Time Series Subsequence Similarity Search under Dynamic Time Warping Distance on the Intel Many-core Accelerators*

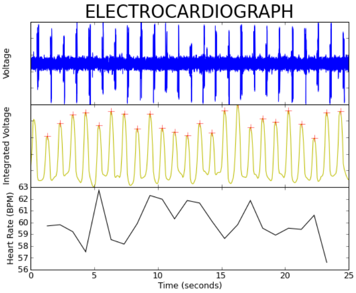
Aleksandr Movchan, Mikhail Zymbler

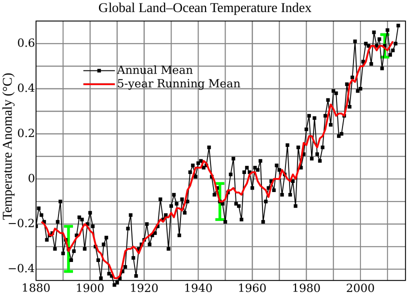
South Ural State University (Chelyabinsk, Russian Federation)

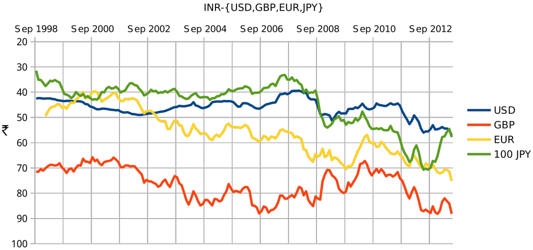
SISAP 2015, 8th International Conference
on Similarity Search and Applications
Glasgow, Scotland, UK, October 12-14, 2015

* This work was financially supported by the Ministry of education and science of Russia ("Research and development on priority directions of scientific-technological complex of Russia for 2014-2020" Federal Program, contract No. 14.574.21.0035).

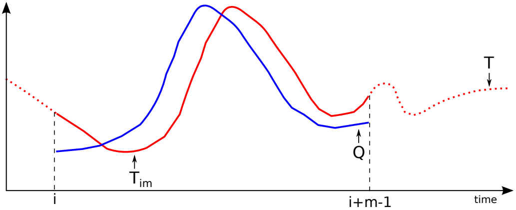
Time Series in Real Life





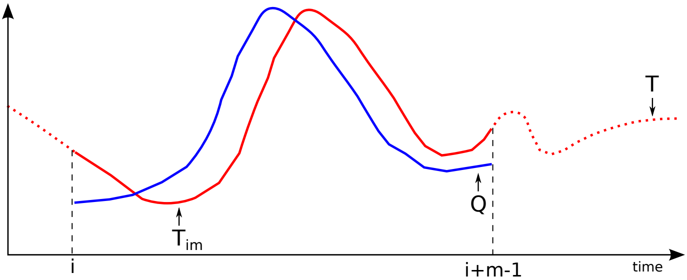


Formal Definitions



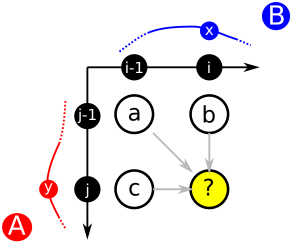
- *Time series* T
  - $T = t_1, t_2, \ldots, t_N$ where $t_i \in \mathbb{R}$
  - N is a length of the sequence
- *Query* Q
  - Q is a time series to be found in T
  - n is a length of the query, $n \ll N$
- *Subsequence* $T_{i\,m}$
  - $T_{i\,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$
  - $1 \leq i \leq N$ and $i + m \leq N$

Best-match Search



- Find $T_{i\,n} \in T$
  - $\forall m, 1 \leq m \leq N - n, D(T_{i\,n}, Q) < D(T_{m\,n}, Q)$
- D is a *similarity measure*.

DTW Similarity Measure

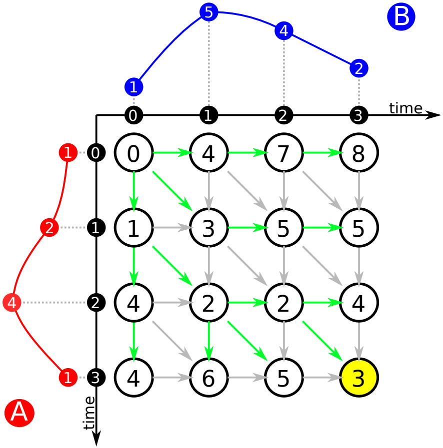


$$d(0,0) = 0$$

$$d(i,j) = |x - y| + min \begin{cases} d(i-1,j) \\ d(i,j-1) \\ d(i-1,j-1) \end{cases}$$

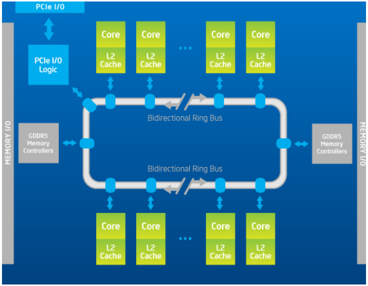
$$= |x - y| + min(a,b,c)$$

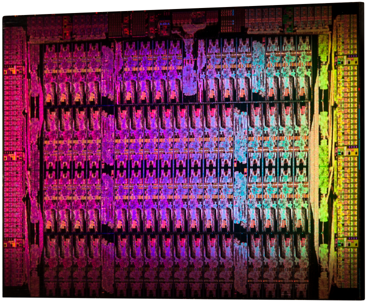
DTW(A, B) = d(N, N)

