SPP Project
Global Alignment

Aarush Jain
Siddh Jain

Outline

Global Alignment

A global alignment is defined as the end-to-end alignment of two strings A and B.

Finding the minimum edit distance using dynamic programming is the global alignment problem. Closely related sequences which are of same length are very much appropriate for global alignment. Here, the alignment is carried out from beginning till end of the sequence to find out the best possible alignment.

Global alignment using Dynamic Programming

- ➔ Our goal is to maximise the performance of creating the matrix for global alignment.
- ➔ We take the inputs from two fasta files and then compute the matrix for global alignment.
- ➔ The Time Complexity is O(m*n).
- ➔ The Space Complexity is O(m*n).

Compute Configuration

CPU Model	Intel Core i7-10750H
Generation	10th
Frequency Range	0.8GHz - 5GHz
Number of cores	6
Hyperthreading availability	Yes
SIMD ISA	AVX-512
Cache size	12288 KB
Main memory Bandwidth	45.8GB/s

Number of sockets	1
Cores/socket	4
Max CPU Frequency	5
Cycles/sec	5*1e9
Flops/cycle	8

Correctness Evaluation

We are checking the correctness of our program by using a checker function which runs after the execution of our optimised functions.

So using this we can check if the executed function is giving us the right answers. We are using the row-wise dynamic programming approach for the checker function.

This function is present in the align.c file.

```
int checker(char* A,char* B,int A_len,int B_len,int match,int mismatch,int gap
{
    //fill matrices
    for(int i=1; i < A_len; ++i){
        for(int j=1; j < B_len; ++j){
            int score_diag = M[i-1][j-1] + (A[i] == B[j] ? match : mismatch);
            int score_left = M[i-1][j] + gap;
            int score_up = M[i][j-1] + gap;
            if(score_diag <= score_left && score_diag <= score_up)
                M[i][j] = score_diag;
            else if(score_left <= score_up && score_left <= score_diag)
                M[i][j] = score_left;
            else if(score_up <= score_left && score_up <= score_diag)
                M[i][j] = score_up;
            // printf("%d ",M[i][j]);
        }
    }
    return M[A_len-1][B_len-1];
}
```

Performance Evaluation

The performance of our program is done by calculating the execution time of the matrix generation function- 'GlobalAlignment'.

It is done using the functions tick and tock which measure the start and stop time of the function.

The functions tick and tock are present in helper.c file.

```c
void tick(struct timeval *t)
{
    gettimeofday(t, NULL);
}

double tock(struct timeval *t)
{
    struct timeval now;
    gettimeofday(&now, NULL);
    return(double) (now.tv_sec - t->tv_sec) +
    ((double)(now.tv_usec - t->tv_usec)/1000000.);
}
```

Datasets Used

The dataset used for execution is from the sample dataset provided for alignment problems. These are two fasta files containing the DNA information of-

- HIV-1 isolate MB2059 from Kenya, complete genome
- HIV-1 isolate SF33 from USA, complete genome

The compute time is based on the test_1 dataset in the small dataset. The length of each sequence in dataset is of order 10e4.

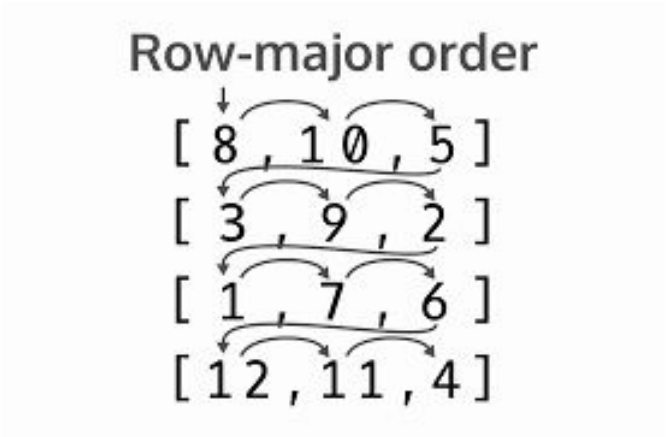
Performance Achieved

Baseline performance

The baseline performance was achieved using row wise dynamic programming.

We get a decent performance here as the cache utilization is good. We get a clear cache miss only on first cell of each row of the matrix.

The execution time in row wise dynamic programming: 1430 milliseconds.



