UNIVERSITÀ
DEGLI STUDI
DI PADOVA

The Smith-Waterman algorithm

a multithreaded implementation

Fabio Cimino Giovanni Gaio Lorenzo Paolin

December 16, 2024

Table of Contents

Table of Contents

The problem

Genome and protein alignment are fundamental bio-informatics techniques used to compare sequences of DNA, RNA, or proteins to identify regions of similarity.

These alignments reveal evolutionary relationships, functional similarities, or conserved regions across species or within different genes and proteins of the same organism.

The challenge

Genomes can consist of billions of base pairs (the human genome has over 3 billion base pairs!).
Aligning such large datasets requires significant computational resources.

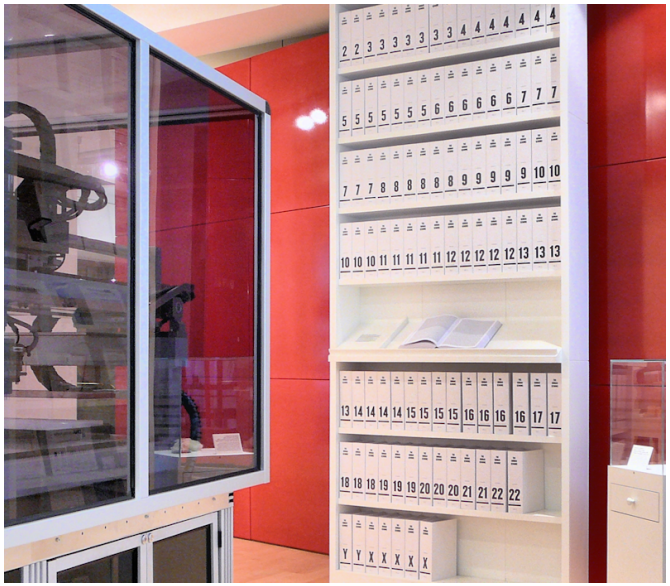
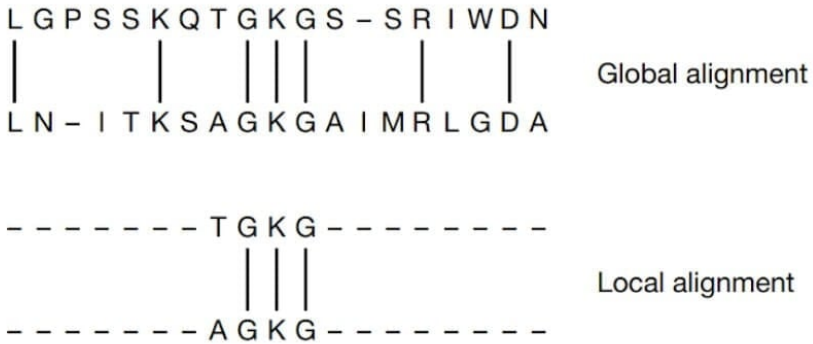


Figure: Human genome printout

Figure: Enter Caption

How does the algorithm work?

The Smith-Waterman algorithm is a dynamic programming algorithm used specifically for local sequence alignment, that is to identify the most similar subsequences between two sequences.



How does the algorithm work?

A scoring matrix H is created and filled as follows. The first row and column are initialized to zero.

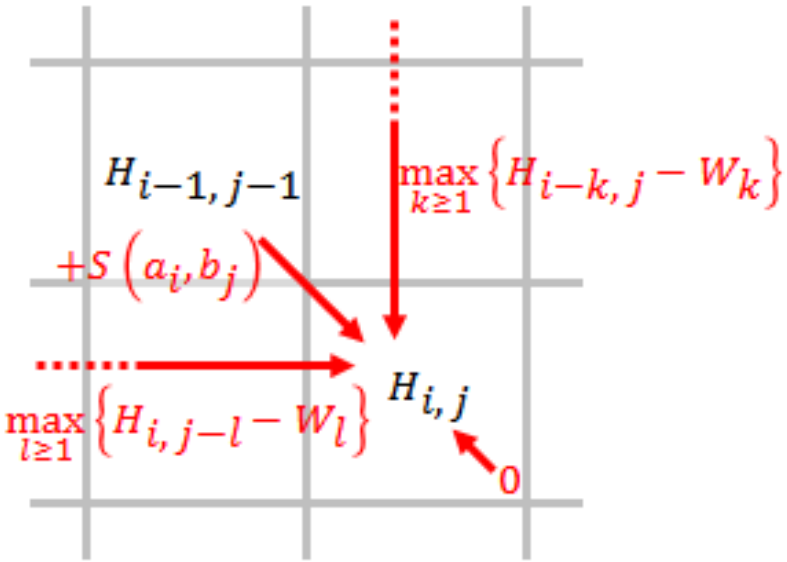
$$H(i,j) = \max \begin{cases} 0 & \\ H(i-1,j-1) + s(i,j) & \text{(match/mismatch)} \\ H(i-1,j) + g & \text{(deletion)} \\ H(i,j-1) + g & \text{(insertion)} \end{cases}$$

- $H(i,j)$ is the score at position (i,j),
- $s(i,j)$ is the score for a match or mismatch,
- g is the penalty for a gap. It can be a function of the gap size.

