**21/12/2020**

**Substring matching with application to genomics/proteomics**

**Report by:**

**Mladen Jurkic – Chiara Balduzzi**

**Introduction:**

The aim of the developed algorithm is to find a group of characters (called ‘*pattern’*) among a larger set (indicated as *‘sequence’*). Both sets are only composed out of 4 elementary characters which refer to a nitrogenous base that could be found inside a nucleic acid chain (i.e.: DNA): adenine (A), thymine (T), guanosine (G), cytosine (C).

In our program, the comparison between *pattern* and *sequence* can be made in both forward and backward directions. Assuming the length of the sequence and of the pattern to be, respectively, *‘slen’* and *‘plen’*, we can have a total or partial overlapping. In this latter case, a match is found when the sub-sequence and pattern are equal up to a limited number of contiguous errors (denoted as *‘holes’* – at most 10% of *plen*) having a maximum allowed width, which in our implementations is set to 0.1% of the length *plen*. These two percentages come from a trial-and-error kind of approach and seemed reasonable to us.  
In either case it is possible to get the positions of the sequence at which the correspondence was found. Obviously, by adding the possibility to take holes into account we increase complexity and overall computational time of the program.

A first idea to solve the given problem, could be to use the built-in C function strcmp: int strcmp(const char \*sub\_sequence, const char \*pattern).  
This function can compare the *pattern* with a sub-string of the *sequence,* both having the same length *‘plen’*. By selecting, with proper indices, the sub-string of the *sequence* and ending the loop when we reach ‘*slen-plen’*, it’s possible to obtain the number of perfect correspondences. However, we abandon this solution favouring a “manual” approach for two main reasons. First, by using the built-in function we cannot appreciate the effect of the parallelization as the function is already optimized and very fast. Furthermore, the need to include partial overlaps makes this solution not useful for us.

To implement the ‘perfect-match’ solution we had to compare single characters of the sequence, one at a time: in case of a match of a character, we proceed in the comparison of the successive ones and obtain a correspondence if all of them match. Otherwise, we simply increment a counter variable, shifting the beginning of the sequence and de facto comparing the pattern with a new sub-sequence.  
The same reasoning lies behind the partial-match solution, where in addition we have an error counter to keep track of the holes. The comparison is abandoned if the number of holes or their width is greater than a given threshold.

**Preliminary analysis of the algorithm:**

In the serial code development, the first idea was that better performances could be obtained by writing a single function that is able to scan the sequence in a forward direction (called *‘search\_fwd’*). The backward scanning can simply be obtained reversing the pattern and inspecting the sequence forward once more. This solution is presented in the file *‘serial\_v1.c’* and has the advantage of re-utilizing the same code. What is more, the pattern reversal time is practically irrelevant, even for large patterns. However, having in mind that our goal is to implement a parallel version of the code, we provided another version (*‘serial\_v2.c’*) in which a single function sweeps the sequence both in forward and backward direction. Our hope was to be able to better parallelize this version by having some threads working to scan in one direction and some others in the opposite direction.  
The common features of the two versions are the main function where we ask the user to insert the desired length of both sequence and pattern and the *‘generate’* function that randomly generates the two sequences. We also provided a version of the codes where we do not generate the sequences but read them from file instead, if the user wishes to do so (same names as before with *‘\_file’* suffix).

Running the codes, we can analyse the difference in timings of the two versions taken using *omp\_get\_wtime()*: results can be seen in Table 1. For simplicity, the string generation time was omitted as it was quite constant throughout the different combinations of sequence and pattern lengths and can be even smaller and more negligible if we read from file. The analysis was performed without reading from file.

What we can notice is that having a fixed pattern length, the increase of *‘slen’* provides a slightly better time and fraction of parallelizable code. Instead, keeping a constant sequence length and enlarging the pattern, the improvement obtained is much greater.

Below, we can also appreciate in Figure 1 the difference between match and total time, that is to say the behaviour of the percentage of parallelizable code, of version V1 in the serial implementation if we increment the two strings’ size. The serial V2 plot is similar to the first version.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SLEN | PLEN | MATCH TIME 1 | TOTAL TIME 1 | Parallelizable % 1 | MATCH 2 | TOTAL TIME 2 | Parallelizable % 2 |
|  | | **Version 1** | | | **Version 2** | | |
|  | **10** | 0.00003 | 0.0001 | 29.62188 | 0.00003 | 0.00010 | 28.81991 |
|  | **10** | 0.0002 | 0.00046 | 43.478 | 0.00019 | 0.00046 | 40.15063 |
|  |  | 0.00019 | 0.00038 | 51.31718 | 0.00019 | 0.00038 | 49.77864 |
|  | **10** | 0.00194 | 0.00305 | 63.73054 | 0.00175 | 0.00296 | 59.17082 |
|  |  | 0.00177 | 0.0029 | 60.88906 | 0.00173 | 0.00287 | 60.37757 |
|  |  | 0.00448 | 0.00581 | 77.00919 | 0.00387 | 0.00522 | 74.13492 |
|  |  | 0.01949 | 0.02967 | 65.68999 | 0.01547 | 0.02525 | 61.25466 |
|  |  | 0.04122 | 0.05085 | 81.06793 | 0.03991 | 0.04979 | 80.14578 |
|  |  | 0.19065 | 0.28348 | 67.25468 | 0.15494 | 0.24833 | 62.39440 |
|  |  | 0.44433 | 0.53645 | 82.82867 | 0.39396 | 0.48612 | 81.04224 |
|  |  | 12.81745 | 12.90995 | 99.2835 | 13.10603 | 13.19893 | 99.29617 |
|  |  | 4.33195 | 5.23731 | 82.71319 | 3.98295 | 4.89723 | 81.33073 |
|  |  | 137.68859 | 138.59632 | 99.34505 | 144.16187 | 145.08619 | 99.36292 |

Table 1: comparison between serial V1 and serial V2.

