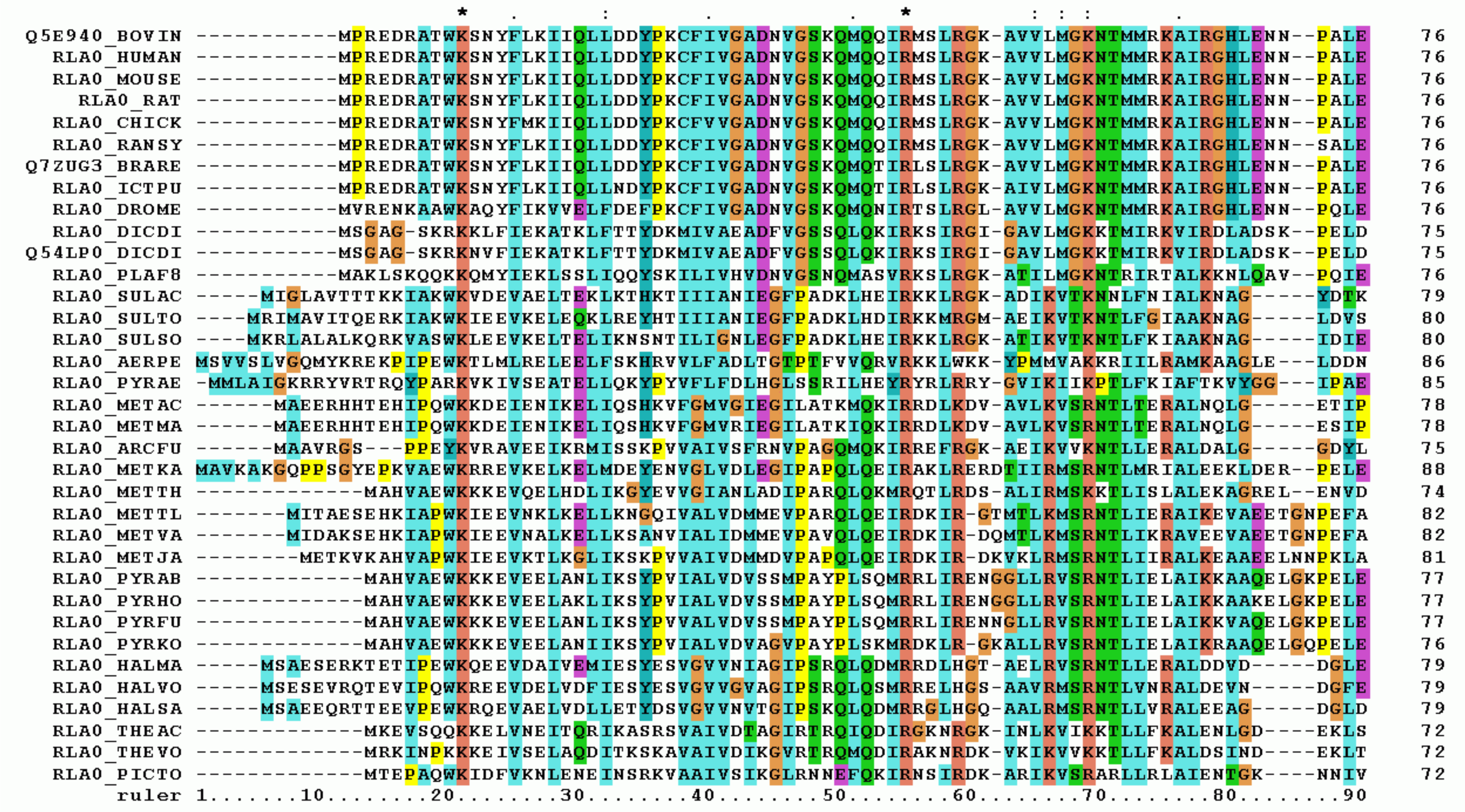
A* on Hypercube for Multiple Sequence Alignment
OpenMP Implementation

