Though a direct implementation based on the foregoing description would be significantly slower than a brute force search, be aware of this. It is definitely far more expensive to compute a hash function that uses each character than it is to just compare the letters. However, Rabin and Karp demonstrated that computing hash functions for character substrings M in constant time is straightforward (after some preprocessing). In the real-world scenarios we've mentioned, this results in a linear time (average performance) search for substrings.

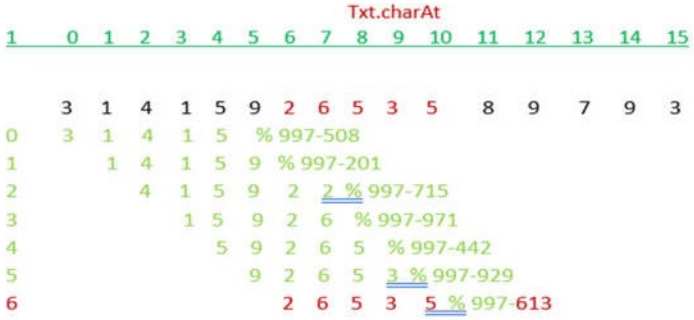
How does the Hash work?

The fundamental idea is that a string of length M equates to a base-R integer with M digits (R: alphabet size). Any base-R M-digit number must be converted into an int value between 0 and 1. In order to use a hash table of size Q for keys of this type. To do this, we require a hash function. Modular hashing is sometimes a wise choice:

$$hash(x) = x \ mod \ Q$$

Basically, we use Q, a random prime number, as large as possible

**** Example- Find the pattern in the text 3141592653589793**
R=10
Choose Q=997 and compute the hash value 26535 % 997 =613 and then look for a match



As was already said, Rabin-Karp does not compute hashing on all strings of length M in order to identify the string that has the same hash and then perform the comparison. But it uses a "mathematical technique" that Rabin-Karp has employed.

After some preparation, this method allows for the constant-time computation of hashes. This method relies on Horner's rule, which stipulates that a polynomial of degree N can be evaluated by performing precisely N additions and N multiplications.

Example: -

$a_2 * x^2 + a_1 * x + ... + a_0 = (a^2 * x + a_1) * x + a_0$

The identity holds true for Modular Arithmetic, the flavor of arithmetic used for hashing, where the following applies: -

$(a_2 * x^2 + a_1 * x + ... + a_0) \bmod z = (((a_2 * x + a_1) \bmod z) * x + a_0) \bmod z$

The Rabin-Karp method is based on efficiently computing the hash function x_{i+1} for the position $i + 1$ in the text, given its value for position i, x_i.

Example:

$a_2 * x^2 + a_1 * x + \ldots + a_0 = (a^2 * x + a_1) * x + a_0$

This identity holds true for Modular Arithmetic, the flavor of arithmetic used for hashing, where the following applies:

$(a_2 * x^2 + a_1 * x + \ldots + a_0) \bmod z = (((a_2 * x + a_1) \bmod z) * x + a_0) \bmod z$

The Rabin-Karp method is based on efficiently computing the hash function x_{i+1} for the position $i + 1$ in the text, given its value for position i, x_i. Example:

$x_i = t_i * RM^{-1} + t_i + 1 * RM^{-2} + \ldots + t_i + M^{-1} * R0$

$x_{i+1} = t_{i+1} * RM + t_i * RM^{-1} + \ldots + t_i + M * R0$

$x_{i+1} = (x_i - t_i * RM^{-1}) * R + t_i + M$

We do not need to keep the values of the numbers, but the remainder when divided by Q. Then the algorithm is used iteratively, by calculating the hash of the next string, using that of the previous one as in the formula above.

pat.charAt(j)

1	0	1	2	3	4	
	2	6	5	3	5	
0	2	%	997	=	2	
1	2	6	%	997	=	(2 * 10 + 6) % 997 = 26
2	2	6	5	%	997	= (26 * 10 + 5) % 997 = 265
3	2	6	5	3	%	997 = (265 * 10 + 3) % 997 = 659
4	2	6	5	3	5	% 997 = (659 * 10 + 5) % 997 = 613

This algorithm is the first and most well-known instance of a Monte Carlo algorithm, which has a guaranteed completion time and an arbitrary low chance of false positives.

