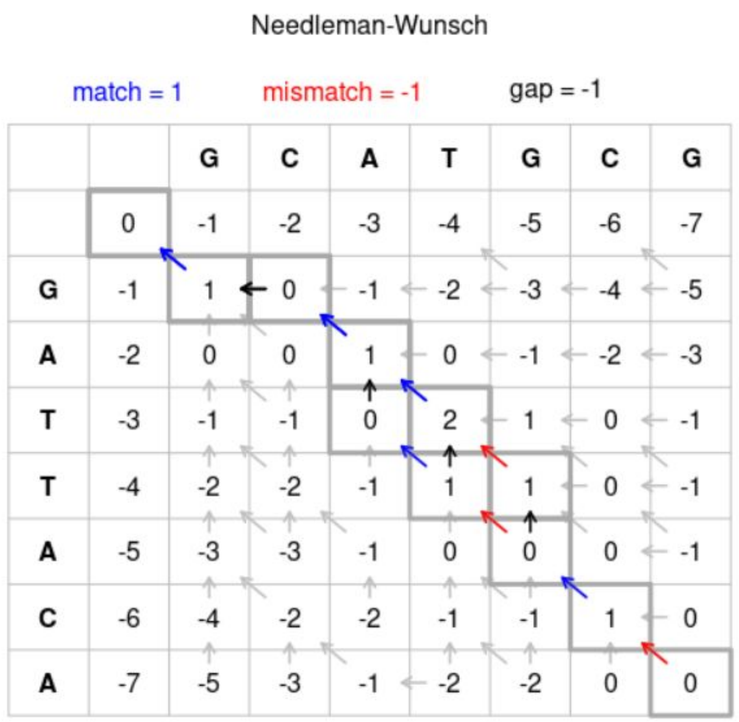
A Comparison of Global Sequence Alignment Algorithms for Shared and Distributed Memory Machines

Max Farrington

University at Buffalo The State University of New York

What is global sequence alignment?

- Global sequence alignment is a bioinformatics technique for aligning two [or more] protein sequences with respects to the whole sequence.
- Every alignment is evaluated by maintaining a scoring matrix.
- Positive and negative scores are granted based on matches or mismatches.
    - Based on the use case, you can change the scoring scheme (ex: +1 match, -1 insertion/deletion, -1 mismatch)
- The best alignment is then found by backtracing from the bottom right

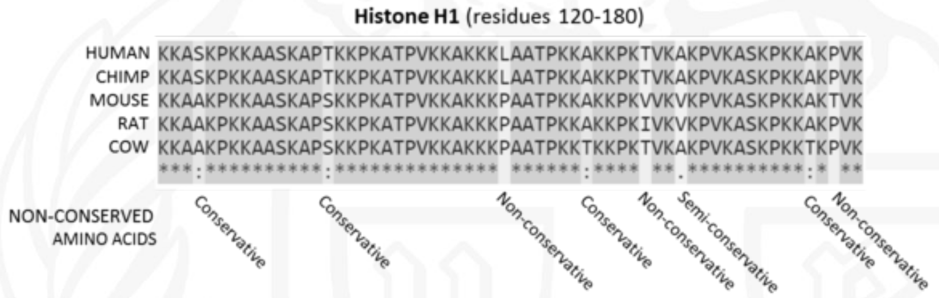


		G	C	A	T	G	C	G
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

What is it actually used for?

- When comparing the sequences of two subjects that share a common ancestor, you can view the mismatches, insertions, and deletions as mutations from that ancestor.
- You can then derive the importance of specific subsequences by how they are preserved in descendants of that ancestor.
- Millions of subsequences have also been tagged/identified for specific behavior.
    - You can find similarities between untagged/tagged sequences to find known genes in a sequence.