Jian Chen

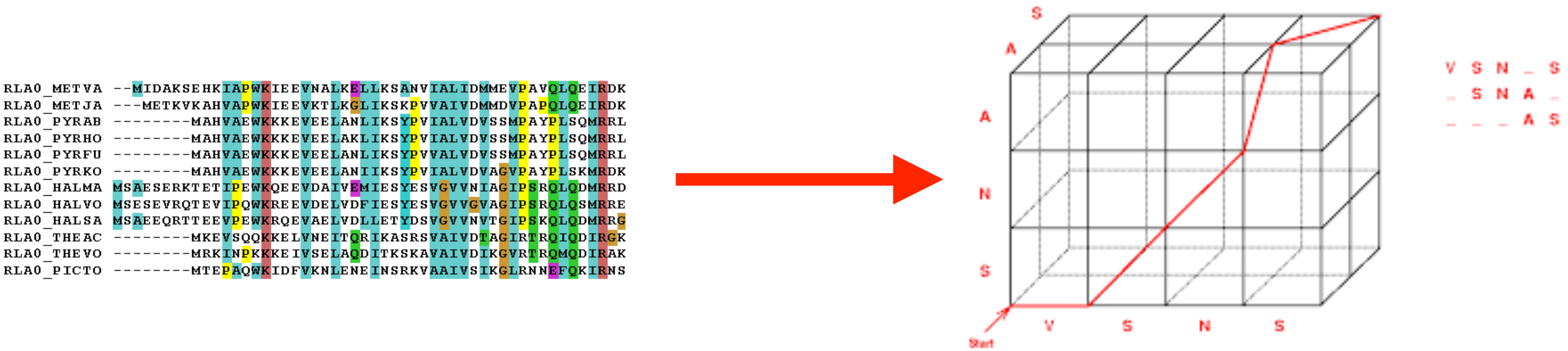
11/21/2019

Protein Sequence Multiple Alignment



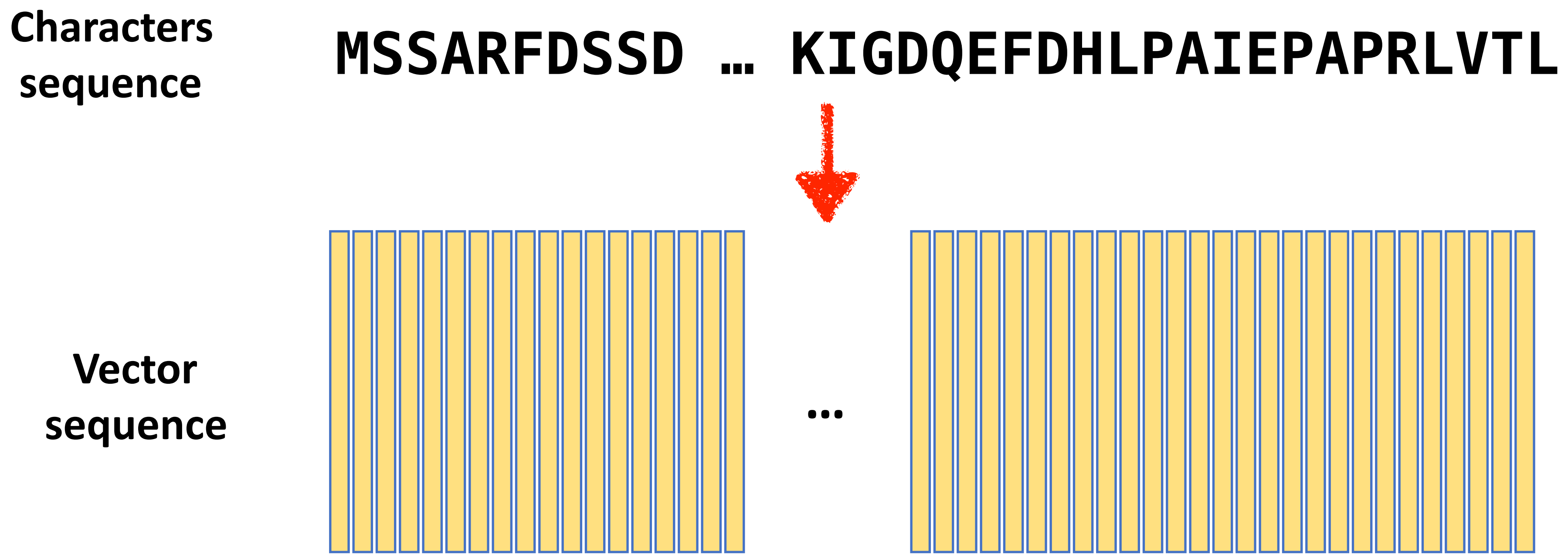
Motivation:

The main problem in multiple sequence alignment makes it so much harder than pairwise alignment is the curse of dimensionality of the cost hypercube. The memory cost to save the value in a hypercube increases exponentially.

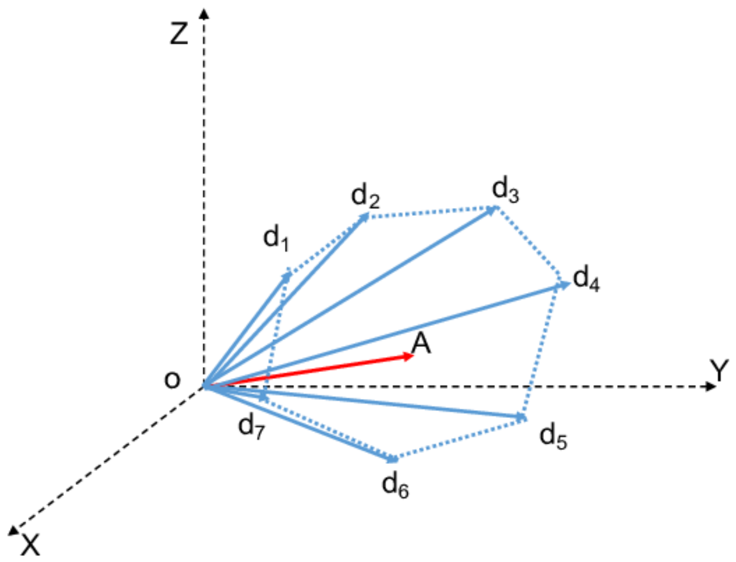


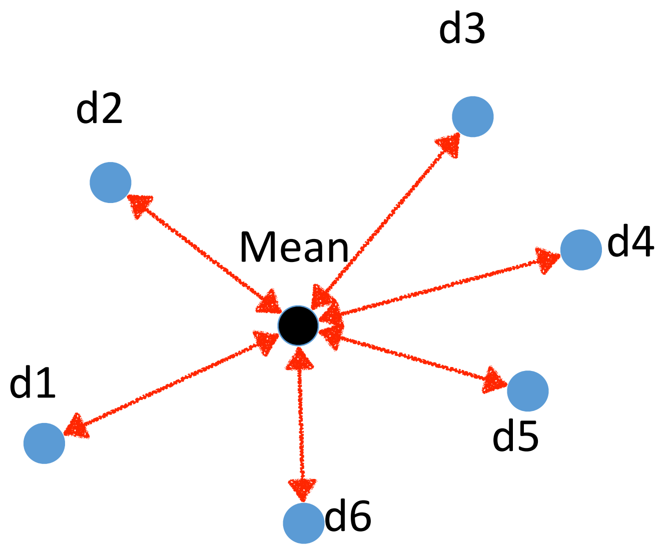
Method:

We can substitute the Matching score with an embedding vector cosine similarity. We find this embeddings have the ability to measure the dissimilarity between aligned patches.