We may or may not wish to check again by comparing each character once we obtain a hash value for a substring of text with characters M that matches the hash value of the pattern.

Double-checking reduces efficiency since each character of the text string M must be compared to each character of the pattern. If there are enough false positives, the problem becomes an O (M N) problem, much like with the brute-force technique. We can unilaterally reduce the likelihood of a collision (false-positive) by choosing a large value for Q. We set the likelihood of a collision to 1020 for Q values higher than 10^{-20}.

However, we decided to double-check the pattern and the substring each time their hashes matched in our version of the algorithm. This is because utilizing a very high number for Q can cause overflows or cause the execution to lag. This also implies that once a match is identified by our algorithm, it will undoubtedly be a match.

Our implementation of the Rabin-Karp algorithm is as follows:

It is based on the C++ code found at the following geeks for geeks website: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

```
// RABIN-KARP implementation
int search(char txt[] ,char pat[])
{
    long long int M = strlen(pat);
    long long int N = strlen(txt);
    long long int i, j;   // indexes
    long long int hash_p = 0; // hash value for pattern
    long long int hash_t = 0; // hash value for txt
    long long int h = 1;
    long long int pat_appear = 0; // the number of the pattern appearance

    // The value of h would be "pow(R, M-1)%q"
    for (i = 0; i < M-1; i++){
        h = (h*D)%Q;
    }

    // Calculate the hash value of pattern and first
    // window of text
    for (i = 0; i < M; i++){
        hash_p = (D*hash_p + pat[i])%Q;
        hash_t = (D*hash_t + txt[i])%Q;
    }
```

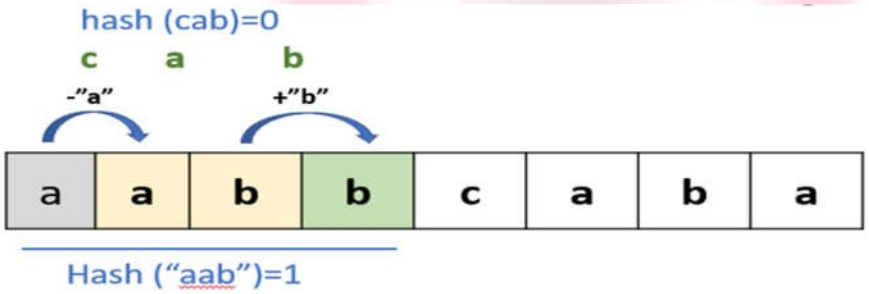
```
// Slide the pattern over text one by one
for (i = 0; i <= N - M; i++)
{
    // Check the hash values of current window of text
    // and pattern. If the hash values match then only
    // check for characters on by one
    if ( hash_p == hash_t )
    {
        // Check for characters one by one
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        // if p == t and pat[0...M-1] = txt[i...i+M-1]
        if (j == M)
            pat_appear++;
    }

    // Calculate hash value for next window of text:
    // Remove leading digit, add trailing digit
    if ( i < N-M )
    {
        hash_t = (D*(hash_t - txt[i]*h) + txt[i+M])%Q;
        // We might get negative value of t, converting it to positive
        if (hash_t < 0)
            hash_t = (hash_t + Q);
    }
}
// return the number of the pattern appearance in the given text string
return pat_appear;
```

A Prior Study of Parallelism:

Problem:

The Rabin-Karb algorithm for string matching uses a rolling-hash, as was already noted, to avoid having to compute the hash for each window of text from scratch each time and to achieve linear time completion in both the best-case and typical cases.



This effectively means that the current iteration and the one before it are always dependent on the same data. Statements in one iteration of a loop depend on statements in another iteration of the loop, which is an example of a loop-carried reliance. The algorithm itself is difficult to parallelize due to the existence of this dependence, and we are unable to employ the loop-unrolling method.

The Solution for the above problem:

Our strategy involved dividing the text into N pieces (or N "sub-texts"), where N is the number of slave processes, and allocating a sub-text to each slave process.