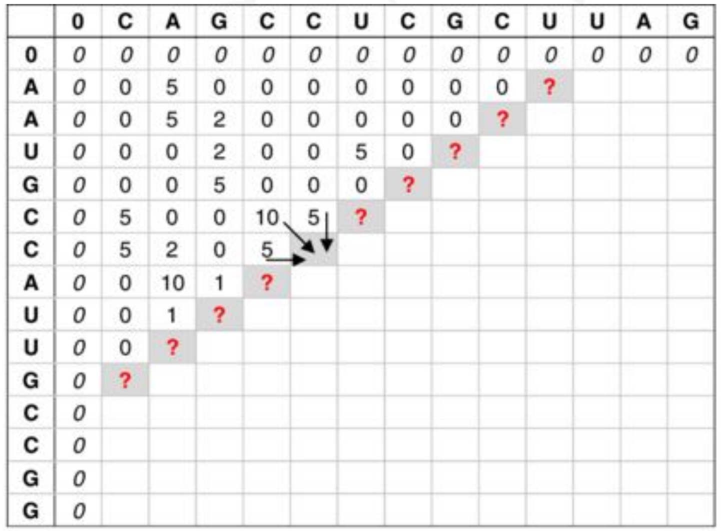
Histone H1 (residues 120-180)

```
HUMAN KKASKPKKAASKAPTKKPKATPVKKAKKKLAATPKKAKKPKTVKAKPVKASKPKKAKPVK
CHIMP KKASKPKKAASKAPTKKPKATPVKKAKKKLAATPKKAKKPKTVKAKPVKASKPKKAKPVK
MOUSE KKAAKPKKAASKAPSKKPKATPVKKAKKKPAATPKKAKKPKVVKVKPVKASKPKKAKTVK
  RAT KKAAKPKKAASKAPSKKPKATPVKKAKKKPAATPKKAKKPKIVKVKPVKASKPKKAKPVK
  COW KKAAKPKKAASKAPSKKPKATPVKKAKKKPAATPKKTKKPKTVKAKPVKASKPKKTKPVK
      ***:**********:************** ******:**** **.**********:* **
```

https://en.wikipedia.org/wiki/Sequence_alignment

Why is it a good parallel programming problem?

- To compute the running score, there are minimal data dependencies, allowing for computation to be done in parallel either row/column-wise, or along the anti-diagonal.
- These methods have tradeoffs in terms of efficiency and space complexity.
    - For anti-diagonal solutions, you only need to store the current and previous anti-diagonal to get the score, which changes in size as you fill the matrix. This is possible since you are only taking the maximum of cells to the top, left, and top left of a given cell.
    - For row/column wise solutions you need to store the current and previous row, but the size stays fixed. Getting a value for a cell is more complicated in parallel, but load balancing is better than the antidiagonal method.

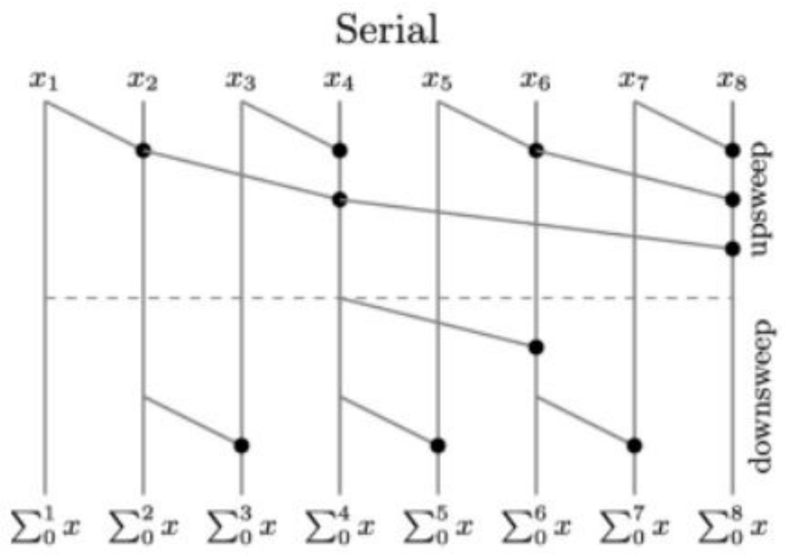


	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	5	0	0	0	0	0	0	0	?			
A	0	0	5	2	0	0	0	0	0	?				
U	0	0	0	2	0	0	5	0	?					
G	0	0	0	5	0	0	0	?						
C	0	5	0	0	10	5	?							
C	0	5	2	0	5									
A	0	0	10	1	?									
U	0	0	1	?										
U	0	0	?											
G	0	?												
C	0													
C	0													
G	0													
G	0													

https://www.researchgate.net/figure/Anti-diagonal-method-and-dependency-of-the-cells_fig11_222408669

How the algorithm relates to parallel prefix

- Parallel prefix takes a binary associative operator (such as +, -, *, MAX(), etc.) and an array of n elements, and for each element, computes and stores a running total based on the chosen operator.
- In the case of the Needleman Wunsch algorithm, we are effectively keeping a running total, but the value in each spot depends on the max value of its neighbors that have already been computed.
- Needleman Wunsch also allows for negative values in the matrix, so the work can very easily be split into chunks with the preceding values communicated via parallel prefix.


