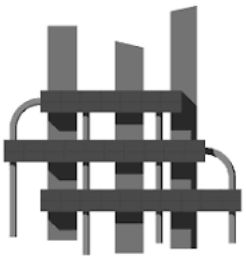


Advanced Computer Architecture - Report

Genomics and proteomics substring matching: a look at Rabin-Karp's algorithm

Vito Giacalone (546646)

April 22, 2024

UNIVERSITÀ DI PAVIA

Contents

The algorithm

Rabin-Karp's algorithm is an hash-based algorithm: for each substring of the main text, of the dimension of the pattern, the algorithm computes the hash of that substring and compare it with the hash of the pattern.

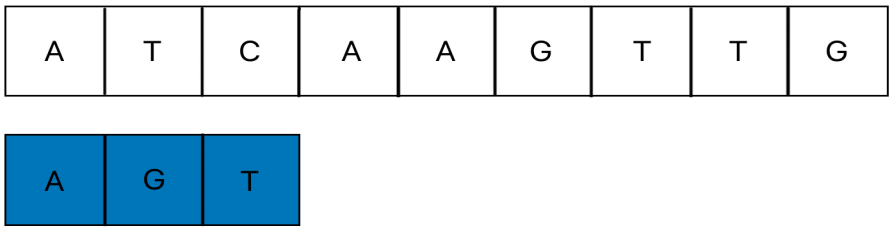


Figure 0.1: Check between pattern and text

If the two hashes coincide then we need to check character by character the two strings and if all the characters are the same, we have a match.

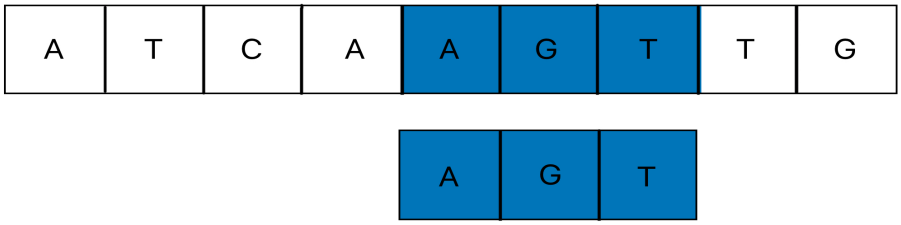


Figure 0.2: Pattern occurrence found in the text

Said n the length of the text and m the length of the pattern, the algorithm has a computational cost in the **best** case of $O(m+n)$ and in the **worst** case of $O(mn)$.
Checking character by character it's very very expensive in terms of computational cost then, to avoid this problem, the Rabin-Karp's algorithm suggests to introduce a **hash** function.
The *hash* function maps a string into a number, in this way only one comparison is required (instead of m) to understand if the substring and the pattern could coincide.
In this project, five different hashing function have been used because of their simplicity both in computational simplicity and algorithmic complexity.

- **polyHash**
- **sdbm**
- **djb2**
- **loselose**
- **rolling hash**

PolyHash is simply a polynomial hash computed considering each element of the string as the coefficient of a polynomial of degree $m-1$ and reducing it modulo $UINT\_MAX$.
sdbm is a simple algorithm created for a database library and it has the property to have a good distribution of the hash. This function is still used in Berkley DB (Oracle).
djb2 and **loselose** are not the best algorithm but they have been used for their extreme simplicity.
Rolling Hash Similar to polyhash. It's the better among the five hash functions, because the hash is not computed from scratch each time but is simply updated.
Said $s_0, s_1, \ldots, s_{m-1}$ the characters of a string **s** of length m the rolling hash is computed as follow:

$$o_i = s_i P^{(m-1)} + s_{i+1} P^{(m-2)} + \cdots + s_{i+m-2} P^{(1)} + s_{i+m-1}$$

o_i is the number that represents an m charcters string. P reprsents the prime number nearest to the size of the alphabet.

$$hash(o_i) = o_i \% M$$

. If we consider the shift of our string over a text, instead to compute the hash from scratch we can update it, "removing" the leftmost character and "adding" the rightmost character, and this is done according to the following equation:

$$o_{i+1} = (o_i - s_i P^{(m-1)})P + s_{i+m}$$

Performance of the serial algorithm and a-priori study of parallelism

To do an a-priori study of parallelism, we need to know which is the fraction of code that we want to parallelize.
An accurate analysis using the profiling software **gprof** gave us those results with an input file of $\approx 2.1GB$, a pattern of 8 characters and the rolling hash function:

%time	cumul. sec.	self sec.	calls	self s/call	tot. s/calls	name
95.97	23.87	22.91	1	22.91	22.91	rabin_karp2
4.02	23.87	0.96	1	0.96	0.96	readFile

index	%time	self	children	called	name
		23.87	0.00	1/1	main [2]
[1]	100.0	23.87	0.00	1	rabin_karp2 [1]
					<spontaneous>
[2]	100.0	0.00	23.87		main [2]
		22.91	0.00	1/1	rabin_karp2 [1]
		0.96	0.00	2/2	readFile [3]

As it's possible to see the program presents 2 principal functions:

- **readFile**: that reads the text file and the pattern file and stores each of them into an array.
- **rabin_karp2**: that computes the Rabin-Karp algorithm for substring search into the text.

