UNIVERSITÀ DI PAVIA

Advanced Computer Architecture - Project Report

Substring Matching with application to Genomics

Diego Mastella (525163) - **Matteo Ragni** (520038)
University of Pavia - Master's Degree in Computer Engineering

1 Serial algorithm analysis

The Algorithm The substring search algorithm used in this project is the `Rabin-Karp fingerprint search`. It is based on hashing: for each possible M-character substring of the text, we compute a hash function for the pattern and use that function to search for matches. If a text substring is found that has the same hash value as the pattern, we can search for a match. In practice, a linear-time substring search `O(M+N)` is achieved thanks to the work of Rabin and Karp, who showed how it is easy to compute hash functions for M-character substrings in constant time. [1]

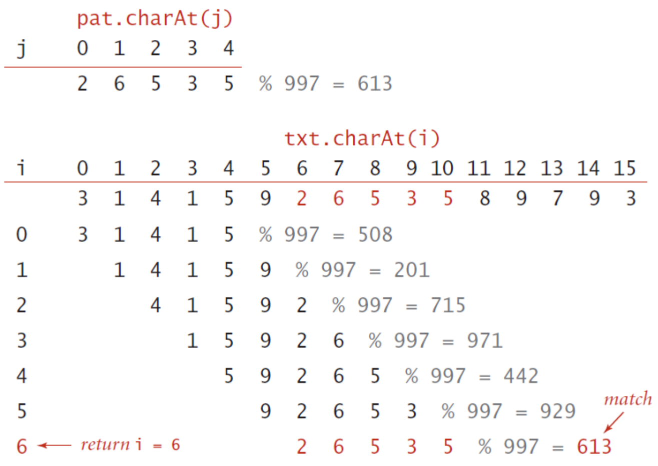


Figure 1: **Example** - Rabin-Karp substring search [1]

Rolling hashing The Rabin-Karp method is based on efficiently computing the hash function for position `i+1` in the text, given its value for position `i`. As a result of not having to repeatedly iterate through each character to add their numerical values, time complexity is reduced. The following equations explain how hash function works:

$$x_i = t_i D^{M-1} + t_{i+1} D^{M-2} + \cdots + t_{i+M-1} \tag{1}$$

where x_i is the number corresponding to an M-character substring, t_i represent the character in the i^{th} position of the `txt` substring and D is the alphabet's dimension. We know that $hash(x_i) = x_i \;\%\; Q$. Shifting one position right in the text corresponds to replacing x_i by:

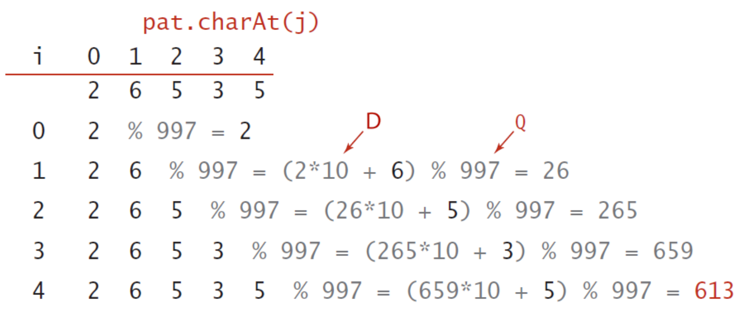
$$x_{i+1} = (x_i - t_i D^{M-1})D + t_{i+M} \tag{2}$$

We subtract off the leading digit, multiply by D, then add the trailing digit. The result is that we can effectively move right one position in the text in constant time, whether M is 5 or 100 or 1,000.

Hash collisions When we find a match, we must ensure that it is a true match rather than a hash collision. We can make the hash table size Q as large as we want, resulting in an extremely low probability that a random key hashes to the same value as our pattern. As a result, when we find a match, we perform a char-by-char comparison between the `pat` string and the considered `txt` substring to ensure correctness.