The dissimilarity between multiple patches can be captured be the convex cone formed by the N embedding vectors, since they preserve the relationship in their cosine similarity.





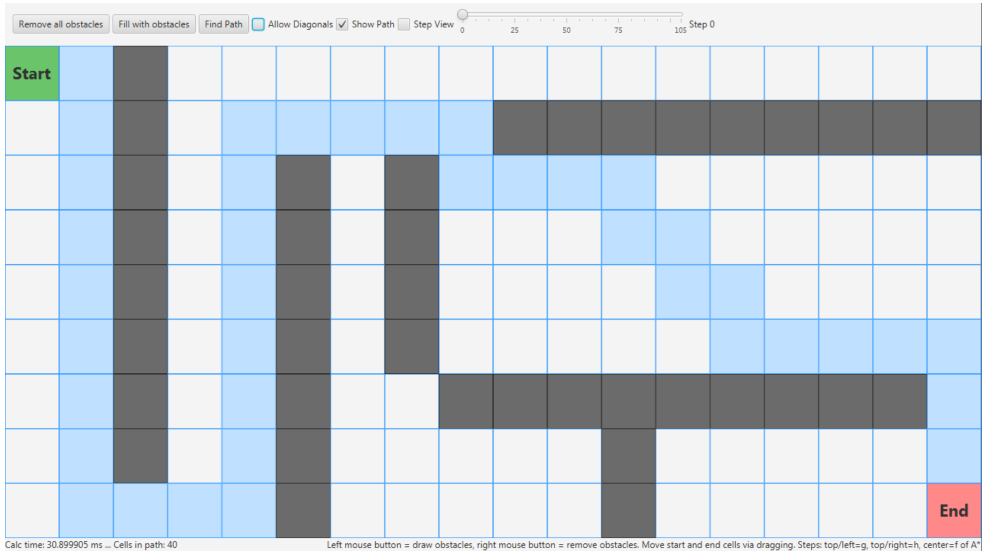
A* algorithm

A* is a classical path search algorithm. At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where *n* is the next node on the path, ***g*(*n*)** is the cost of the path from the start node to *n*, and ***h*(*n*)** is a heuristic function that estimates the cost of the cheapest path from *n* to the goal. Which encourage the agent go directly to the goal.

In this way, we only need to save the values on the path. It is a way to reduce the computational complexity to O(NL).



The max over an exponential number of choices is still a barrier in the path search on hypercube.
e.g. find the max/min value in the bellman recursion step.

$$V(I_0) = cosy(I_0) + \beta \max_{a:I_0 \to I_1} V(I_0, a)$$

As shown in the figure, the possible move for even a single step increases exponentially w.r.t. the number of sequences to work on (note as N).

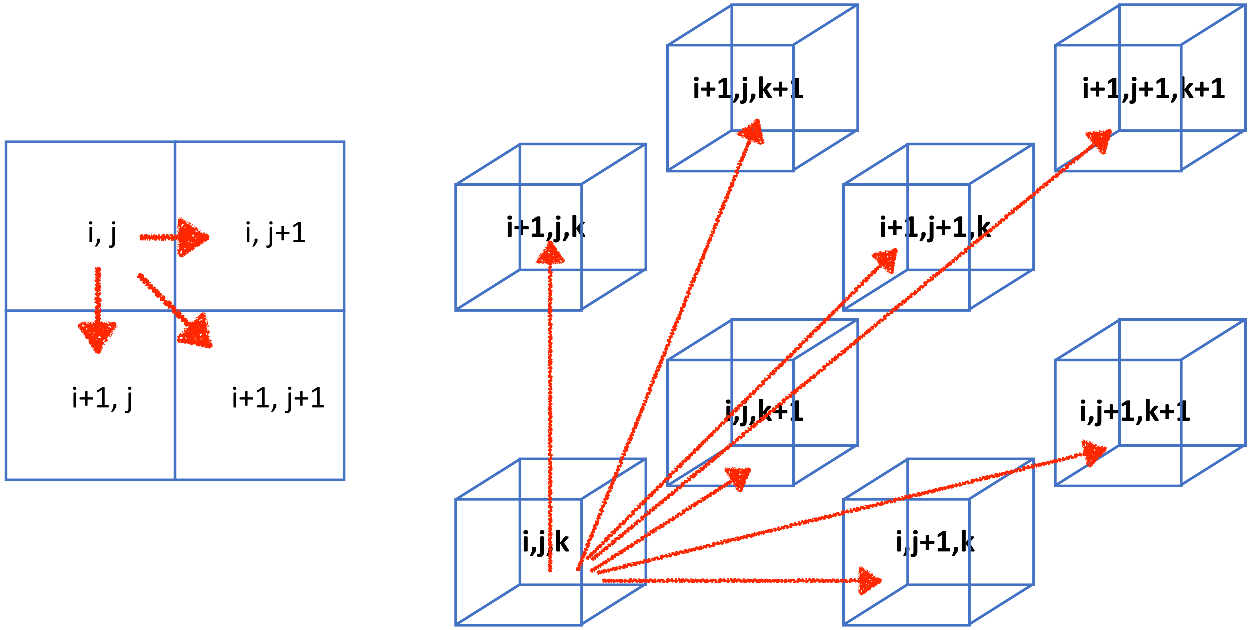
It is easy to see the single step chooses correspond to all vertex on the hypercube except for the origin.