Serial
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
upsweep
downsweep
$\sum_0^1 x$ $\sum_0^2 x$ $\sum_0^3 x$ $\sum_0^4 x$ $\sum_0^5 x$ $\sum_0^6 x$ $\sum_0^7 x$ $\sum_0^8 x$

https://link.springer.com/chapter/10.1007/978-3-031-12597-3_21

$$T[i,j] = \max\begin{cases} T[i-1,j-1] + f(a_i,b_j), \\ T[i-1,j] - g, \\ T[i,j-1] - g. \end{cases}$$

$$w[j] = \max\begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h). \end{cases}$$

Then,

$$T_2[i,j] = \max\begin{cases} w[j], \\ T_2[i,j-1] - g. \end{cases}$$

Let

$$\begin{aligned} x[j] &= T_2[i,j] + jg \\ &= \max\begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g, \end{cases} \\ &= \max\begin{cases} w[j] + jg \\ x[j-1]. \end{cases} \end{aligned}$$

Aluru et all.

University at Buffalo The State University of New York

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

Parallel example - mpi

	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	**-1 (1)**	**-3 (1)**			**-9* (1)**			
A	-4								
T	-6								
G	-8								

$$w[j] = \max\begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h). \end{cases} \qquad T_2[i,j] = x[j] - jg.$$

$$\begin{aligned} x[j] &= T_2[i,j] + jg \\ &= \max\begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g, \end{cases} \\ &= \max\begin{cases} w[j] + jg \\ x[j-1]. \end{cases} \end{aligned}$$

w[1] = Max(T[0,0] + Match(B[0],A[0]) ,T[0,1] - 2) = -1

x[j] = max(-1 + 1(2), -∞) = 1

T[1,1] = 1 - 1(2) = -1

w[2] = Max(T[0,1] + Match(B[0],A[1]) ,T[0,2] - 2) = -3

x[j] = max(-3 + 2(2), 1) = 1

T[1,2] = 1 - 2(2) = -3

w[5] = Max(T[0,4] + Match(B[0],A[4]) ,T[0,5] - 2) = -9

x[j] = max(-9 + 5(2), -∞) = 1

T[1,5] = 1 - 5(2) = -9

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

$$w[j] = \max\begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h). \end{cases} \qquad T_2[i,j] = x[j] - jg.$$

$$\begin{aligned} x[j] &= T_2[i,j] + jg \\ &= \max\begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g, \end{cases} \\ &= \max\begin{cases} w[j] + jg \\ x[j-1]. \end{cases} \end{aligned}$$

Parallel example - mpi

	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	-1 (1)	-3 (1)	-3* (3)	-5* (3)	-9* (1)	-11* (1)	-15* (1)	-15* (1)
A	-4								
T	-6								
G	-8								

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

$$w[j] = \max \begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h). \end{cases} \qquad T_2[i,j] = x[j] - jg.$$

$$\begin{aligned} x[j] = T_2[i,j] + jg \\ &= \max \begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g, \end{cases} \\ &= \max \begin{cases} w[j] + jg \\ x[j-1]. \end{cases} \end{aligned}$$

Parallel example - mpi

	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	-1 (1)	-3 (1)	-3* (3)	-5* (3)	-9* (1)	-11* (1)	-15* (1)	-15* (1)
A	-4								
T	-6								
G	-8								

Prefix time!

Binary associative operator - Max(x)

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

$$w[j] = \max\begin{cases} T_1[i,j-1]-(g+h), \\ T_3[i,j-1]-(g+h). \end{cases} \qquad T_2[i,j] = x[j] - jg.$$