Parameter	Meaning
txt	Text string
pat	Pattern string
N	Text length
M	Pattern length
D	Alphabet's dimension (256)
$hash(x)$	Hash function
t_i	i^{th} character of txt
Q	Prime number

Table 1: Cited parameters



```
                pat.charAt(j)
 i    0  1  2  3  4
      2  6  5  3  5
 0    2  % 997 = 2                          D          Q
 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
```

Figure 2: **Example** - Computing hash value for the pattern [1]

2 A-priori study of parallelism

Serial program's functionalities It has two major routines:

- `read_txt`: the one that reads the `txt` file and saves it into an array;
- `rabin_karp`: the one that develops Rabin-Karp substring search in the given text.

The result of the `gprof` profiling tool is following.

```
                                                           - txt file dimension = 2 GB
                                                           - searched pattern = "TAAACC"
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     15.17    15.17        1    15.17    15.17  rabin_karp
  0.00     15.17     0.00        1     0.00     0.00  read_file

      Call graph


granularity: each sample hit covers 4 byte(s) for 0.07% of 15.17 seconds

index % time    self  children    called     name
               15.17    0.00       1/1           main [2]
[1]    100.0   15.17    0.00       1         rabin_karp [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   15.17                 main [2]
               15.17    0.00       1/1           rabin_karp [1]
                0.00    0.00       1/1           read_file [3]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[3]      0.0    0.00    0.00       1         read_file [3]
-----------------------------------------------

Index by function name

   [1] rabin_karp              [3] read_file
```

Listing 1: analysis.txt

Different approaches We thought of a few possible approaches:

- The strategy we used to create this algorithm in parallel fashion is the following: divide the `txt` string among several cores[1], allowing each core to look for a specific pattern within the text string's allocated part. Every process prints its results on the `stdout` file. This may cause some synchronization problem:

[1] *Note:* Since all the processes can access the same file pointer, no text is sent by the master process to the slaves. Instead, all the slave processes receive the point from which to start reading the file.

it can happen that two processes attempt to print simultaneously, causing the overlapping of some strings on the video. However it does not happen so frequently: using a 2 `GB` file and a very short pattern of six characters with almost half a million mathes it happens less than 50 times.

- We implemented a new solution by having every process to create a file in which to store the results, solving the previous problem. After all the cores have completed the execution the master process merges the files in a unique output file. This solution has not been deployed on `GCP`.
- Another approach *(not implemented)* would be to assign a single core to a single pattern (in the case of a multi-pattern search) and then search in the `txt` string for that specific pattern. This method does not effectively use *MPI functions* because it does not require core-to-core communication given that each core would independently count its own pattern within the provided main string.

Communication and synchornization strategy The program employs a divide-and-conquer approach, where the master process is responsible for computing the point from which to start reading the file dividing the input `txt` into smaller chunks. Each worker process then independently performs the Rabin-Karp algorithm on its assigned chunk and sends the results back to the master process. When the text cannot be divided into equal parts, the master process also takes care of the remaining part. These are the data sent by the master process to the other processes:

- `send`: an array containing `bufsize`, that is the length of text each core has to analyze, `M` that is length of the `pat`, and the rest of $\frac{\texttt{N}}{\texttt{bufsize}}$ where `N` is the overall length of the `txt` string and string;
- `pat`: the actual string to be searched into the `txt` file;

When `M > bufsize + rest + 1`, the number of cores is decreased because otherwise the chunk of `txt` given to each core is smaller than the pattern to be found. This solution might seem inadequate, but since in genomics the average length of a gene string is about 65000 characters and that the human genome's lenght is of several billion, you will very hardly happen upon this case. Therefore a flag variable is sent to all the processes in order to let them know if they have to be turned off.