To estimate the percentage of parallelizable code, the overall execution time has been measured and also the execution time of the routine that performs the algorithm. The ratio of the second one on the first one gives the percentage of code that we want to parallelize. According to some measurements that have been computed, the portion of code that can be parallelized amount to $\sim 96\%$. Since the IO operations are so expensive, the remaining $\sim 4\%$ is the time necessary to read the two text files.
The **Amdahl's Law** give us the theoretical speedup that can be reached if we adopt a parallelism strategy with n cores. The formula that describes the speedup is:

$$speedup = \frac{n}{n + p(1-n)}$$

fixed $p = 0.96\%$ the characteristic of the speedup is the following:

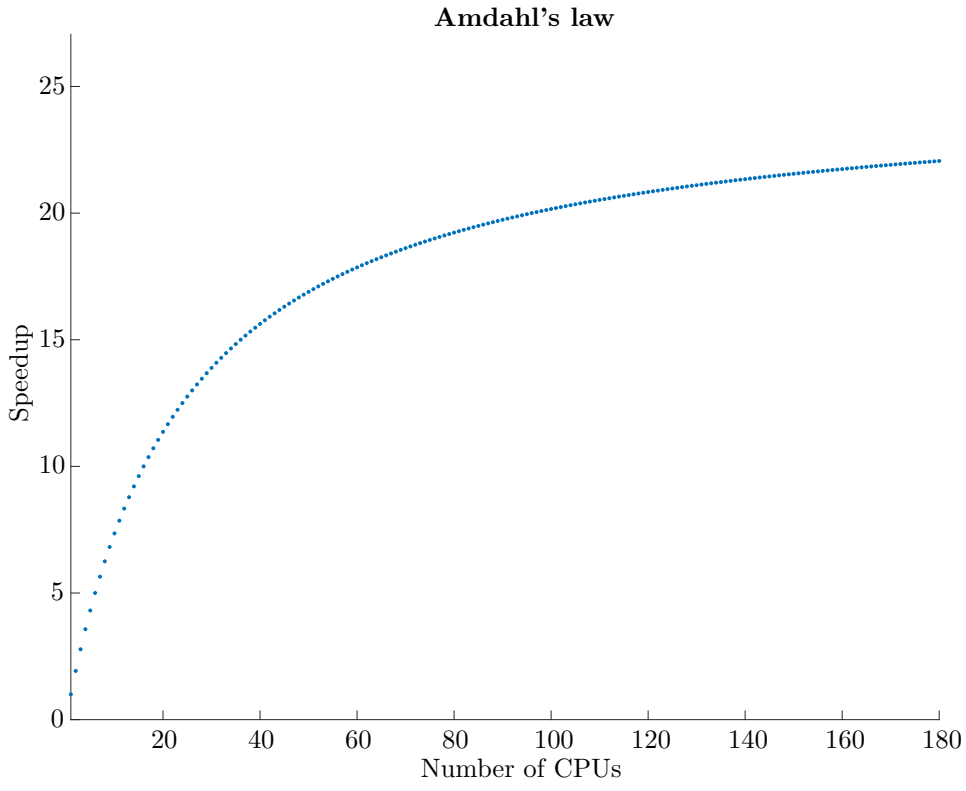


Figure 0.3: Amdahl's law

The strategies

The strategy adopted to parallelize the code consist in the splitting of the text among all the processes. Each slave process, except the last one, receives from the master an amount of characters that is given by:

$$\frac{text\ length}{number\ of\ processes} + pattern\ length - 1$$

because we need to take into account also some matches that could be overlapped among the portion of text assigned to two processes.
The last process receives a number of characters slightly greater than the others:

$$\frac{text\ length}{number\ of\ processes} + pattern\ length - 1 + text\ length\ \textbf{mod}\ \#cores$$

(This solution has been deployed on Google Cloud Platform)
A second approach (but not implemented for timing reasons) could be to perform a multiple pattern search either sharing the patterns to all processes and each process performs the search many times as the number of patterns, or split the text to some subgroups of cores and each subgroup performs the algorithm respect to a single pattern.

Communication and synchronization strategy

This program is based on a *divide and conquer* approach. The master process reads the text and the pattern and sends to all the processes the pattern and the chunk that each process must analyze. Each slave runs the Rabin-Karp algorithm and the master process collect all the occurrences that each process found.
Another aspect that has been taken in considerations is the activation/inactivation of each process. If we are dealing with a large pattern and a text which dimension is comparable to the pattern, the number of active cores (**executors**) is decreased because the chunk that each process receive from the master is smaller than the pattern to be found. A flag is sent to each slave from the master to let them know if they must be active or inactive.
This case is quite rare in genomics applications because the average dimensions of the genes are about of $(60-70)K$ characters and each chromosome has an average dimension of 10^9 characters.