The Role of the Master Process:

1. The master process begins by reading the entire text as well as the pattern that needs to be searched. The text is divided into N equal pieces, each of which is sent to a slave along with a unique copy of the pattern.

2. The master then awaits the return of each slave process's result, which is the quantity of occurrences discovered in a slave's sub-text.

3. A slave uses MPI Probe and MPI Get count to determine the size of the incoming message. Then, it allots memory space large enough to hold both the pattern to be searched for and the allotted sub-text.

4. The sub-text and the pattern are separated by a predefined char (which clearly cannot appear in the text) in the message from the master, which is a single string.

5. When finished searching, the slave calls the master back and returns the results after running Rabin-Karp on the sub-text and the pattern.

A Practical Hypothesis:

We can disregard the remaining portion of the integer difference between text size and the number of slaves if the subtext is significantly greater than the pattern.

For example:

N = 7 (Number of slave processes, using N + 1 = 8 total cores).

M = 1000 (Size of the pattern).

T = 1000000 (Size of the full text).

T mod N = 1 << M

Therefore, looking for a pattern of size M in a section of text of size TmodN is neither worthwhile or even practicable.

S and P Calculations for the Amdahl-Law's Equation

S : portion of serial (non-parallelizable) code
P : portion of parallelizable code

Lines of Python codes :

The function to count how many lines in a file without multiline comment

```
def count_lines(pathname):
    """
    Counts significant lines of code in a C program without multiline comment
    """
    import re
    with open(pathname, "r") as f:
        source = f.read()
    source = source.split("\n")
    source = [line for line in source if len(line.strip()) > 0]
    source = [re.sub("\\s+", "", line) for line in source]
    source = [line for line in source if line[:2] != '//']
    return len(source)
```

Applying the function above to calculate S and P

```
# the rabin-karp part and roughly half of the main (slave).
parallelizable = count_lines("../headers/rabin_karp.h") + count_lines("../parallel/main.cpp") / 2

# total
total = parallelizable + count_lines("../parallel/main.cpp") / 2 + count_lines("../headers/read_file.h")

P = parallelizable / total
S = 1 - P
print("The percentage of serial code is:", round(S, 2))
print("The percentage of parallel code is:", round(P, 2))
```

```
The percentage of serial code is: 0.39
The percentage of parallel code is: 0.61
```

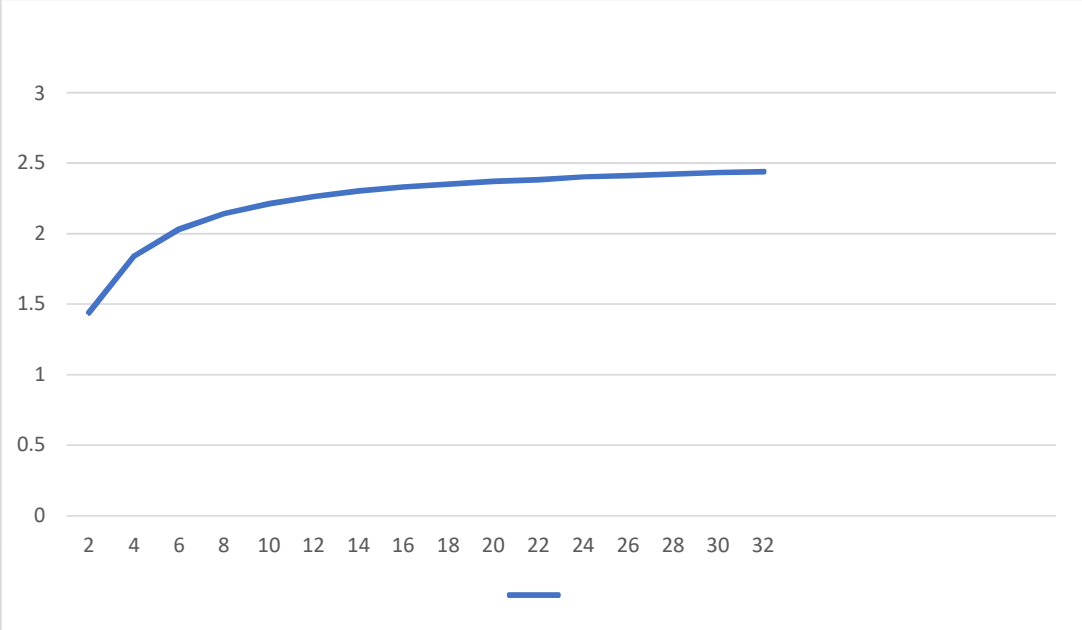
Applying Amdahl Law's Equation

```
speedup = lambda N, S, P: 1 / (S + P / N)

n_slaves = 2
print('With', n_slaves, 'slave cores, the expected speed-up is:', speedup(n_slaves, S, P))
```

```
With 2 slave cores, the expected speed-up is: 1.4367816091954024
```