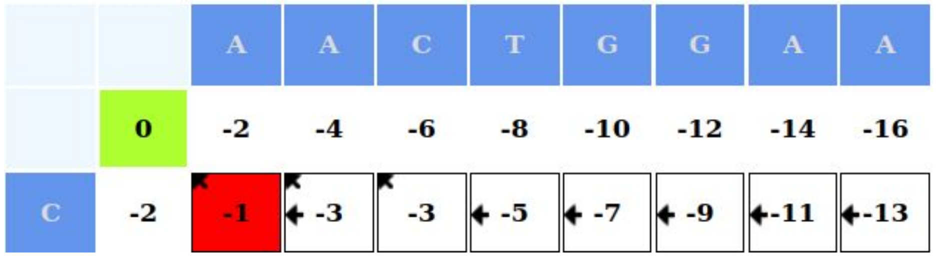
$$\begin{aligned} x[j] &= T_2[i,j] + jg \\ &= \max\begin{cases} w[j]+jg \\ T_2[i,j-1]+(j-1)g, \end{cases} \\ &= \max\begin{cases} w[j]+jg \\ x[j-1]. \end{cases} \end{aligned}$$

Parallel example - mpi

	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	-1 (1)	-3 (1)	-3 (3)	-5 (3)	-7 (3)	-9 (3)	-11 (3)	-13 (3)
A	-4								
T	-6								
G	-8								

Now, recompute x[j] using the value received during the prefix scan, and use for calculating T[i,j]

Additionally, share the last value in your local row with the processor next to you



		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	↖ -1	↖← -3	↖ -3	← -5	← -7	← -9	←-11	←-13

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

Parallel example - openmp

Wavefront algorithm starting from M[1][1]

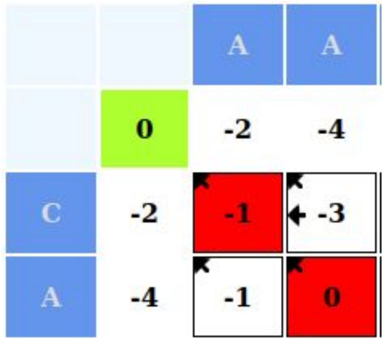
	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	-1							
A	-4								
T	-6								
G	-8								

M[1][1] = Max(
M[0][1] + penalty,
M[1][0] + penalty,
M[0][0] + Match(A[0],B[0])) = -1

Gap penalty: -2, mismatch: -1, match 1
A = AACTGGAA
B = CATG

Parallel example - openmp

	0	1	2	3	4	5	6	7	8
		A	A	C	T	G	G	A	A
	0	-2	-4	-6	-8	-10	-12	-14	-16
C	-2	-1	-3						
A	-4	-1							
T	-6								
G	-8								



		A	A
	0	-2	-4
C	-2	-1	-3
A	-4	-1	0

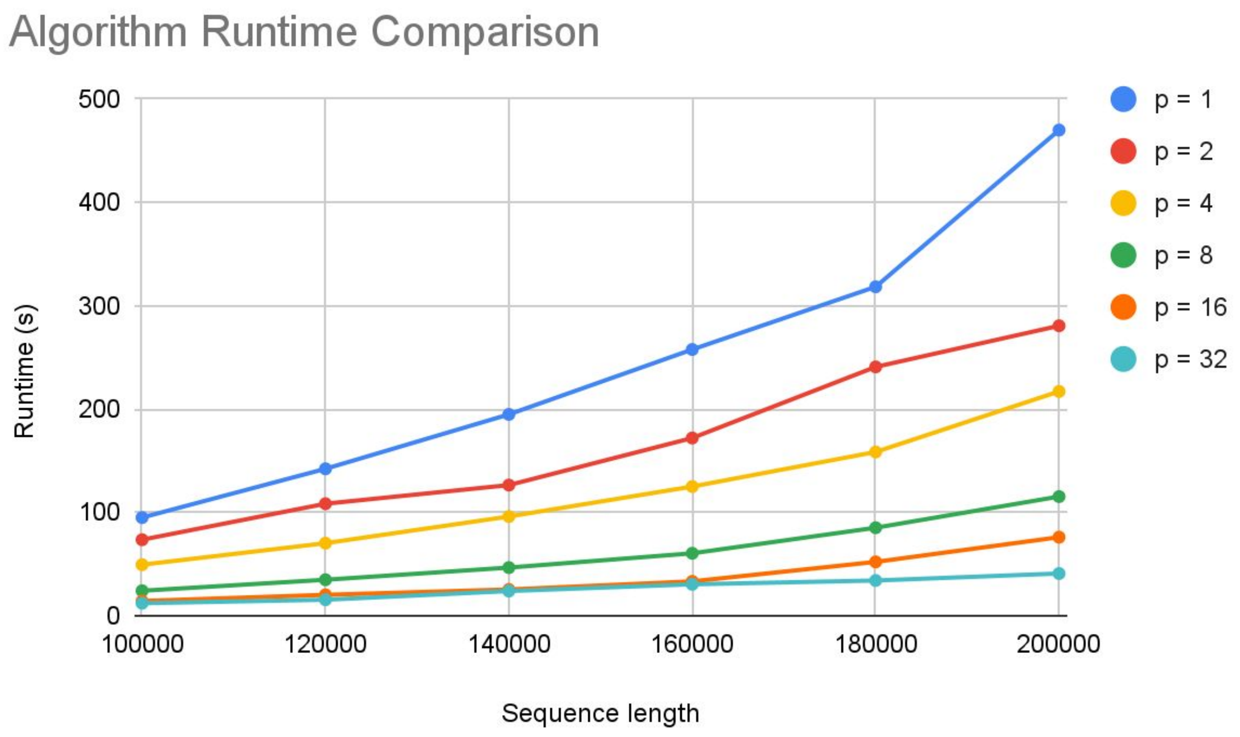
Wavefront algorithm on second antidiagonal