Parallel implementation with Open MPI

Before performing the algorithm, each process must have some important data. The master process provides those informations to all the other process. Using **MPI_Bcast()** all the processes receive the number of executors cores, the length of the text and the length of the pattern. With **MPI_Scatter()** the vector of flags is scattered among the processes and each process will know if is active or not.

```
1  //Sending to all the processes the number of executors cpus
2      MPI_Bcast(&executors, 1, MPI_INT, 0, MPI_COMM_WORLD);
3
4  //Sending to all the processes active/inactive flag
5      MPI_Scatter(flag, 1, MPI_INT, &isActive, 1, MPI_INT, 0, MPI_COMM_WORLD);
6
7  //Sending to all the processes the total length of the text
8      MPI_Bcast(&txtlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
9
10 //Sending to all the processes the length of the pattern
11     MPI_Bcast(&patlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
```

Figure 0.4: Preliminary variables sending

The number of active cores is simply computed using a specific function **who_is_active()**.
The splitting of the text is computed by the master process using the function **split_dataset()** that splits the initial text into chunks and sends, using **MPI_Send()**, each chunk to the corresponding process.
On the slave side, the function **receive_dataset()** that embeds **MPI_Recv()**, is invoked to receive the chunk sent from the master.
After the computation of the Rabin-Karp algorithm to find the occurrences in the text, the master collects the results from all the other processes using **MPI_Reduce**.

Performance and scalability analysis on GCP

Many many test have been done on GCP's Computer engine. The performance have been measured using the function **clock_t()**. The function has been placed at the beginning of the source code and at the end.

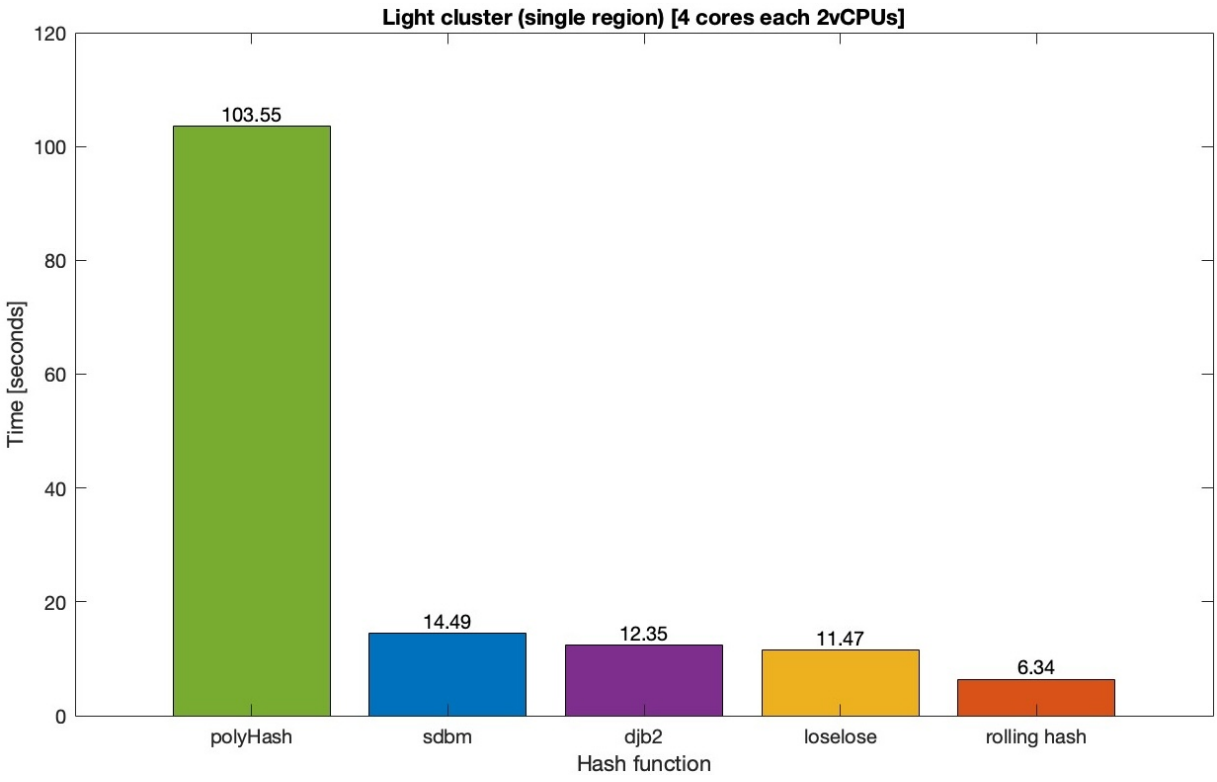
Dividing **end - start** by the system variable **CLOCKS_PER_SEC** we have an estimate of the execution time. Repeating the measurement several times we have a realiable estimate of the execution time.
The analysis takes into account the execution only over N2 virtual machines offered by GCP's marketplace.