Plotting the expected curve
(From 1 to 32 slaves)



MPI Parallel Implementation:

Here are the main parts of our parallel C++ implementation using open MPI.

Master's side
Send the sub-text to the slaves

```
// master sends a 'subtext'  to each of the slaves
long long int offset = 0;
for (int p = 1; p < size; ++p)
{
    string subtxt = txt.substr(offset, payLoadSize);
    string message = string(subtxt)+SEPARATOR+string(pat)+"\0";
    // cout<<"number of elements in the message: "<<message.length()<<"\n";

    ret_val = MPI_Send(message.c_str(),  message.length() , MPI_CHAR, p, TAG, MPI_COMM_WORLD);
    offset += pay_load_size;
}
```

Waits for the slaves to finish and receives the occurrences count.

```
long long int result;
long long int results=0;
// master receives results from slaves
for (int p = 1; p < size; ++p){
    //int result;
    MPI_Recv(&result, 1, MPI_INT, p, TAG, MPI_COMM_WORLD, &status);
    results+=result;
}
```

Slave's side

Probe for an incoming message from master. When probe returns, the status object has the sizeand other attributes of the incoming message.

```
MPI_Status status;
MPI_Probe(0, TAG, MPI_COMM_WORLD, &status);

// When probe returns, the status object has the size and other
// attributes of the incoming message. Get the message size
int message_size; //size of the incoming string
MPI_Get_count(&status, MPI_CHAR, &message_size);
```

So we can get the message size and allocate the necessary memory for the salve to store it.

```
// Allocate a buffer to hold the incoming chars
long long int num_bytes = sizeof(char)*(message_size);

char* buf = (char*)malloc(num_bytes);
```

Then it receives the "message" from the master, splits the string in sub-text and pattern. Finally proceeds with the search.

```
ret_val = MPI_Recv(buf, message_size, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);


// split into subtext and pattern
char* pattern;
char* text;
// get subtext
text = strtok(buf, SEPARATOR);
// get pattern
pattern = strtok(NULL, SEPARATOR);

// search for pattern appearances
long long int result = search(text, pattern);
```

Finally, send the result to the master.

```
ret_val = MPI_Send(&result, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
```

Testing and Debugging:

For our test of performance, we used
Clock () : a C++ function that returns the processor time consumed by the program, dividedby CLOCKS_PER_SEC we obtain time in seconds.

The data-sets used for tests are already mentioned previously i.e.:

- Full Human Genetic Pattern: 3267134034 base pairs
- Half Human Genetic Pattern: 1547079334 base pairs

On which the search is made for a gene of variable size, but typically in the order of a billion letters (nitrogenous bases).

Our tests for the performance and code debugging are done in the way that we have selected the two different datas that is the full human gene and the half human gene separately and run the both one by one for fat cluster and light cluster with the same number of the selected cores.

Debugging:

1. Previously the data size (about 400 Mb) was small so the running time was very quick and we were not able to do the analysis properly as the theory suggest. So, we have downloaded a big data which is almost about 3 Gb.