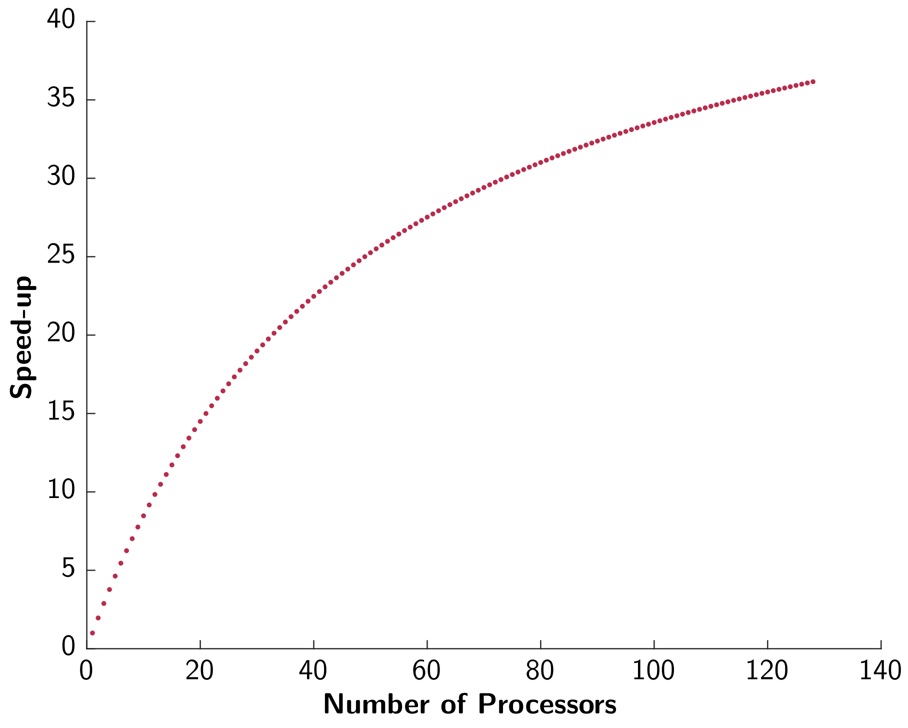


Figure 3: Amdahl's Law

Theoretical assessment By measuring the performance of the parallel system with the Rabin-Karp algorithm, the predictions of Amdahl's law can be verified. The law states that the maximum improvement to the overall system performance is limited by the fraction of the workload that can be parallelized. In the case of the Rabin-Karp algorithm, we can estimate that almost 98% of the algorithm is parallelizable. In the remaining 2% we include the computation of the actual number of cores to be used and the calculation of the sum of the frequencies received by the master process. The formula that describes the speedup model in Fig.3 is:

$$speedup = \frac{n}{n + p(1-n)} \tag{3}$$

where n is the number of processors and p is the fraction of code that can be parallelized.

3 MPI parallel implementation

Communication between cores Of course, the master process is in charge of opening the communication between processes: before carrying out the search in its text area, it sends the necessary information to all the others to allow them to begin the execution. This was accomplished through the use of the MPI_Scatter and MPI_Bcast functions. Using the first one the master process sends to all the slaves the respective flag (equal to 1 if it has to performe the searching, otherwise it will be equal to 0). Through the second one, process 0 sends first the send array and then the pat string.

```
//Sending preliminary variables in send array
    MPI_Bcast(send, 3, MPI_INT, 0, MPI_COMM_WORLD);

//Sending the pattern
    MPI_Bcast(pat,pat_len,MPI_CHAR,0,MPI_COMM_WORLD);

//Sending the flag variable
    MPI_Scatter(flag, 1, MPI_INT, &flag_send, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 2: Sending Functions

File management In order to read the txt file, we chose an approach that would allow us to achieve a good compromise between memory usage and execution speed, given that the file is very large. The file pointer is unique so that all cores could open the file through the MPI_File_open function. Then each process, once the file is opened, moves to the indicated location through the MPI_File_set_view function and subsequently allocate a vector containing only the chunk of txt that has been read thanks to MPI_File_read function.

```
    MPI_File_open(MPI_COMM_SELF, text_name, MPI_MODE_RDONLY,MPI_INFO_NULL, &file_in);
```

Listing 3: File opening - substringSearch_MPI.c

```
    MPI_File_set_view(file, position, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);