Stato	Nome	Zona	Suggerimenti	Utilizzato da	IP interno	IP esterno
✓	master1	europe-west1-b			10.132.0.4 (nic0)	34.76.73.155 (nic0)
✓	slave1	europe-west1-b			10.132.0.5 (nic0)	34.77.91.221 (nic0)
✓	slave10	europe-central2-a			10.186.0.4 (nic0)	34.118.121.6 (nic0)
✓	slave11	europe-central2-a			10.186.0.5 (nic0)	34.116.223.16 (nic0)
✓	slave12	europe-west4-a			10.164.0.2 (nic0)	34.90.102.228 (nic0)
✓	slave13	europe-west4-a			10.164.0.3 (nic0)	35.204.175.255 (nic0)
✓	slave14	europe-west4-a			10.164.0.4 (nic0)	35.204.174.49 (nic0)
✓	slave15	europe-west4-a			10.164.0.5 (nic0)	34.90.94.194 (nic0)
✓	slave2	europe-west1-b			10.132.0.6 (nic0)	34.79.129.137 (nic0)
✓	slave3	europe-west1-b			10.132.0.7 (nic0)	34.140.57.167 (nic0)
✓	slave4	europe-west10-a			10.214.0.2 (nic0)	34.32.37.184 (nic0)
✓	slave5	europe-west10-a			10.214.0.3 (nic0)	34.32.31.146 (nic0)
✓	slave6	europe-west10-a			10.214.0.4 (nic0)	34.32.35.111 (nic0)
✓	slave7	europe-west10-a			10.214.0.5 (nic0)	34.32.5.81 (nic0)
✓	slave8	europe-central2-a			10.186.0.2 (nic0)	34.116.223.129 (nic0)
✓	slave9	europe-central2-a			10.186.0.3 (nic0)	34.118.81.38 (nic0)

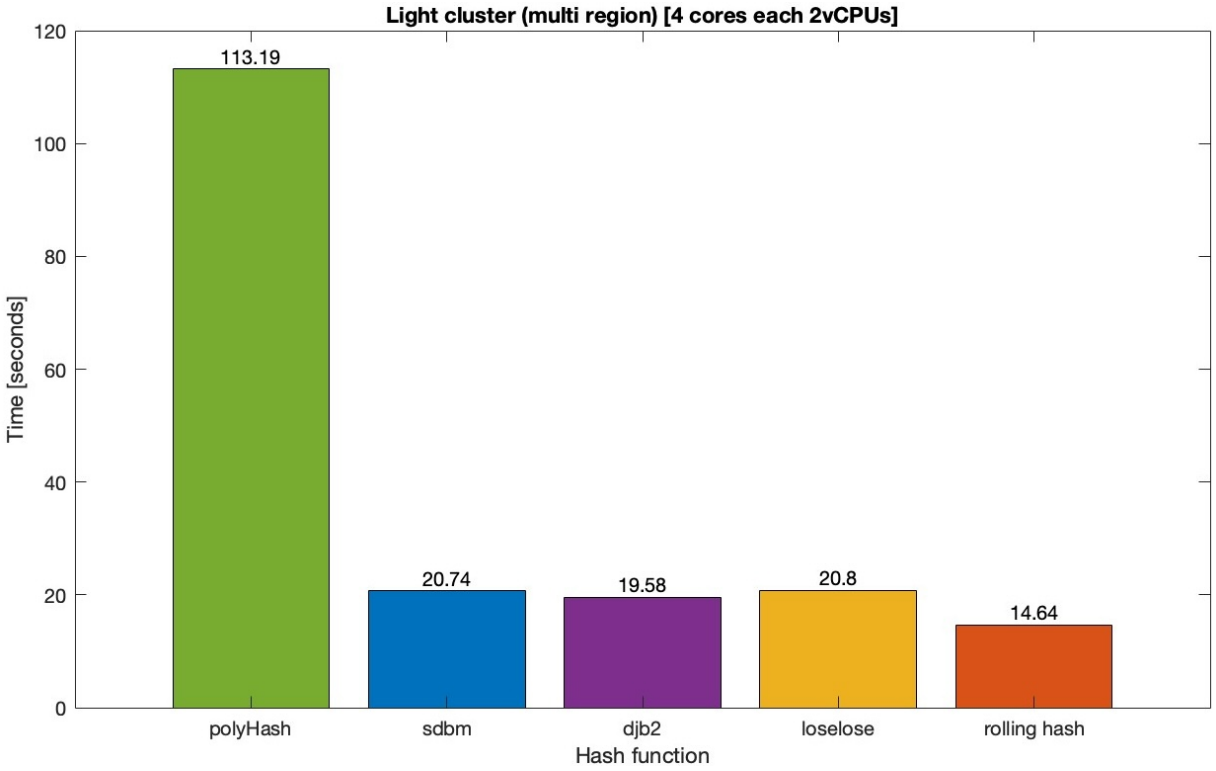
Figure 0.5: GCP virtual machines setup

The tests have been done with a file to analyze of $\sim 2.1GB$ and a pattern of 8 characters "$TAACCCTA$"
The various configurations used to do the tests are:

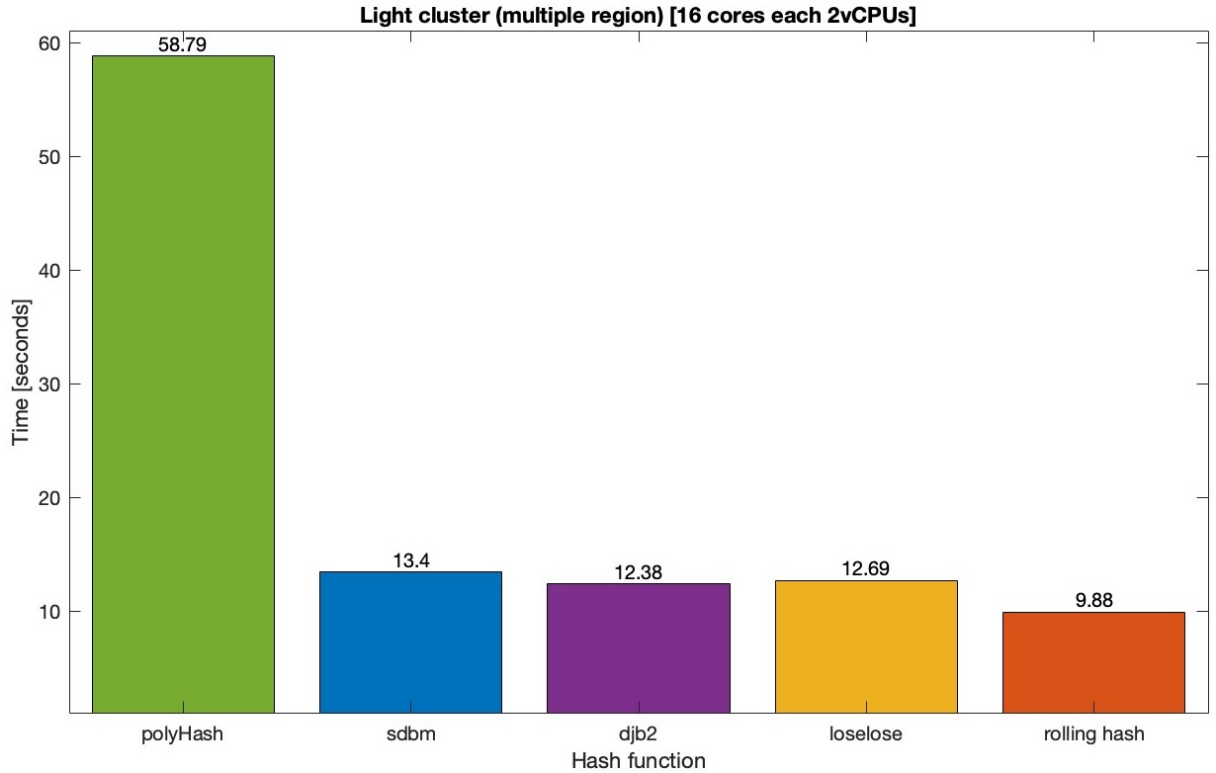
- Light cluster single region (europe-west1-b) 4 cores each with 2 vCPUs (1 core/vm)



- Light cluster multiregion (europe-west1-b, europe-central2-a, europe-west4-a, europe-west10-a), 4 cores each with 2 vCPUs (each core in different region) (1 core/vm)