$$\begin{array}{c} (1,0) \\ (1,1) \\ (0,1) \end{array}$$

Choices on square

$$\begin{array}{cc} (1,0,0) & (0,1,1) \\ (0,1,0) & (1,0,1) \\ (0,0,1) & (1,1,0) \\ & (1,1,1) \end{array}$$

Choices on 3D cube



N sequences, ND, $2^N - 1$ Neighbors.

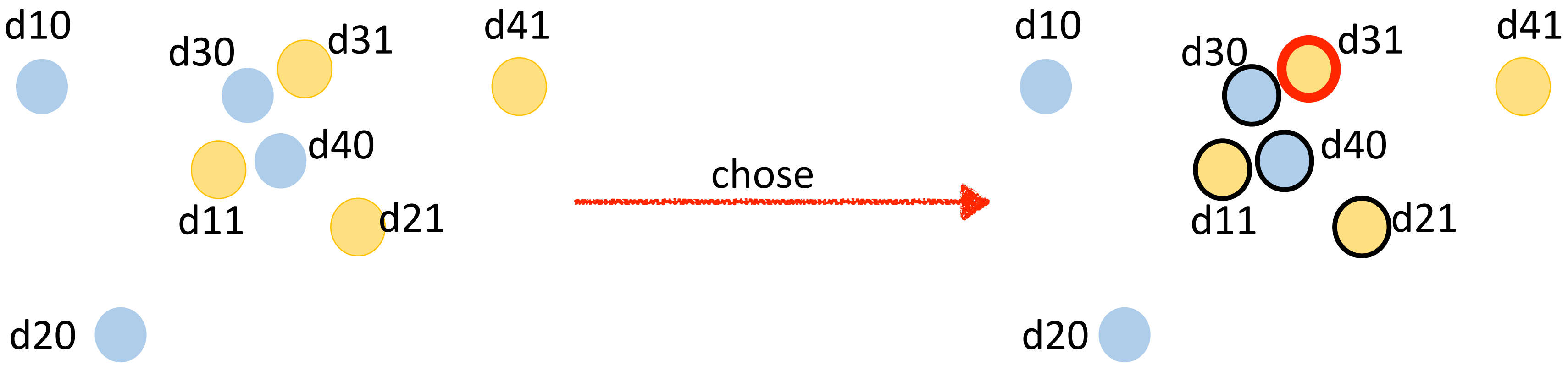
So the min choice is equal to:
Given N pair of data, each pair has one data from set S0 and one from S1. Try to choose on element from each pair of data, such that the chosen element form the most compact cluster.

A solution example on 4D hypercube, we chose d11, d21, d30, d40 to form the most compact cluster. We give up on d31, because we can only chose 1 element in each pair (each column). And the choice we made give use the step (1,1,0,0)