Algorithm data dependencies

Highlight the dependence of $H(i,j)$ from:

- the previous element along the diagonal $H(i-1,j-1)$,
- the previous element along the column $H(i,j-1)$,
- the previous element along the row $H(i-1,j)$.



Furthermore, sequences can be insanely huge!

Table of Contents

Implementation challenges

This is a great and fun challenge!

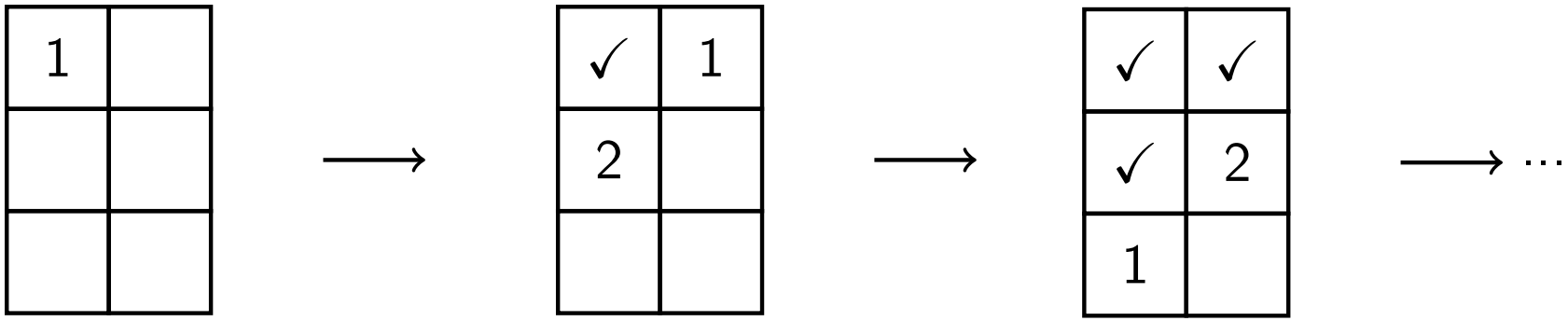
- How do we decompose the problem to handle the dependency chain?
- How do we handle synchronization, shared memory access and cache efficiency?
- How do we balance the load?

Our solutions

We have experimented 3 different approaches using OpenMP:

1. Using locks
2. Using atomic counters
3. Using tasks

Locks



Divide H in blocks and assign a lock to each. All threads set their locks.

When a thread wants to work on a block, it sets the lock corresponding to the previous block along the column. When it's done, it destroys such lock, and unsets its own.

Atomic counters

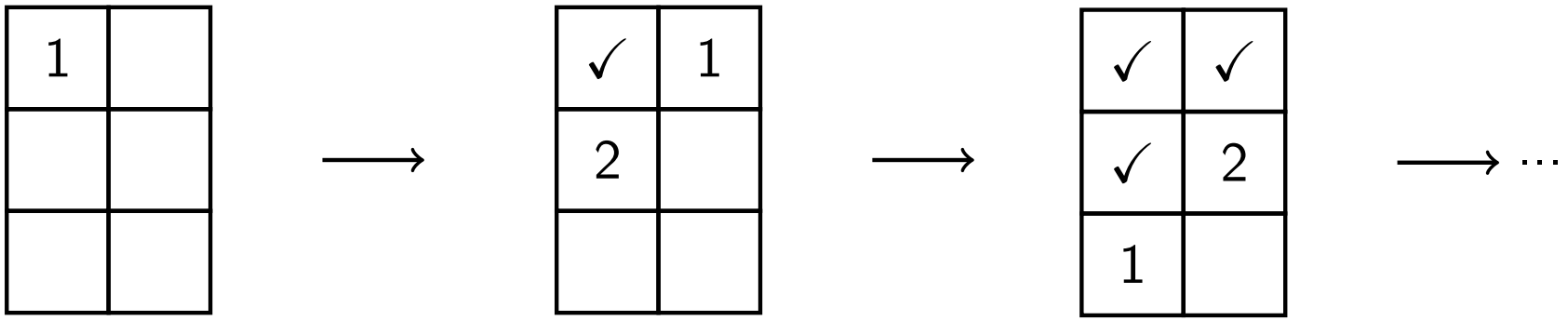


Figure:
$C_1 = 0, C_2 = 0$

Figure:
$C_1 = 1, C_2 = 0$

Figure:
$C_1 = 2, C_2 = 1$

Each thread holds a counter of how many blocks it has processed.
Thread $i + 1$ waits until thread i has processed at least one more block.
When threads jump the condition is a little different.