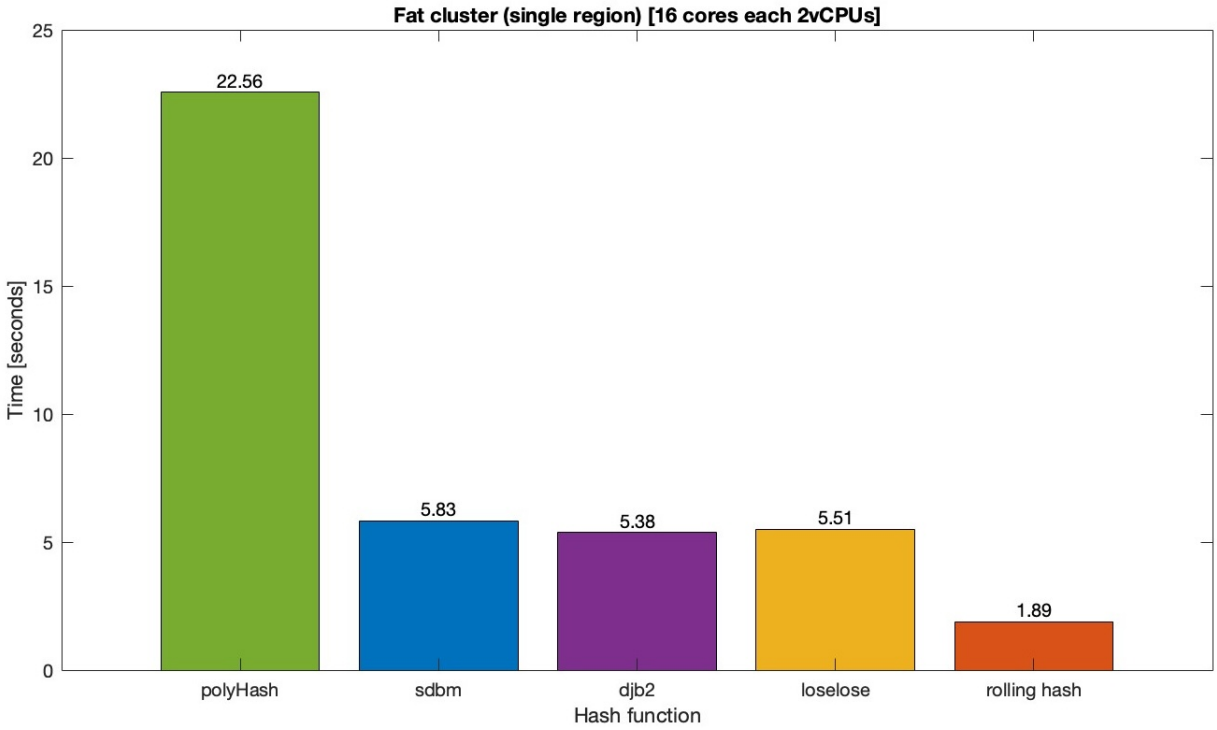


- Light cluster multiregion (europe-west1-b, europe-central2-a, europe-west4-a, europe-west10-a) 16 cores each 2 vCPUs (1 core/vm)



Since the number of core into a single region is fixed to 8, using the form available on "quotas" section, the number of cores has been increased to 32.

- Fat cluster single region (europe-west1-b) 16 cores each 2 vCPUs (16 cores/vm)



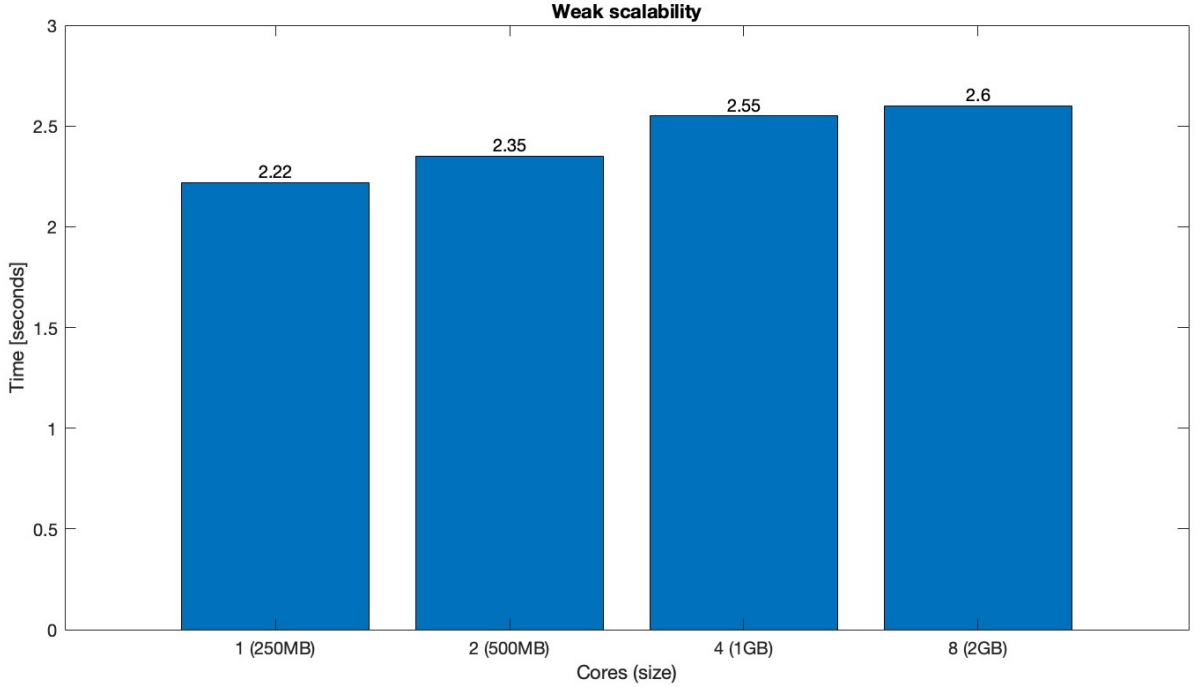
The serial program runs in 23.87 seconds, the best performance on parallel algorithms is of 1.89 seconds. We can easily estimate that the speedup is of ~ 12.62 almost similar to the theoretical speedup proposed by the Amdahl's law.