    MPI_File_read(file, seq, bufsize+M-1, MPI_CHAR, &status);
```

Listing 4: File reading - read.h

Frequency computation In order to compute the number of hits of the pattern inside the text, each process, once the searching is finished, sends his frequency to the master process by using the MPI_Gather function.

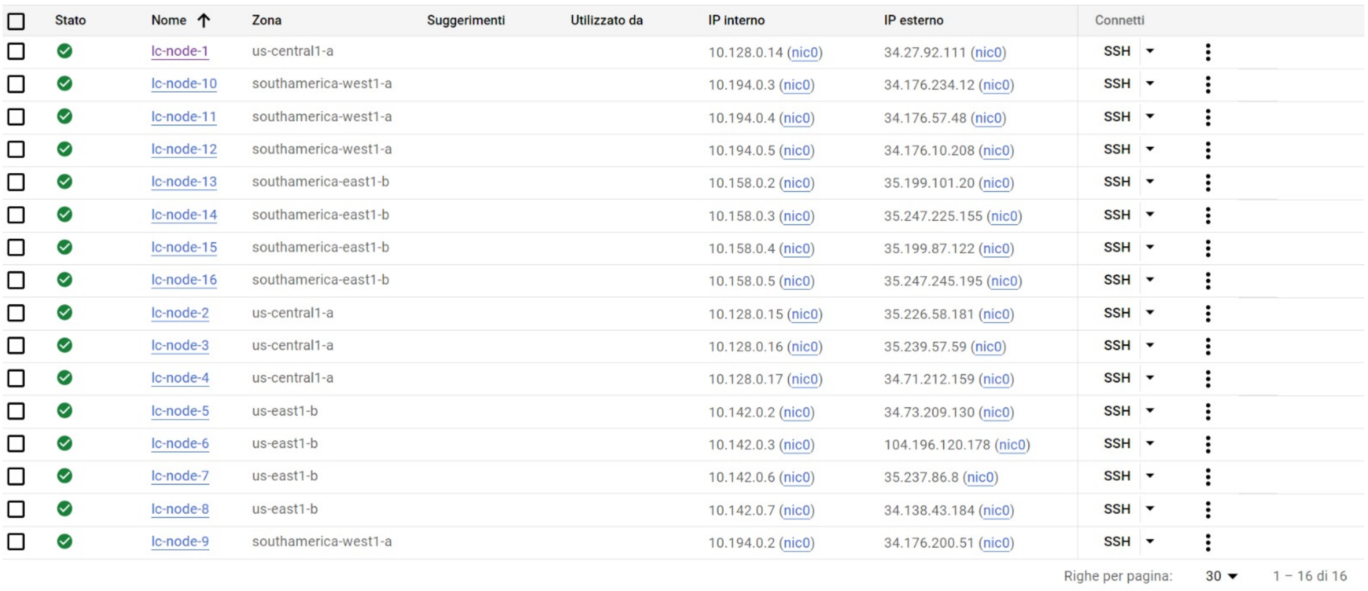
```
MPI_Gather(&freq, 1, MPI_INT, rec_freq, 1, MPI_INT, 0, MPI_COMM_WORLD);
if(myrank == 0){
    int total_freq = 0;
    for(int i = 0; i<size;i++){
      total_freq = total_freq + rec_freq[i];
    }
    printf("PATTERN FOUND %d TIMES\n",total_freq);
```

Listing 5: Frequency computation

4 Performance and scalibility analysis

GCP deployment Several tests have been done. The execution time has been calculated using the function `clock()` at the begin and at the end of the source code and then divided their difference by the constant `CLOCKS_PER_SEC`. It may not be the actual execution time, but at least a fair measurement among the several tests. Let's start from strong scaling analisys (fixed size: 2 GB text file - 6 chars pattern):

1. Light cluster, multi-region, 16 cores each 2 vCPUs (N2 series).



☐	Stato	Nome ↑	Zona	Suggerimenti	Utilizzato da	IP interno	IP esterno	Connetti
☐	✓	lc-node-1	us-central1-a			10.128.0.14 (nic0)	34.27.92.111 (nic0)	SSH
☐	✓	lc-node-10	southamerica-west1-a			10.194.0.3 (nic0)	34.176.234.12 (nic0)	SSH
☐	✓	lc-node-11	southamerica-west1-a			10.194.0.4 (nic0)	34.176.57.48 (nic0)	SSH
☐	✓	lc-node-12	southamerica-west1-a			10.194.0.5 (nic0)	34.176.10.208 (nic0)	SSH
☐	✓	lc-node-13	southamerica-east1-b			10.158.0.2 (nic0)	35.199.101.20 (nic0)	SSH
☐	✓	lc-node-14	southamerica-east1-b			10.158.0.3 (nic0)	35.247.225.155 (nic0)	SSH
☐	✓	lc-node-15	southamerica-east1-b			10.158.0.4 (nic0)	35.199.87.122 (nic0)	SSH
☐	✓	lc-node-16	southamerica-east1-b			10.158.0.5 (nic0)	35.247.245.195 (nic0)	SSH
☐	✓	lc-node-2	us-central1-a			10.128.0.15 (nic0)	35.226.58.181 (nic0)	SSH
☐	✓	lc-node-3	us-central1-a			10.128.0.16 (nic0)	35.239.57.59 (nic0)	SSH