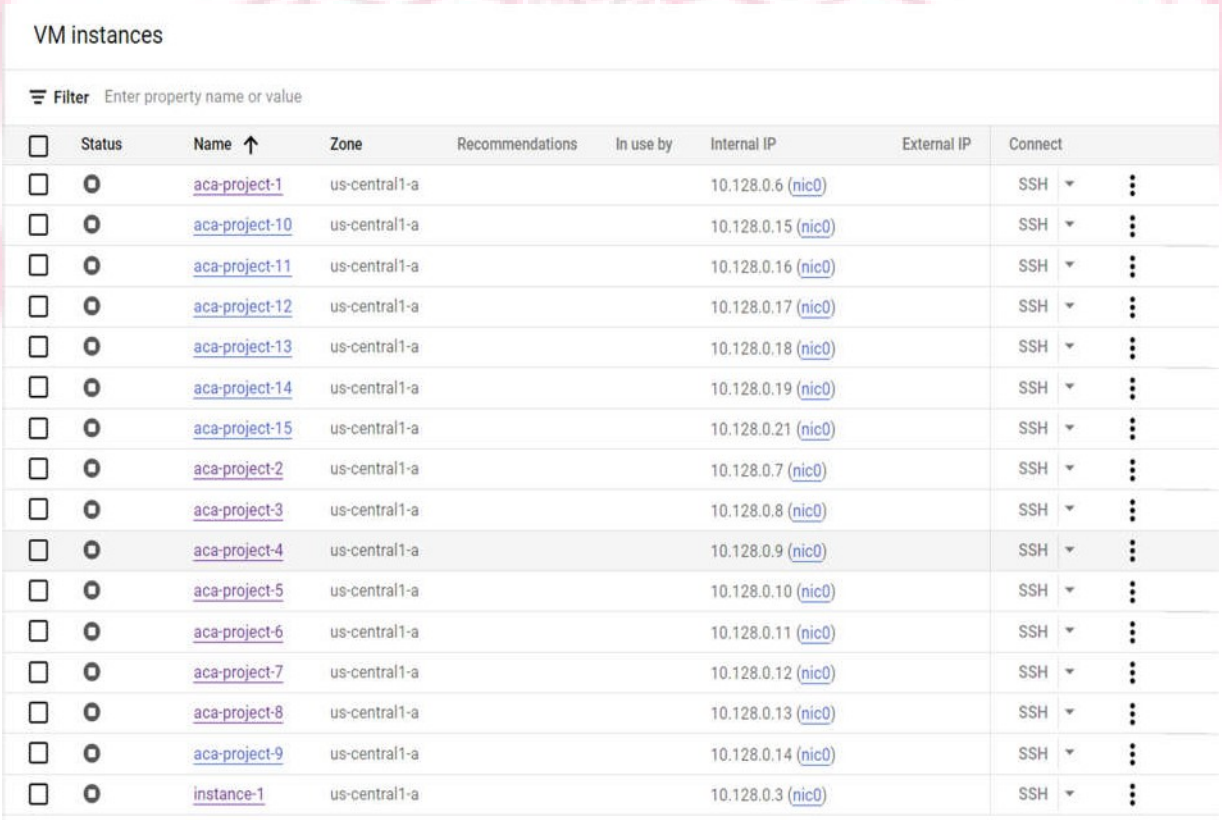
2. In our Rabin Karp Algorithm, we use INT_MAX (2147483647) in our hash function. But as our data size (3267134034) was bigger than INT_MAX, the function was not giving the correct result. So, in order to solve this problem, we used ULONG_MAX (4,294,967,295). Therefore, all of the int variables used in the serial and prallel implementation were changed to long long int as well.

3. When we ran our code for light cluster after creating the instances using the machine image the connection were not established automatically. In order to solve this problem, we have to manually create connection between the different nodes for the first time otherwise the code dose not work.

Performance and Scalability Analysis:

On the Google Cloud Platform, all performance and scalability testing were carried out.