M[1][2] = Max(
M[0][2] + penalty,
M[2][1] + penalty,
M[1][0] + Match(A[1],B[2])) = -1

M[2][1] = Max(
M[1][1] + penalty,
M[2][0] + penalty,
M[1][0] + Match(A[1],B[0])) = -3

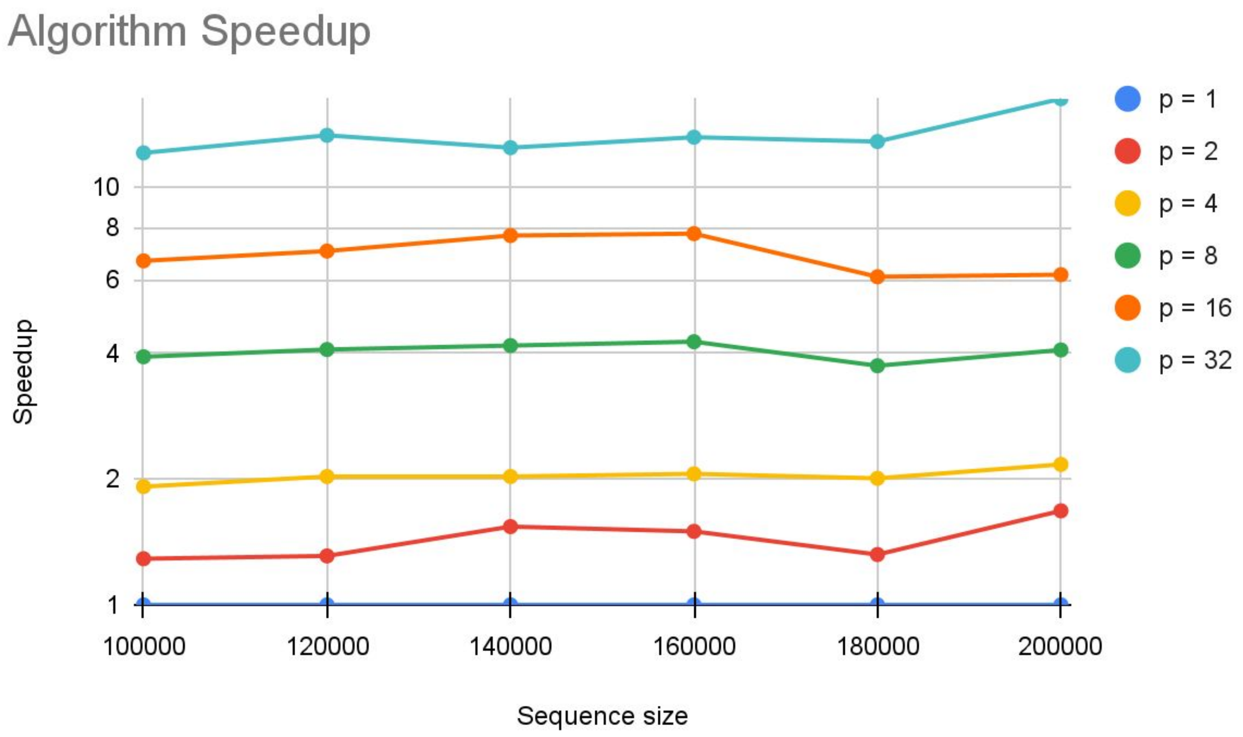
Benchmarking and scalability

Runtime analysis - mpi



n/p	100000	120000	140000	160000	180000	200000
1	94.972	142.176	194.823	257.642	318.329	470.006
2	73.5367	108.38	126.419	171.986	240.79	280.597
4	49.3603	70.1906	95.9919	125	158.477	217.044
8	24.1509	34.7976	46.5727	60.3988	85.1384	115.308
16	14.2428	20.195	25.385	33.2346	52.0935	75.9925
32	7.85093	10.6655	15.6509	19.5389	24.7102	28.8214