The optimisation techniques used

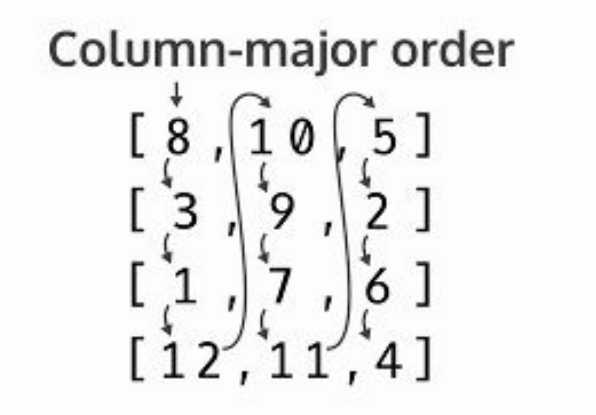
- ➔ Column wise dynamic programming
- ➔ Parallelisation using diagonal computing.
- ➔ Tiling with different block sizes. (with row wise implementation in a tile)
- ➔ Tiling with different block sizes. (with parallelising diagonally in a tile)

Column wise Dynamic Programming

The first optimisation used was using column wise dynamic programming.

We get the worst performance here as the cache utilization is quite bad. We end up getting cache misses on every cell of a column as in every iteration the cache is filled up with the row of the matrix.

The execution time in column wise dynamic programming:1130 milliseconds.

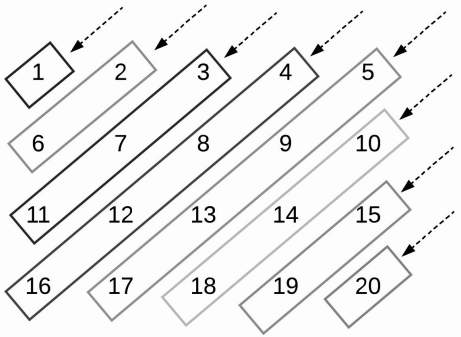


Diagonal Computing

The performance dropped if we traverse the matrix diagonally.

This is as the cache utilization is not good. We end up getting a lot of cache misses.

The execution time in diagonal compute dynamic programming: 1280 milliseconds.



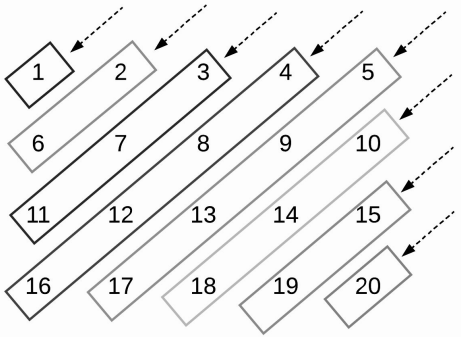
1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 15, 19, 20

Diagonal Computing with Parallelization

The performance increases drastically if we traverse the matrix diagonally and parallelise it.

Even though the cache utilization is not optimum, we get quite a good performance boost.

The execution time in Diagonal Computing with Parallelization: 210 milliseconds.

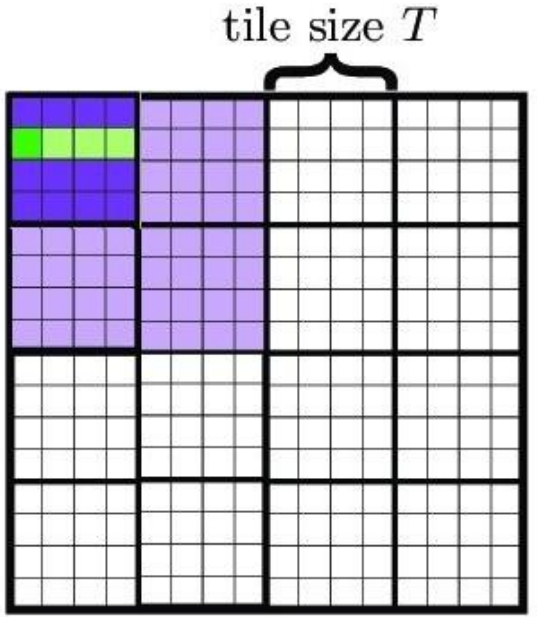


1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 15, 19, 20

Tiling/Block formation

A variety of performance boost and depreciation is seen in tiling depending on the tile size. Here we tile the matrix before executing the tiles parallely.