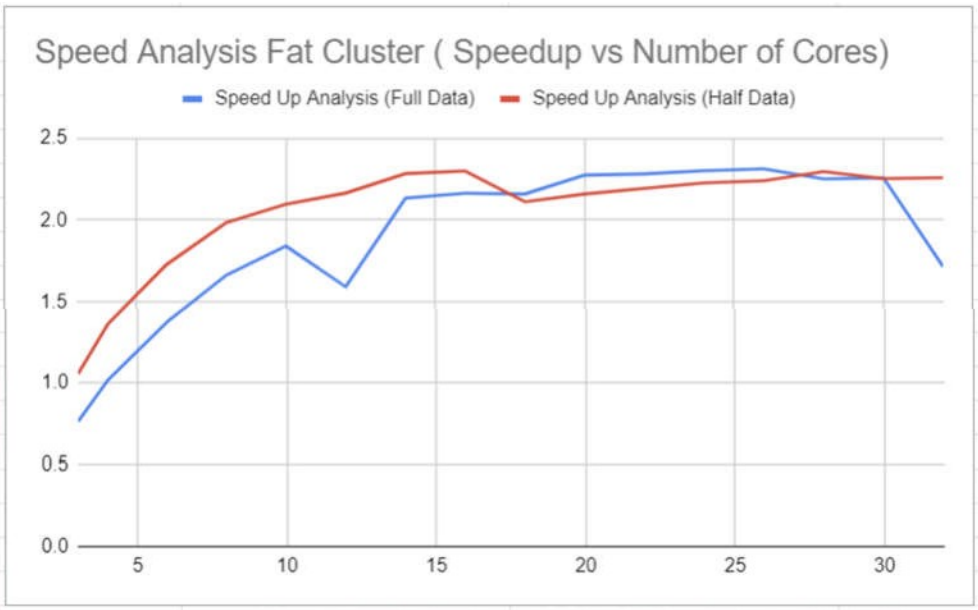
In particular, using the "light cluster" technique, we built a cluster with 16 machines (1 master and 15 slaves) has 8GB of RAM and 2 cores in each but the master having 2 cores and 16 GB of memory. Every machine is located in the same area.



Analysis of Speed Up (Performance Analysis):

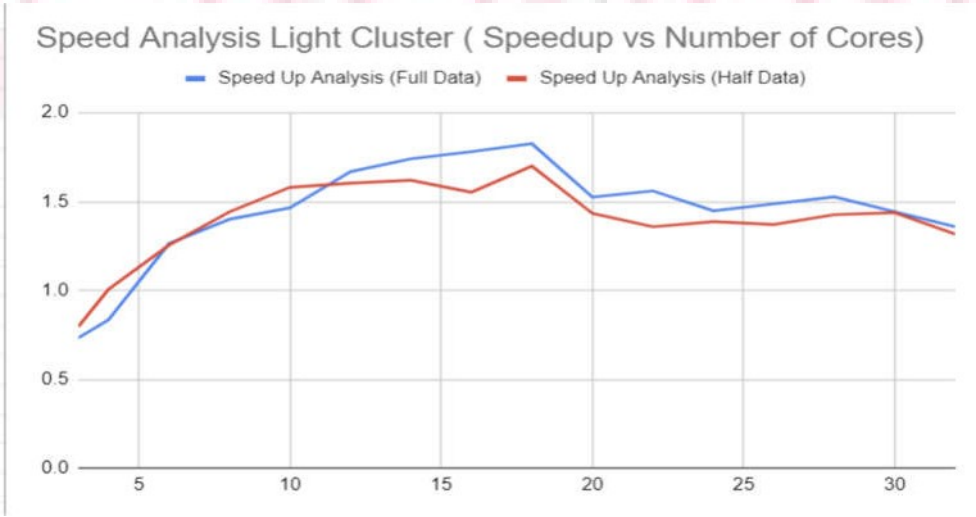
(Fat Cluster)

The performance trend starting on a fat cluster (2 VM) with 32 cores is displayed in the following graph. That is the proportion between the parallel algorithm's and serial algorithm's execution times.



(Light Cluster)

The performance trend starting on a Light cluster (16 VM) with 32 cores is displayed in the following graph. That is the proportion between the parallel algorithm's and serial algorithm's execution times.



Due to Open MPI's massive overhead and the fact that there is only two slaves performing the search, this is the case. On the other hand, as the number of slaves increases, you can observe how each new slave's speedup gradually improves.