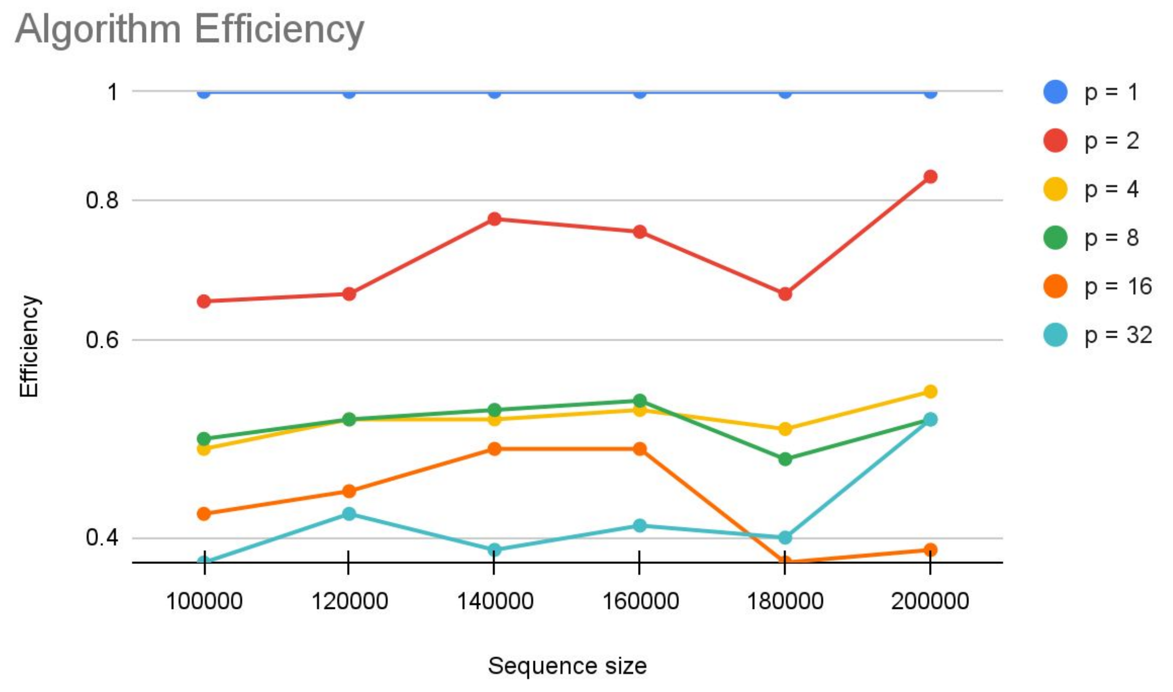
Speedup - mpi



	100000	120000	140000	160000	180000	200000
1	1	1	1	1	1	1
2	1.29	1.31	1.54	1.5	1.32	1.68
4	1.92	2.03	2.03	2.06	2.01	2.17
8	3.93	4.09	4.18	4.27	3.74	4.08
16	6.67	7.04	7.67	7.75	6.11	6.18
32	12.1	13.33	12.45	13.19	12.88	16.31