Scalability

Other tests have been done on the Fat cluster to test the weak scalability and the strong scalability and these are the results:

- Weak scalability

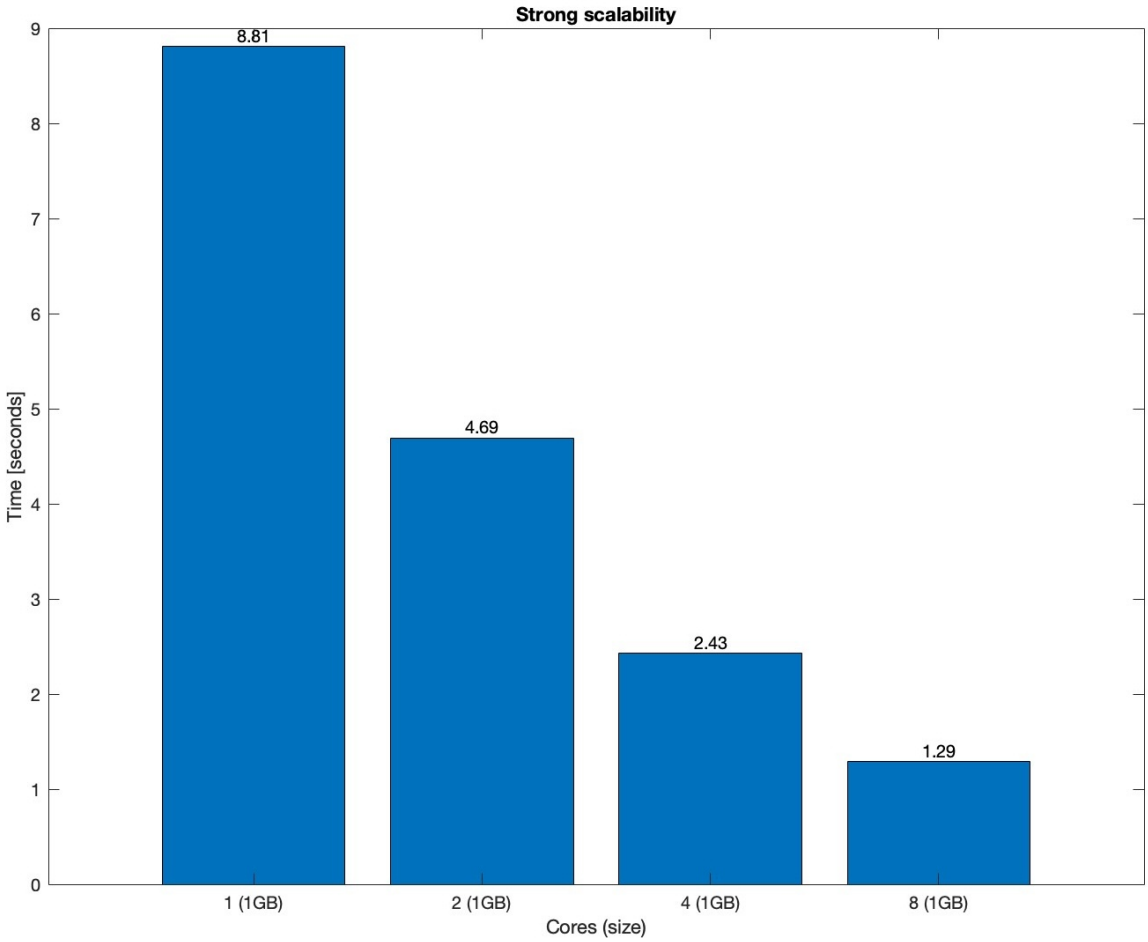
Text size	# cores	Execution time
250MB	1	~ 2.22
500MB	2	~ 2.35
1GB	4	~ 2.55
2GB	8	~ 2.60



As it's possible to see, the performance of the program are not too many different if the input and the number of cores is doubled, but there's a slightly increasing trend, so there isn't weak scalability.

- Strong scalability

Text size	# cores	Execution time
1GB	1	~ 8.81
1GB	2	~ 4.69
1GB	4	~ 2.43
1GB	8	~ 1.29



Keeping fixed the dimension of the problem and changing the number of cores, the results show a great improvement on the execution time. There's strong scalability.