☐	✓	lc-node-4	us-central1-a			10.128.0.17 (nic0)	34.71.212.159 (nic0)	SSH
☐	✓	lc-node-5	us-east1-b			10.142.0.2 (nic0)	34.73.209.130 (nic0)	SSH
☐	✓	lc-node-6	us-east1-b			10.142.0.3 (nic0)	104.196.120.178 (nic0)	SSH
☐	✓	lc-node-7	us-east1-b			10.142.0.6 (nic0)	35.237.86.8 (nic0)	SSH
☐	✓	lc-node-8	us-east1-b			10.142.0.7 (nic0)	34.138.43.184 (nic0)	SSH
☐	✓	lc-node-9	southamerica-west1-a			10.194.0.2 (nic0)	34.176.200.51 (nic0)	SSH

Righe per pagina: 30 1 – 16 di 16

Figure 4: **GCP** - Light Cluster -

N. of cores	Execution time (sec.)
1	17.96
2	10.97
3	10.00
4	6.96
5	5.97
6	4.84
7	4.23
8	3.68

Table 2: Ligth cluster (1/4)

N. of cores	Execution time (sec.)
9	3.36
10	3.06
11	2.76
12	2.56
13	2.32
14	1.77
15	1.61
16	1.50

Table 3: Ligth cluster (2/4)

N. of cores	Execution time (sec.)
17	1.89
18	1.79
19	1.64
20	1.56
21	1.56
22	1.50
23	1.38
24	1.34

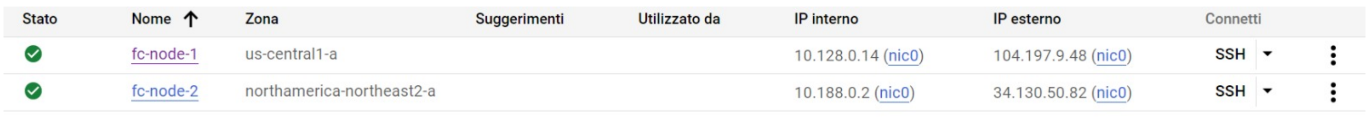
Table 4: Ligth cluster (3/4)

N. of cores	Execution time (sec.)
25	1.47
26	1.68
27	1.33
28	1.35
29	1.33
30	1.37
31	1.35
32	1.23

Table 5: Ligth cluster (4/4)

2. Light cluster, single-region (us-central1-a), 4 cores each 2 vCPUs (N2 series). Execution time $\sim$ 3.15 sec.
3. Light cluster, 4 cores each 2 vCPUs (N2 series) each core in one different region. Execution time $\sim$ 3.50 sec.

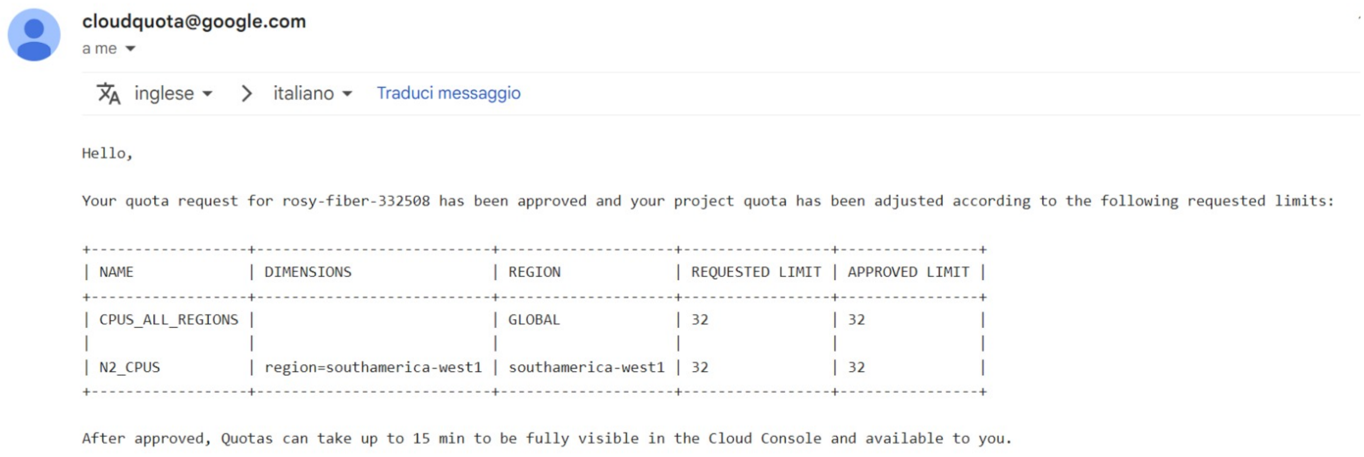
4. Fat cluster, multi-region, 2 cores each 16 vCPUs (N2 series). Execution time $\sim$ 1.35 sec.



Stato	Nome ↑	Zona	Suggerimenti	Utilizzato da	IP interno	IP esterno	Connetti
✓	fc-node-1	us-central1-a			10.128.0.14 (nic0)	104.197.9.48 (nic0)	SSH
✓	fc-node-2	northamerica-northeast2-a			10.188.0.2 (nic0)	34.130.50.82 (nic0)	SSH

Figure 5: **GCP** - Fat Cluster -

5. Fat cluster, single-region (southamerica-west1), 2 cores each 16 vCPUs (N2 series). Execution time $\sim$ 1.35 sec. Since it was not allowed to create more than 8 N2 series vCPUs we asked Google to inscrese the limit to 32.



cloudquota@google.com
a me

inglese > italiano Traduci messaggio