$$\text{Speedup} = \frac{T_1}{T_p}$$

Efficiency - mpi

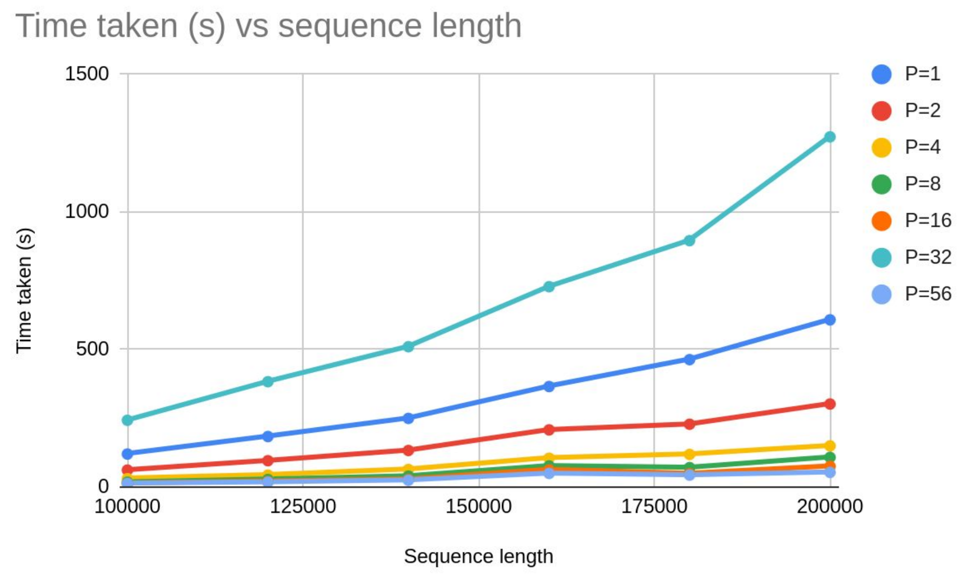


	100000	120000	140000	160000	180000	200000
1	1	1	1	1	1	1
2	0.65	0.66	0.77	0.75	0.66	0.84
4	0.48	0.51	0.51	0.52	0.5	0.54
8	0.49	0.51	0.52	0.53	0.47	0.51
16	0.42	0.44	0.48	0.48	0.38	0.39
32	0.38	0.42	0.39	0.41	0.4	0.51

$$\text{Efficiency} = \frac{T_1}{pT_p}$$

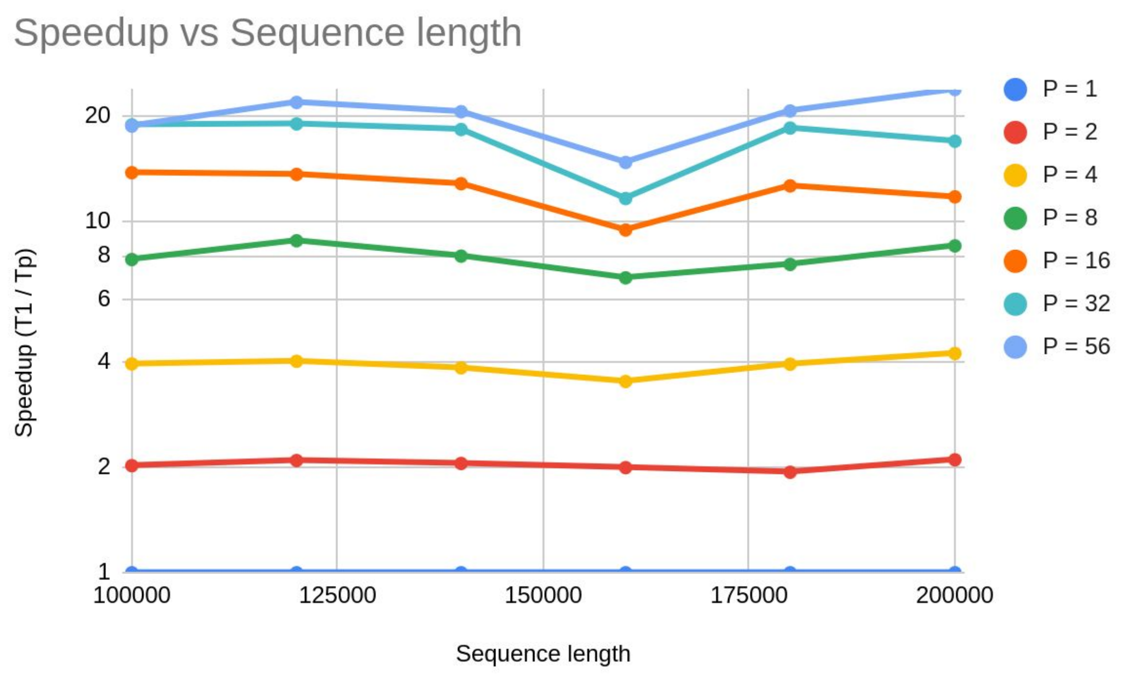
Runtime analysis - OpenMP

	1	2	4	8	16	32	56
100000	242.477	120.287	61.7209	31.1008	17.5827	12.856	12.9571
120000	383.087	183.703	95.7509	43.4616	28.0965	20.1859	17.533
140000	510.382	249.488	133.274	63.9765	39.8006	27.8714	24.8244
160000	728.152	365.687	207.714	105.315	76.8874	62.5928	49.3791
180000	896.089	463.543	228.322	118.692	70.8847	48.5388	43.3767
200000	1273.36	607.803	302.292	149.379	108.307	75.1523	53.5564