[EXTRA] Effect of the hash function on the performance

Let's consider the case of the light cluster multiregion. Keeping fixed the hash function and changing the number of processors we can see that the hash function plays a key role in the performance of our algorithm:

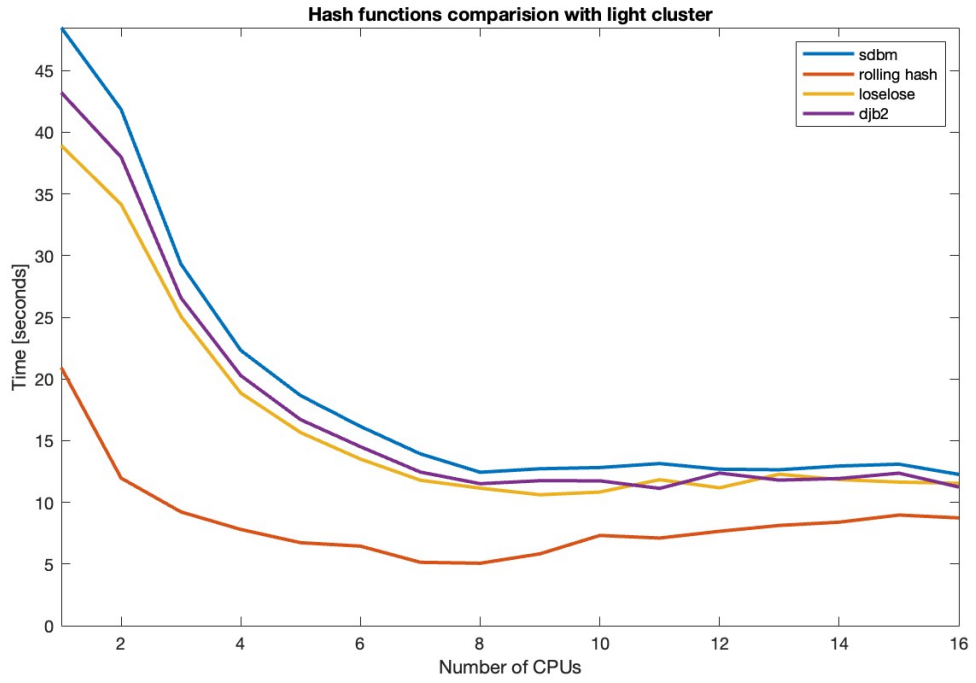


Figure 0.6: Hash functions performance comparison

This plot shows, as explained also by the measurements, that the best hash function is the **rolling hashing function** (red). The **loselose** (yellow) and the **djb2** (purple) are more or less the same, on average. According to the measurements the **djb2** is slightly better than the **loselose**. The **sdbm** seems to be the worst.

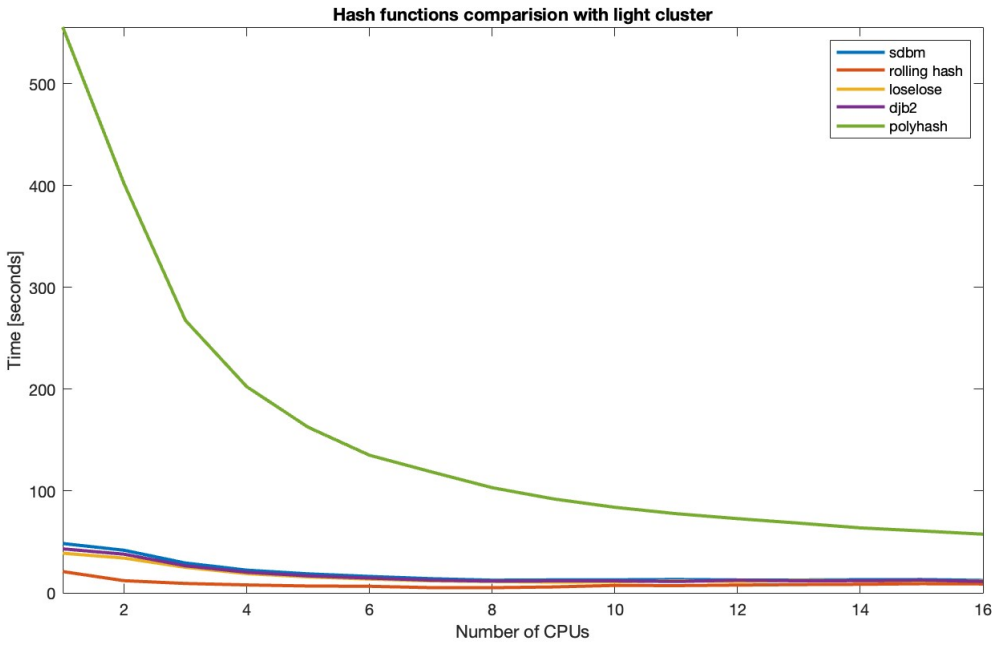


Figure 0.7: Hash functions performance comparison

But if we consider also the **polyHash** function, there's a disastrous decay of the performance. Without any doubt **polyHash** is the worst hash because of its computational cost.

Individual contribution

- Vito Giacalone (546646): worked on writing serial and parallel code and libraries. Took care of the report using LaTeX and made the measurements on Google Cloud Platform to produce the graphic component of this document.

References

1. http://www.cse.yorku.ca/~oz/hash.html
2. R. Sedgewick and K. Wayne, Algorithms. Boston: Addison-Wesley, 2016.
3. https://www.codingninjas.com/studio/library/rabin-karp-algorithm