```
Hello,

Your quota request for rosy-fiber-332508 has been approved and your project quota has been adjusted according to the following requested limits:

+----------------+------------------------+-----------------+---------------+---------------+
| NAME           | DIMENSIONS             | REGION          | REQUESTED LIMIT | APPROVED LIMIT |
+----------------+------------------------+-----------------+---------------+---------------+
| CPUS_ALL_REGIONS |                      | GLOBAL          | 32            | 32            |
|                |                        |                 |               |               |
| N2_CPUS        | region=southamerica-west1 | southamerica-west1 | 32      | 32            |
+----------------+------------------------+-----------------+---------------+---------------+

After approved, Quotas can take up to 15 min to be fully visible in the Cloud Console and available to you.
```

Figure 6: **GCP** - Increased quotas -

Given that the serial program run with one vCPU takes almost 18 seconds and the fastest VM configuration takes 1.23 seconds, we can estimate that running a parallel program produce a speed-up of $\sim$ 14.6, almost as the theoretical speed-up of the Amdahl's Law.
What about weak scaling? We performed a few tests with different text file without changing the setup of the VMs instances (fat cluster, 2 cores each 16 vCPUs (N2 series)). The pattern lenght is always set to six in order to study the worst case, however we also performed substring search with pattern dimension $\sim$ 50 KB (much more actual in the genomic field) resulting in an execution time .

Text size	**Execution time (sec.)**
1 KB	0.01
100 KB	0.02
1 MB	0.02
2 GB	1.35

Table 6: Weak scalability

5 Individual contribution

To carry out this project we decided to contribute together as much as possible to avoid imbalances during implementation. We both worked on writing serial and parallel code. Then we decided to split the remaining work according to our own knowledge:

- *Diego Mastella* was in charge of managing the repository on GitHub during the developement of the code.
- *Matteo Ragni* took care of the report and the presentation using LaTeX;

References

[1] R. Sedgewick and K. Wayne, *Algorithms*. Boston: Addison-Wesley, 2016.