set	1	2	3	4
S0	d10	d20	d30	d40
S1	d11	d21	d31	d41

Step

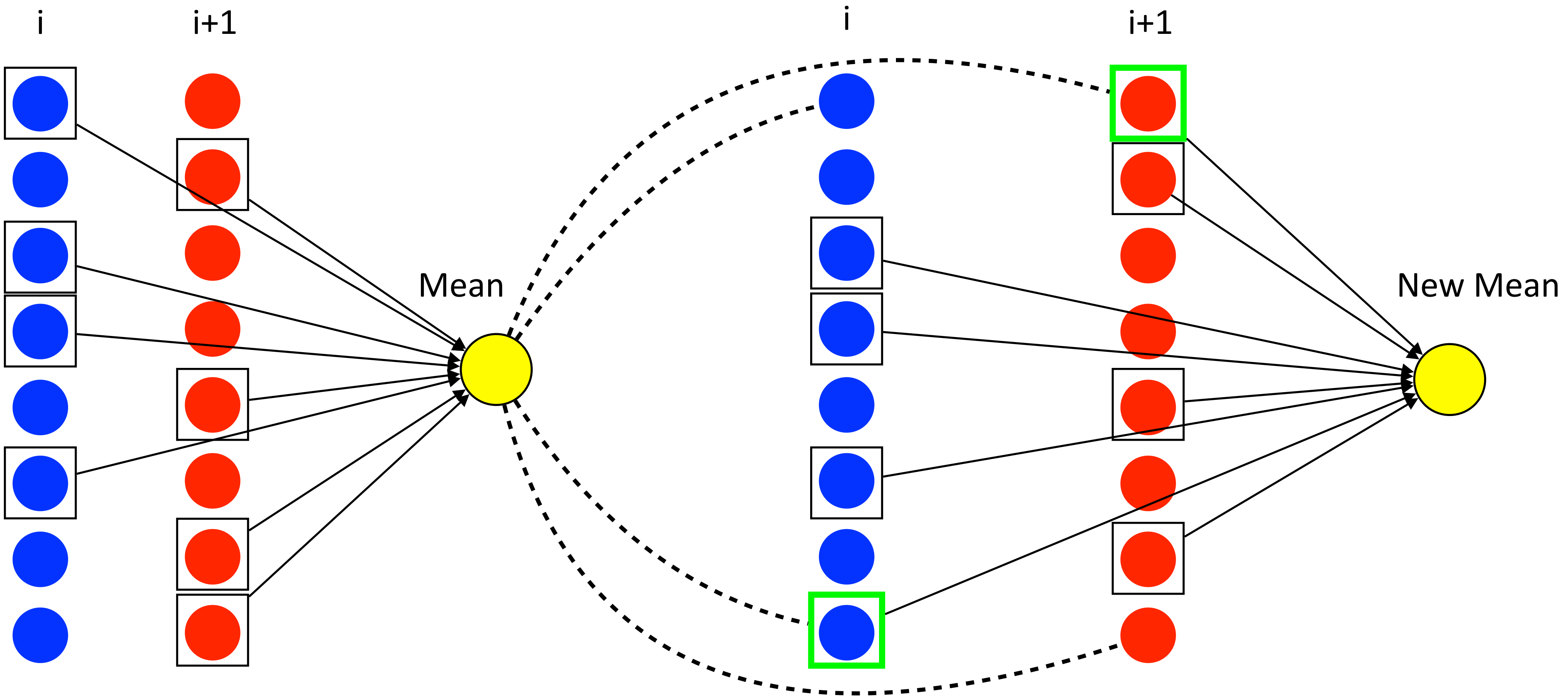
$$(1, 1, 0, 0)$$



We can find a local optimal solution for this min choice problem in a iterative way similar to k-means algorithms. The convergence to a local optimal is guaranteed.

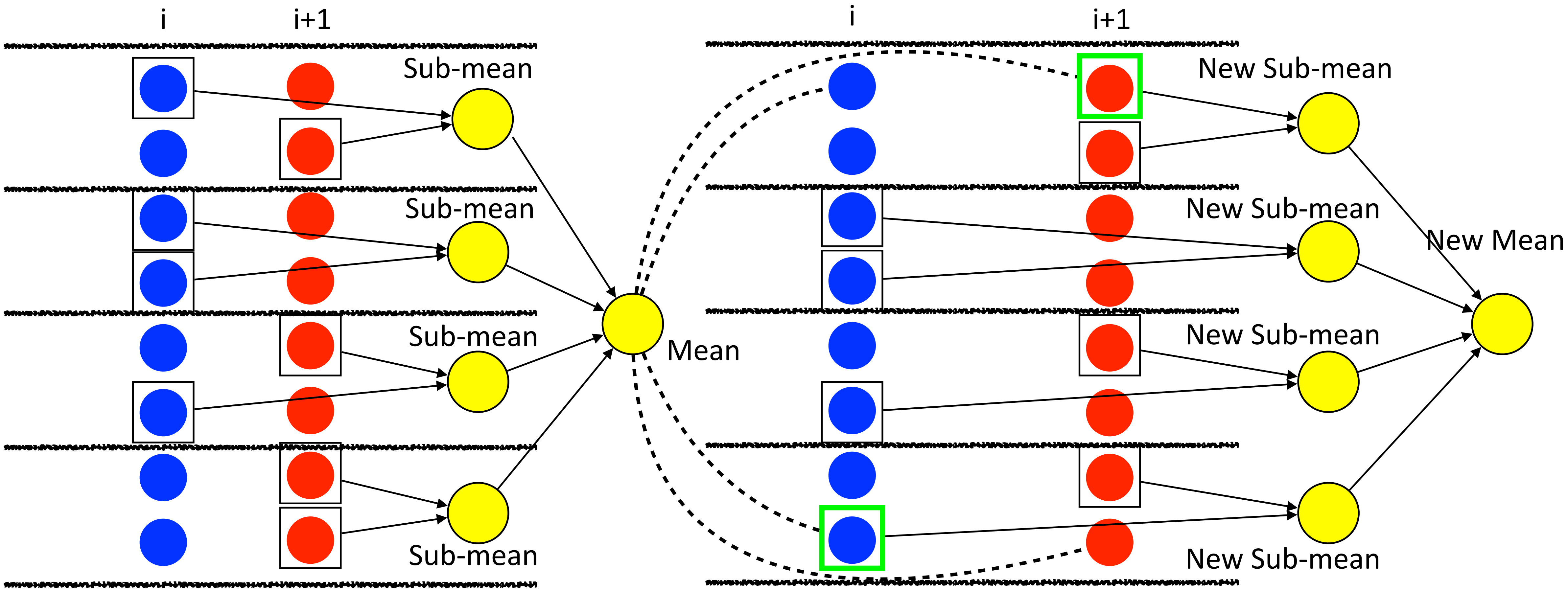
```
chose initial choice to be step = (1, 1, 1, 1, … ,1).
compute the initial center C_0, of by all embeddings at position 1.
C_old = C_0
while C_new != C_old:
        C_old = C_new
        for i = 1 to N:
                current_choice = step[i]     (0 or 1)
                alternate_choice = 1 - step[i]     (1 or 0)
                distance_current = D(C_old, embedding[i][current_choice])
                distance_alternate = D(C_old, embedding[i][alternate_choice])
                if distance_current > distance_alternate:
                        step[i] = 1 - step[i]
        C_new = sum(embedding[i][step[i]])
```

Sequential



(0,1,0,0,1,0,1,1) → (1,1,0,0,1,0,1,0)