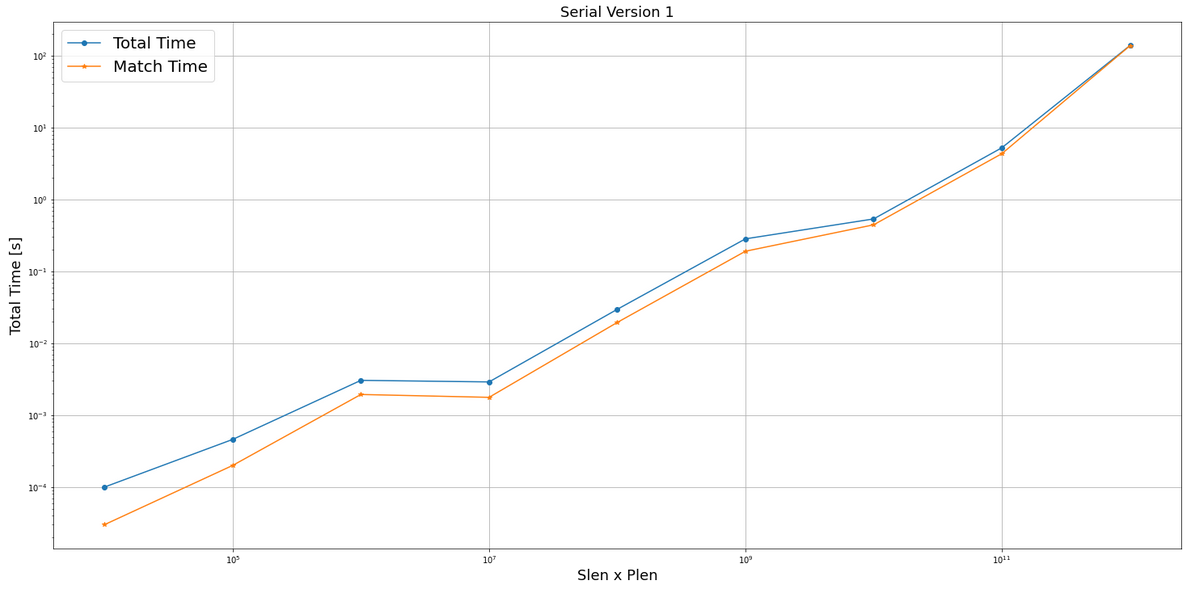


Figure 1: serial V1. Increment of percentage for greater sequence and patterns.

As a result of this analysis, we can apply Amdahl’s law to predict what is the expected speedup in the parallel code.  
Chosen a number of cores *n* and using the percentage of parallelizable code *f* obtained in the previous table we expect a speedup equal to: .

In a practical case, having 16 cores and the combination *slen-plen* of we expect a speedup of 14.45 in the version 1, which is rather promising.

**Parallel openMP implementation:**

To get to a parallel version, first of all we need to choose the number of threads that will be working synchronously. Thus, it was made possible for the user to specify the number of threads (accordingly to the type of architecture the code is run onto). This is then set using the openMP library function *omp\_set\_num\_threads(n\_thr)*.  
As a choice, we did not parallelize the *‘generate’* function as it is a random process of string generation. Instead, we put our efforts on the main core of the whole program, that is the *‘search’* function. Being it a set of two *for* loops, a first and very simple approach could be to use the following openMP construct:  
*#pragma omp parallel for*

in which we obtained the best results selecting a dynamic type of schedule, where each idle thread is given the first available chunk. In order to manage load balancing and to reduce overhead, we decided to increase the chunk size to the ratio between *‘plen’* and the number of threads. Furthermore, we had to insert two critical sections: one is needed to synchronize the update of the counter of correspondences and of their positions. The other one is used to reallocate memory, using the *realloc* C function, in case the correspondence vector gets bigger than what we expected. Clearly, these two critical sections have an impact on execution times as they are synchronization points where threads might need to wait before modifying shared variables. This reasoning is reflected into our *‘parallel\_v1.c’* code.

Another feasible solution could be to implement tasks to parallelize. To do so, we exploited the fact that our version 2 of the serial code had a *search* function that performed both forward and backward in the same function. This new parallel version, indicated as *‘parallel\_v2.c’*, does not need the setting of the number of threads as now we are using tasks. The code contains the *search* function in which we put a parallel section obtained via an openMP pragma. Inside this parallel region, we declare, again using a pragma, two different tasks having *first-private* error counters (initialized to zero): one for the forward search and the other one for the backward scanning. In order to avoid the tasks to be spawned more than once inside the parallel region, we needed to introduce a *#pragma omp single.* Finally, inside the task region we could write our ‘*parallel for’* region that sweeps the two different loops for the two different scanning directions while maintaining the critical sections that we introduced in the first version, too. It is rather obvious that this new version has a much greater complexity and a deeper level of parallelization. It is then possible that we will have to keep in mind a trade-off among complexity and performance.

**Testing:**

After the development of the software and some basic testing to check the correspondences matched, we switched to the “real” testing phase. We greatly enlarged the sizes of both sequence and pattern and started analysing the serial code, first, and then the parallel versions where we could try to improve the results using more cores.

**Performance analysis:**

…