The performance is also dependent on factors like tile size and the method used to traverse the tiles.

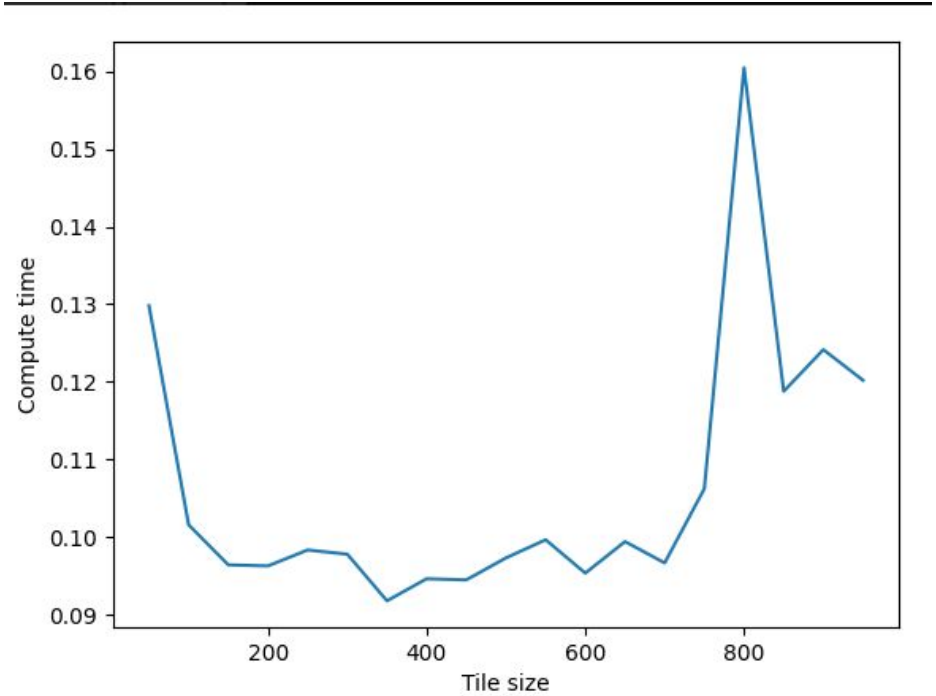


Execution time for Tiling

1. Tiling with tile executed row-wise and no parallelisation = 600 ms.
2. Tiling with tile executed row-wise and parallelisation = 92 ms.
3. Tiling with tiles executed diagonally and no parallelisation = 1430 ms.
4. Tiling with tiles executed diagonally and parallelisation = 330 ms.

The best time is achieved with tile size of ~ 350.

Execution times for Tiling with various tile sizes (executed row-wise and parallelisation)



Tile size	Compute time(in sec)
50	0.129809
100	0.101560
150	0.096398
200	0.096264
250	0.098316
300	0.097775
350	0.091762
400	0.094610
450	0.094462
500	0.097296
550	0.099636
600	0.095332
650	0.099400
700	0.096650
750	0.106218
800	0.160479
850	0.118768
900	0.124137
950	0.120191

Comparing Performances

Column Wise	Row Wise	Diagonal only	Diagonal parallelised	Tiling with with tile executed row-wise and no parallelisation	tiling with tile executed row-wise and parallelisation	tiling with tiles executed diagonally and no parallelisation	tiling with tiles executed diagonally and parallelisation
1130 ms	510 ms	1280 ms	210 ms	600 ms	92ms	1430 ms	330ms

Best Performance and speedup-

The best performance was achieved with tiling with tile executed row-wise and parallelisation which is 110 milliseconds. It is a far improvement from the baseline execution which had a time of 1130 milliseconds.

So the speedup is (Baseline execution time/Best execution time)

which is: 1430/92 = 15.54

Other Insights

Hirschberg's algorithm

Hirschberg's algorithm is a generally applicable algorithm for optimal sequence alignment.

If x and y are strings, where length(x) = n and length(y) = m, the Needleman–Wunsch algorithm finds an optimal alignment in O(nm) time, using O(nm) space. Hirschberg's algorithm is a clever modification of the Needleman–Wunsch Algorithm, which still takes O(nm) time, but needs only O(min{n, m}) space and is much faster in practice.