Parallel



(0,1,0,0,1,0,1,1) → (1,1,0,0,1,0,1,0)

Code Structure

```
#include <stdio.h>
#include <omp.h>
#include <sstream>
#include <iostream>
#include <string>
#include <vector>
```

$$omp\_set\_num\_threads(N);$$

Initialize vectors for sequences

Initialize empty mean vectors

check # thread avail
print, define variables

$$while(not\ over)$$

$$\#pragma\ omp\ parallel\ private(myid)\{$$

Parallel Code

$$myid = omp\_get\_thread\_num();$$

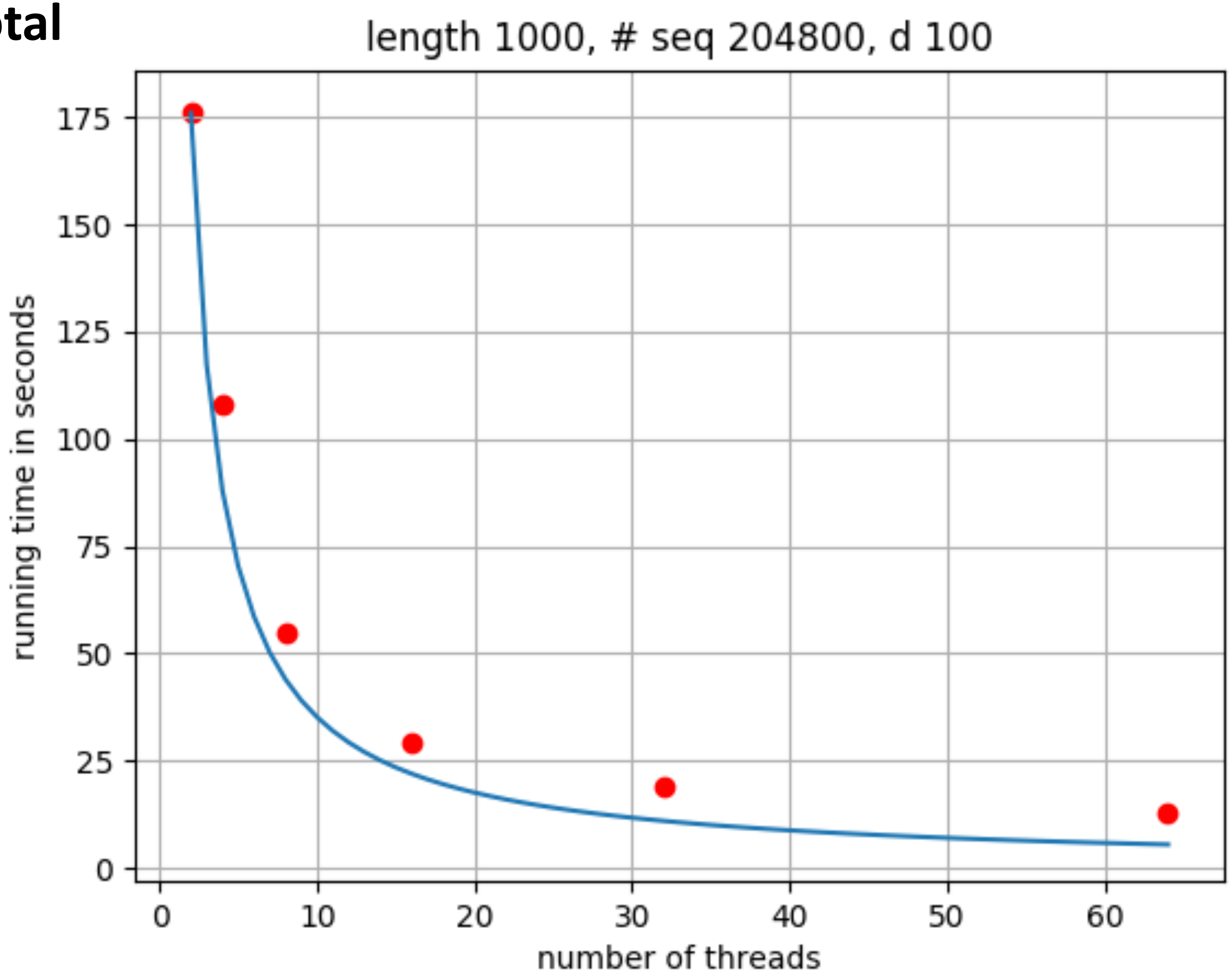
Parallel Code

$$\}$$

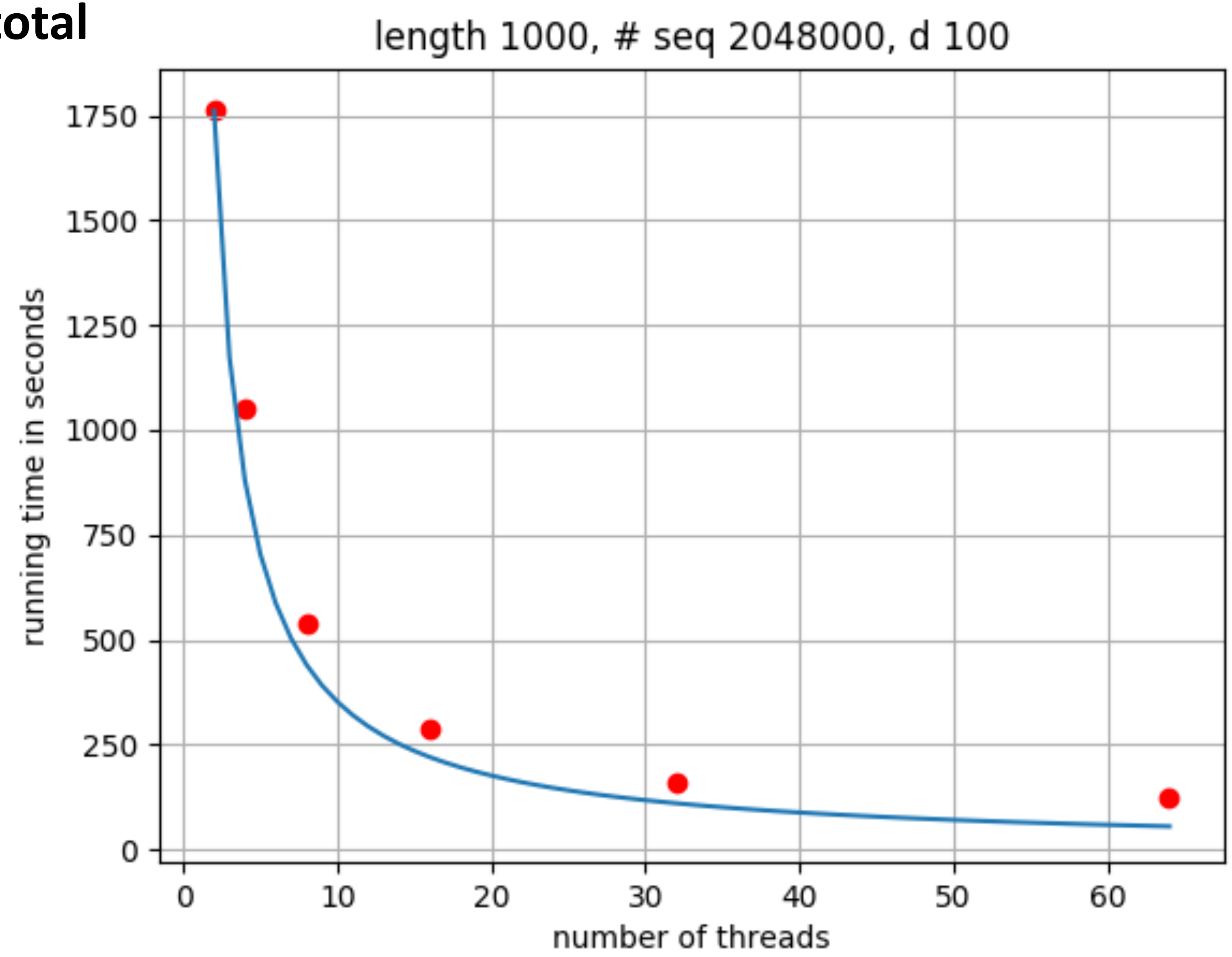
Sequential Code

AWS EC2 instance: m5a.16xlarge (- ECUs, 64 vCPUs, 2.5 GHz, AMD EPYC 7571, 256 GiB memory, EBS only)

