Thibault Chatel

Parallel System – 2020/2021

Shared memory programming project based on Rust

Subject : Robot return to origin
“There is a robot starting at position(0,0), the origin, on a 2D plane. Given a sequence of its moves judge if this robot ends up at (0,0) after it completes its moves.”

1) Sequential solution

A sequential solution of this problem corresponds to the `leetcode_accepted` function. The idea is, we iterate over the chars composing our string with the `chars()` method, then we `map` each direction (‘U’, ‘D’, ‘L’, ‘R’) to a shift in 2d plane coordinate (a tuple) like ‘U’ is mapped to (0,1). After that we `fold`, summing all the coordinates and we return true if the final coordinate is the origin.

2) Parallel solution

The first thing I did was to create a simple struct `Position` to replace the tuple I previously used. With that I implemented the `From<char>` and `Sum` traits to have a more concise code. To parallelize the previous code, int the new function `judge_circle`, I simply modified the `chars()` method by the one from rayon : `par_chars()`. I get a parallel iterator over the chars of the string. Using then the trait’s implementations, I mapped each char to a `Position` instance using `Position::from` and finally I simply have to call the `sum` operation.

3) Evaluation

I wrote a function `generate` in order to get random strings easily. For a moves of size 10_000, the sequential solution took 450 µs to compute the result. For the parallel solution, it depends of the number of threads allowed.
With no specification on the number of threads rayon is allowed to use, when I did my measures, I got the result in 1.3 ms on average. Looking at file xthreads.svg we can see that I use 3 threads. The figure doesn’t look so nice because threads steal half of the work so there is always a thread starving because they are 3, which ends up in too many steals, which delays the fallback to sequential and decreases the performance as we see on the right side of the image.
Now with `RAYON_NUM_THREADS=2` we get a much nicer result, looking at file 2threads.svg . The figure is well organized, there is only a kind of pause at the beginning that I don’t understand. The average time I get is 900 µs so better than with 3 threads which could be counter intuitive without the explanation.

4) Further evaluations

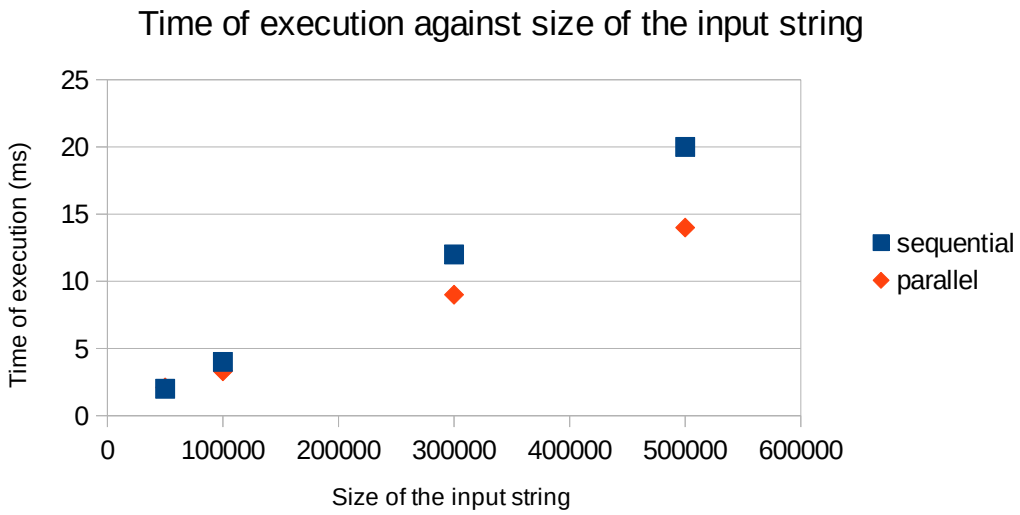
I did more measures in order to compare the sequential solution and the parallel one using 4 threads (I have 2 cores on my machine, using hyper threading).

Here are the results I got with the unit of time being ms :

Size of the string	seq	par(4threads)
50_000	2	2.1
100_000	4	3.3
300_000	12	9
500_000	20	14

(a visualization of the parallel execution for the size 300_00 is available looking at the file 4threads.svg)

Another visualization of these results would be :



We see that both of these curves seem linear, the sequential curve having a higher slope than the parallel one. The bigger the input, the better our parallel solution is compared to the sequential one, in terms of speed of execution.

5) Improvement of the parallel solution

Let's consider a string full of 'U'. A smart algorithm would, after having processed half+1 of the string, stop as it is impossible, given the number of moves left (half-1) and the location of the robot, to go back to origin. With our current implementation there is no such behavior, the whole string is processed.
The ambition of the function `early_stop` is to make this behavior possible while being parallel and having similar performances as `judge_circle`. Let's describe this new function.
We will use 4 atomic variables to help us keep track of the need or not to continue the processing of the string. The 4 variables are :

- keep_going : an `AtomicBool` set by default to true, if its value is false we should not waste anymore time processing the string
- x and y : two `AtomicI32`, representing our current position in the 2D plane, starting (0,0)
- remaining_size : an `AtomicUsize` set by default to the length of the input string, representing the number of char in the string not processed yet

With these, we use our recursive function `tool` which:

- If keep_going is false doesn't do anything
- Else

- If len of the input string<=20_000 we process the whole string input sequentially, we get (dx,dy). After that according to x+dx y+dy and remaining_size we update keep_going
- Else we cut the input string in half and do our recursive calls using join

We return at the end of `early_stop` the boolean `x==0 and y==0`.

Let's evaluate this new function.

First for random strings :

Size of input	seq	par(4threads)	early_stop(4threads)
50_000	2	2.1	2.2
100_000	4	3.3	3.6
300_000	12	9	9.3
500_000	20	14	14

Now for a string full of 'U' :

Size of input	seq	par(4threads)	early_stop(4threads)
50_000	1.7	1.9	1.8
100_000	3.5	3.3	3
300_000	10.3	8.3	5.9
500_000	18	13.8	8.7

We clearly see an improvement where we expected it, so our ambition is fulfilled ! Also we see that in the general case we don't loose that much of performance, maybe adopting another policy for the sequential fallback would increase the performance and make us as fast as the parallel solution (we could use the join_context function to do so) .

Another remark : maybe we could improve the performance using a smarter choice of `Ordering` in our `Atomics` but I don't have the knowledge right now.

6) Possible problem due to `Atomics`

Let's consider the following case :

x=0 ; y=0 ; remaining_size=10 ; and we have 2 threads which own :

t1 t2
len=4 ; dx,dy=(4,0) len=4 ; dx,dy=(-4,0)
and let's consider this order of operations :

l.79
l.84
l.85
x=0 ; y=0 ; remaining_size=6
l.79
l.80
l.81 ← WRONG

(l.X = line number X in the src/main.rs file)

We see that this is problematic and can happen (or maybe the Ordering I used prevent this behavior to happen but I am not sure). I could put the 2 `fetch_add()` lines before the `remaining_size.fetch_sub()` one but I think we could find another example of this kind.
The solution for this would be to wrap the whole x,y and remaining_size inside a Mutex and like that we would make the previous case impossible.