Tasks

This is a recursive Divide & Conquer approach:

```
int solve(int i0, int i1, int j0, int j1,
      char*A, char*B, scores_t*scores,int*
     H, int*Mj, int*Mi){
   int ans0,ans1,ans2,ans3;
   if(/*block is big enough*/){
       int im = (i0+i1)/2, jm = (j0+j1)/2;
      ans0  = solve(i0, im, j0, jm, A, B,
           scores, H, Mj, Mi);
#pragma omp task
      ans1 = solve(im, i1, j0, jm, A, B,
           scores, H, Mj, Mi);
#pragma omp task
      ans2 = solve(i0, im, jm, j1, A, B,
           scores, H, Mj, Mi);
#pragma omp taskwait
      ans3 = solve(im, i1, jm,j1, A, B,
           scores, H, Mj, Mi);
      return max(ans0,ans1,ans2,ans3);
   }else ; // solve normally...
```

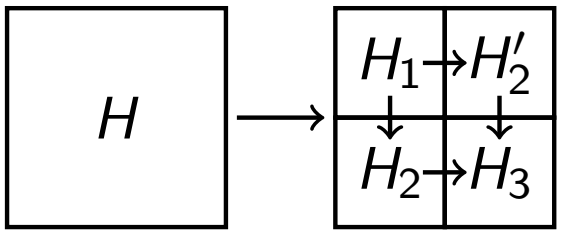


Table of Contents

Test setup

We generated test data starting from the genome of an organism found online. We then copied it and modified it. We then compared the original and the modified versions.
Hardware characteristics:

- CPU: Intel i7-1165G7 (8) @ 2.8GHz
- RAM: 8 GB
- OS: Arch Linux (I use arch btw)

Performances

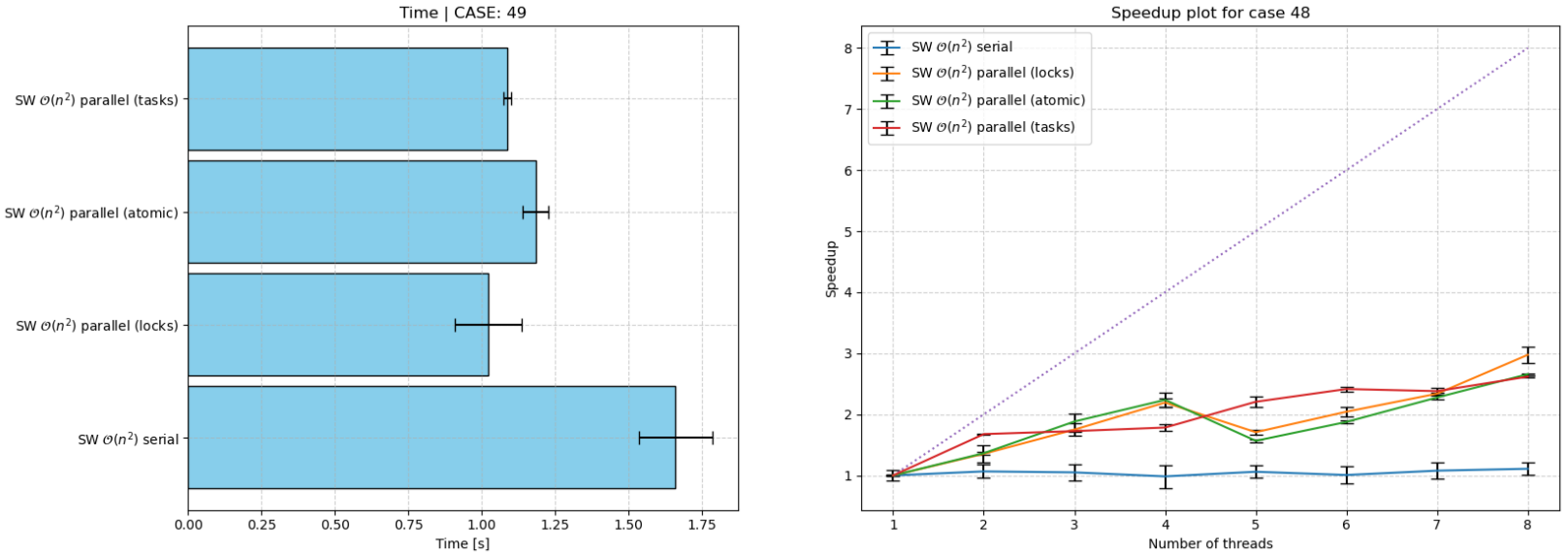


Figure: Running times and speedup plots for each of the algorithms.

Insights

Algorithm	1 thread	2 threads	4 threads	8 threads
Serial				
Locks				
Tasks				

Table: Average running times for a sample input of size X and Y, for different numbers of maximum number of threads.

It is interesting to see approximately the same results for each of the approaches. The only notable difference is that the one using locks has a larger variation of the duration between runs.

Table of Contents

Moving forward

A number of different approaches are possible:

1. Using a similar approach as the ones used, **MPI** could be well suited to solve this problem: the communication between process is much smaller than the computation done by each of them.
2. **GPUs** could also be very effective, as the computation can proceed row by row and column by column.

Thank you!

Backup slide

Some additional content