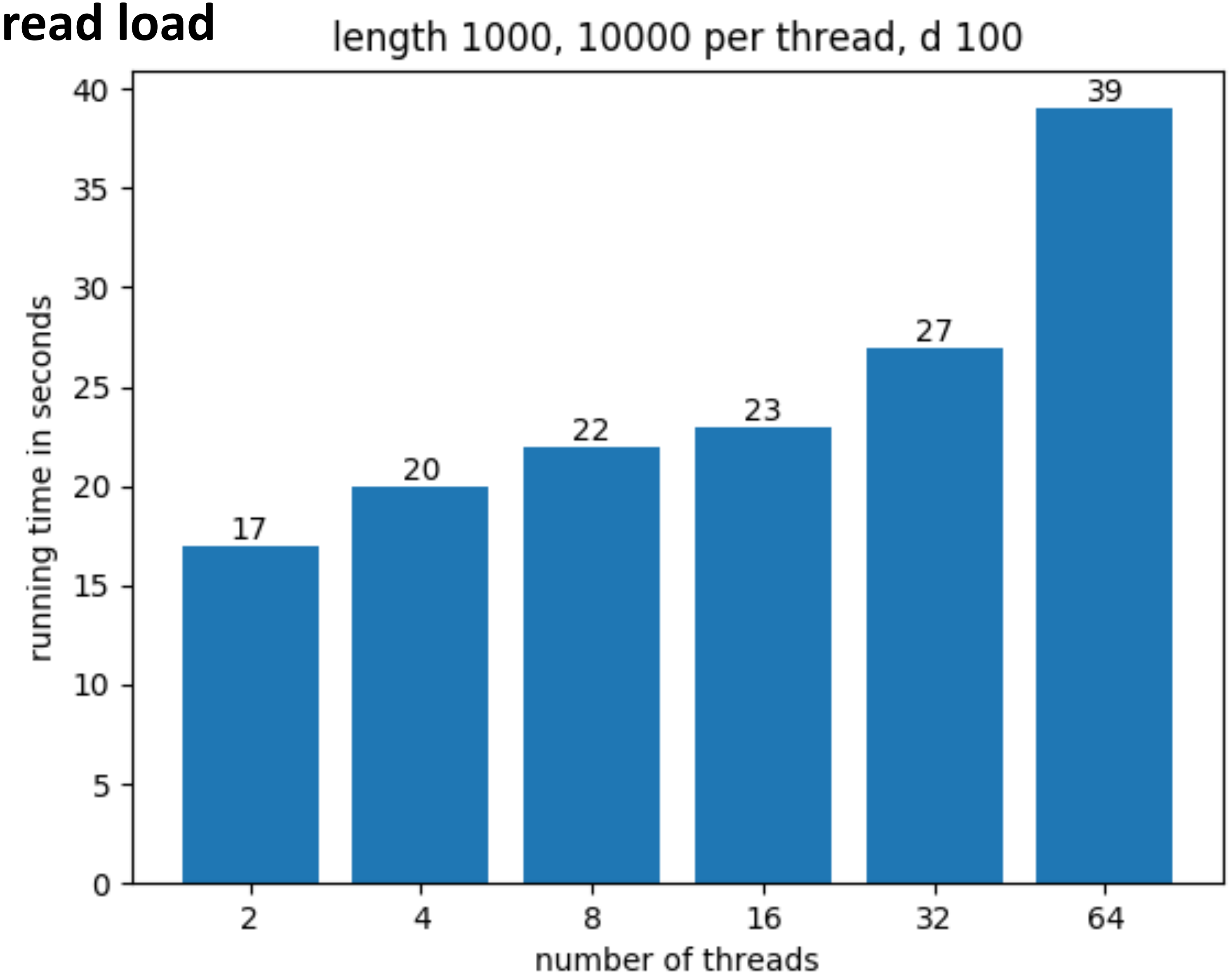
Result fix total



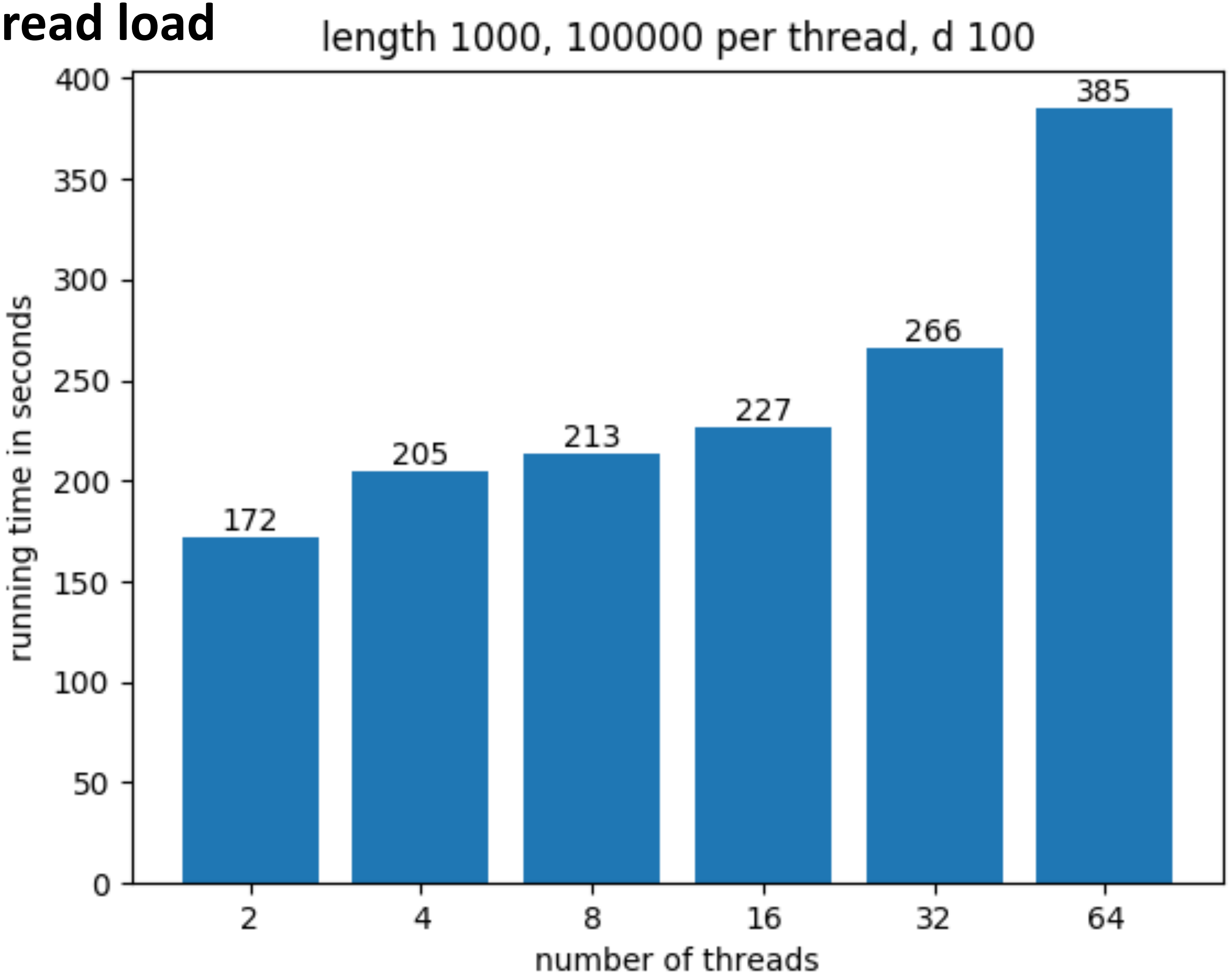
Result fix total



Result fix thread load



Result fix thread load



Thanks