We see that when the number of cores used for substring search rises (from 3 cores and higher), the speedup also increases till core 12 and then it becomes constant as the MPI communication overhead cancel out the increasing no of cores.

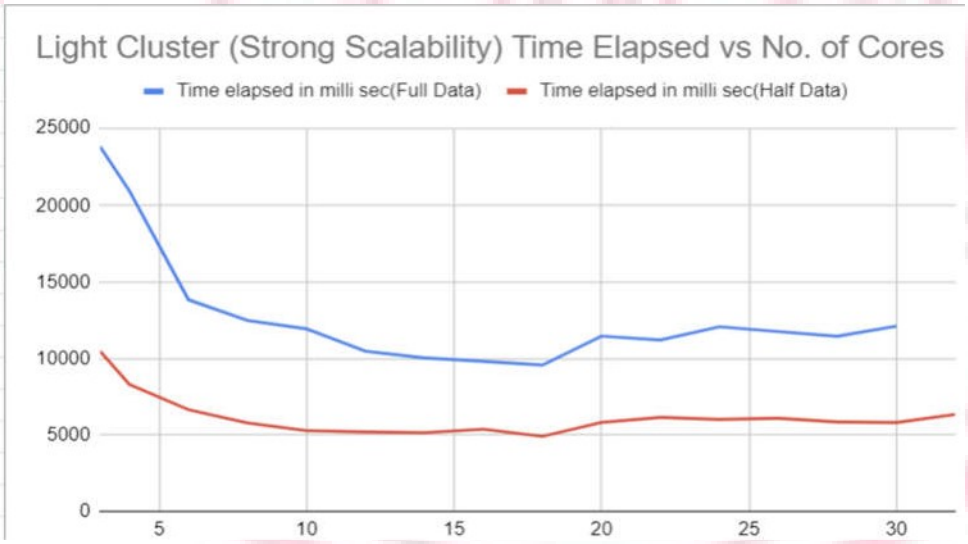
Scalability Analysis:

The findings of the scalability analysis are displayed in this section. Results produced utilizing the Google Cloud Platform cluster. 15 slaves, each with 2 cores, were operated by a single master. Both strong and weak scalability were studied.

Strong Scalability (Light Cluster):

Hypothesis : Regarding robust scalability, we anticipate that if the number of cores (machines / slaves) available doubles while the problem size remains constant, the execution time should be cut in half.

(Light Cluster -16 Machines & total 32 Cores)
Our data is effectively grouped by "number of cores" and the “time elapsed” is displayed.



The graphs for Full Human Genome (Data) and Half Human Genome Data, however, go against what we had anticipated. The presence of network communication overhead (by the computers in the cluster) and the fact that it is more significant when the relevant data chunks are smaller (Full Human Genome (Data) and Half Human Genome Data), respectively, can be used to explain this.

Weak Scalability (Fat Cluster):

Hypothesis: The execution time should remain constant if, for example, the size of the data set doubles, the number of cores (machines / slaves) doubles, and the sub-text analyzed by each slave stays the same size. This is because data for the weak scalability is always acquired on the cluster and launched by our automated test.

(Fat Cluster -2 Machines & total 32 Cores)

Our data is effectively grouped by "number of cores" and the “time elapsed” is displayed.

For this comparison, the sizes of the Full Human Genome and Half Human Genome databases were artificially constrained to 3267134034 and 1547079334 (twice as many) characters.

Conclusion:

1. Strong Scalability for the light cluster can be seen as we increase the number of cores and becomes almost constant after 12 and above.
2. Weak Scalability for the fat cluster can be seen as we increase the number of cores and becomes almost constant after 12 and above.
3. One can observe that the Fat cluster works much faster than the Light cluster as the MPI communication time is much less here.

Data references for the represented graphs above:

The below data are the data which we have noted down while running our multiple VMs in multiple situations.