Speedup - OpenMP

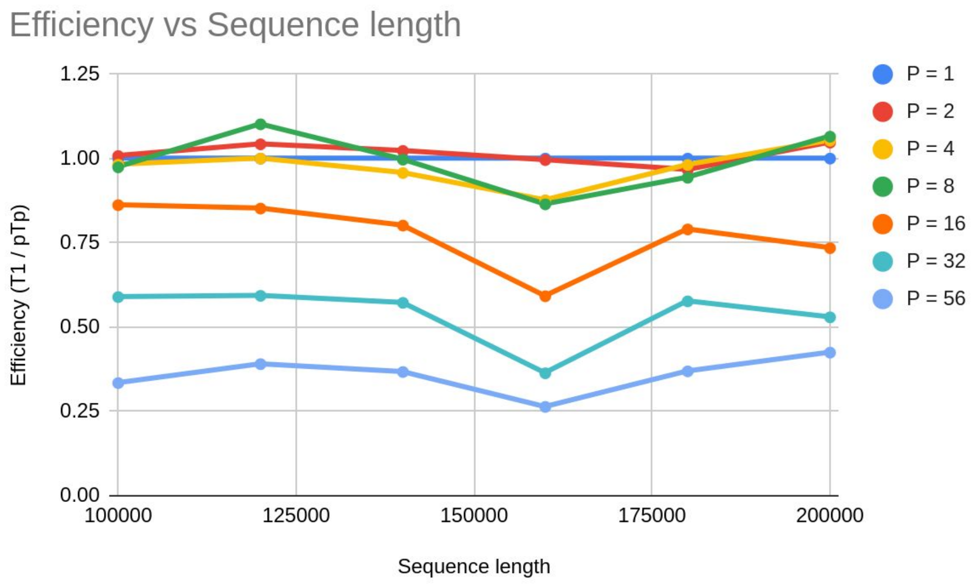
P/|S|	1	2	4	8	16	32	56
100000	1.00	2.02	3.93	7.80	13.79	18.86	18.71
120000	1.00	2.09	4.00	8.81	13.63	18.98	21.85
140000	1.00	2.05	3.83	7.98	12.82	18.31	20.56
160000	1.00	1.99	3.51	6.91	9.47	11.63	14.75
180000	1.00	1.93	3.92	7.55	12.64	18.46	20.66
200000	1.00	2.10	4.21	8.52	11.76	16.94	23.78



$$\text{Speedup} = \frac{T_1}{T_p}$$

Efficiency - OpenMP

P/|S|	1	2	4	8	16	32	56
100000	1.00	1.01	0.98	0.97	0.86	0.59	0.33
120000	1.00	1.04	1.00	1.10	0.85	0.59	0.39
140000	1.00	1.02	0.96	1.00	0.80	0.57	0.37
160000	1.00	1.00	0.88	0.86	0.59	0.36	0.26
180000	1.00	0.97	0.98	0.94	0.79	0.58	0.37
200000	1.00	1.05	1.05	1.07	0.73	0.53	0.42



$$\text{Efficiency} = \frac{T_1}{pT_p}$$

Conclusions

- Clear Strong and weak scaling up to 16 processors for almost all sequence lengths from 100000-20000 with Openmp.
  - Strong and weak scaling begins to fall off around 32 processes.
- Cost of communication vs thread coordination apparent when comparing efficiency of OpenMP and MPI.
  - While Openmp seems to have much better scaling, MPI implementation is clearly much faster. This would only be further accentuated with multiple processes per node with MPI.
- Still more testing to be done!

University at Buffalo The State University of New York

References

- https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm
- https://en.wikipedia.org/wiki/Sequence_alignment
- https://www.researchgate.net/figure/Anti-diagonal-method-and-dependency-of-the-cells_fig11_222408669
- https://link.springer.com/chapter/10.1007/978-3-031-12597-3_21
- Srinivas Aluru, Natsuhiko Futamura, Kishan Mehrotra, Parallel biological sequence comparison using prefix computations,
- https://bioboot.github.io/bimm143_W20/class-material/nw/