The light cluster data:

Full Text Size	Pattern Size	No of Cores	Time elapsed in milli sec(Full Data)	Full Text Size	Pattern Size	Number of Cores	Time elapsed in milli sec(Half Data)
3267134035	1920	3	23888.4	1547079334	2520	3	10483.1
3267134036	1920	4	20962.4	1547079334	2520	4	8303.62
3267134038	1920	6	13847.4	1547079334	2520	6	6655.96
3267134040	1920	8	12495	1547079334	2520	8	5790.83
3267134042	1920	10	11944.4	1547079334	2520	10	5283.07
3267134044	1920	12	10482.5	1547079334	2520	12	5205.28
3267134046	1920	14	10046.2	1547079334	2520	14	5153.24
3267134048	1920	16	9819.84	1547079334	2520	16	5377.31
3267134050	1920	18	9580.62	1547079334	2520	18	4911.43
3267134052	1920	20	11468.3	1547079334	2520	20	5826.87
3267134054	1920	22	11216	1547079334	2520	22	6148.96
3267134056	1920	24	12082.4	1547079334	2520	24	6019.26
3267134058	1920	26	11764.3	1547079334	2520	26	6094.07
3267134060	1920	28	11459.8	1547079334	2520	28	5853.3
3267134062	1920	30	12125.5	1547079334	2520	30	5809.9
3267134064	1920	32	12887.1	1547079334	2520	32	6344.21
3267134034	1920	Serial	17545	1547079334	2520	serial	8373.27

Number of Cores	Speed Up Analysis (Full Data)	Speed Up Analysis (Half Data)
3	0.734456891	0.798739877
4	0.836974774	1.008387908
6	1.267024857	1.258010865
8	1.404161665	1.445953343
10	1.468889187	1.584925053
12	1.673741951	1.608610872
14	1.746431487	1.624855431
16	1.786688989	1.557148463
18	1.831301106	1.70485378
20	1.529869292	1.437009921
22	1.564283167	1.361737595
24	1.452112163	1.391079634
26	1.491376452	1.374002924
28	1.531004031	1.430521244
30	1.446950641	1.44120725
32	1.361438958	1.319828631

The fat cluster data:

Full Text Size	Pattern Size	No of Cores	Time elapsed in milli sec(Full	Full Text Size	Pattern Size	Number of Cores	Time elapsed in milli sec(Half D
3267134034	1920	3	23215.6	1547079334	2520	3	7916.27
3267134034	1920	4	17425	1547079334	2520	4	6139.11
3267134034	1920	6	12851.4	1547079334	2520	6	4822.18
3267134034	1920	8	10610.5	1547079334	2520	8	4199.73
3267134034	1920	10	9579.48	1547079334	2520	10	3972.78
3267134034	1920	12	11078.8	1547079334	2520	12	3847.7
3267134034	1920	14	8264.53	1547079334	2520	14	3646.42
3267134034	1920	16	8150.99	1547079334	2520	16	3621.28
3267134034	1920	18	8167.08	1547079334	2520	18	3943.83
3267134034	1920	20	7753.33	1547079334	2520	20	3857.6
3267134034	1920	22	7728.62	1547079334	2520	22	3796.11
3267134034	1920	24	7663.42	1547079334	2520	24	3739.63
3267134034	1920	26	7627.18	1547079334	2520	26	3717.45
3267134034	1920	28	7834.82	1547079334	2520	28	3627.45
3267134034	1920	30	7810.68	1547079334	2520	30	3696.48
3267134034	1920	32	10289.2	1547079334	2520	32	3687.04
3267134034	1920	1(Serial)	17641.5	1547079334		1(Serial)	8330.25

Number of Cores	Speed Up Analysis (Full Dat	Speed Up Analysis (Half Dat
3	0.759898517	1.052294831
4	1.012424677	1.356914927
6	1.372729819	1.727486324
8	1.662645493	1.983520369
10	1.841592654	2.096831438
12	1.592365599	2.164994672
14	2.134604146	2.284500963
16	2.16433832	2.300360646
18	2.160074347	2.112223397
20	2.275344916	2.159438511
22	2.282619666	2.194417443
24	2.302040081	2.227559946
26	2.31297806	2.240850583
28	2.251679043	2.296447918
30	2.258638172	2.253562849
32	1.714564786	2.25933269