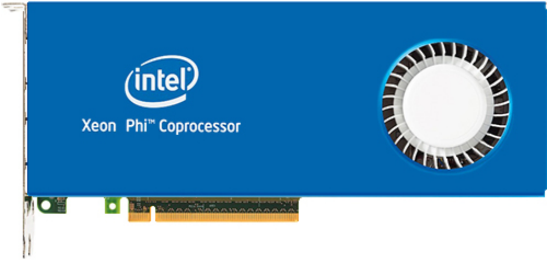


DTW(A, B) = d(3, 3) = 3

Intel Xeon Phi Architecture



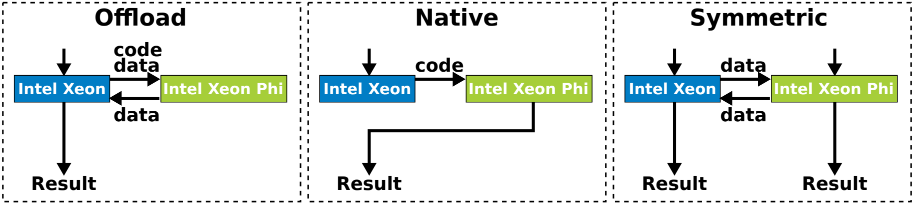




61 core, 244 threads, ≈1.2 TFLOPS, 512-bit SIMD

Intel Xeon Phi Programming Model

- Intel Xeon Phi supports the same parallel programming tools and models as x86 CPU
- Execution modes

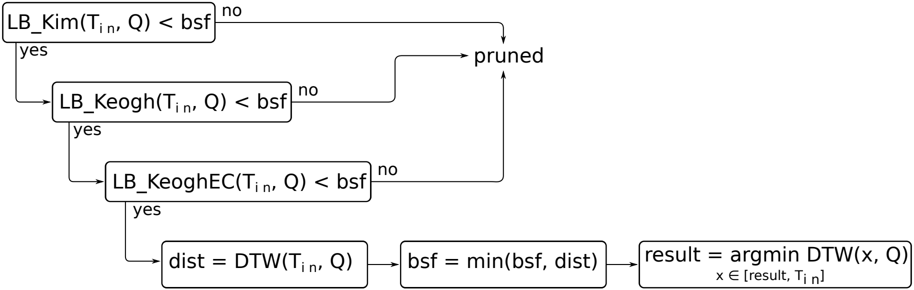


Execution on the CPU, offloading computationally intensive part of work to the coprocessor.

Independent execution on the coprocessor.

Execution on the coprocessor as MPI process.

UCR-DTW Serial Algorithm



Proposed in

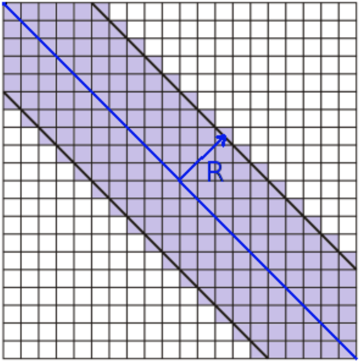
Rakthanmanon T., et al. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping // ACM SIGKDD, 2012. P. 262–270.

UCR-DTW Serial Algorithm

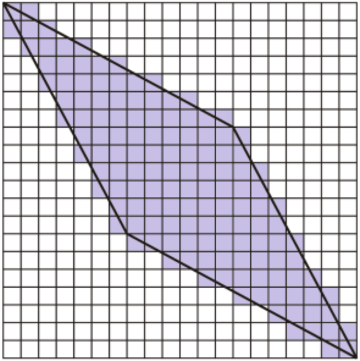
Features

- Dynamic Time Warping as similarity measure
- Exact search
- Z-normalization
  $x_i' = \frac{x_i - \mu}{\sigma}, i \in N,$
  μ – mean, σ – standard deviation
- Possible to search in large time series
- High level of data parallelism
- One of the fastest

DTW Restrictions



Sakoe-Chiba band



Itakura parallelogram

DTW Bounds

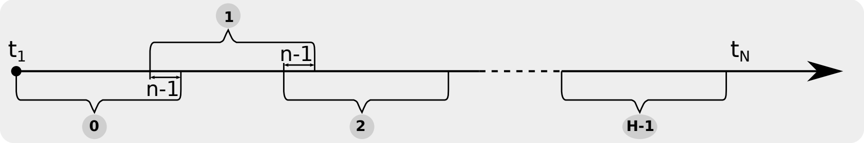
- $LB_{Kim} = \sqrt{(t_0 - q_0)^2 + (t_{n-1} - q_{n-1})^2}$
  Complexity: $O(1)$.
- LB_{Keogh}
  Sequences U and L are constructed for query Q
  $u_i = max(q_{i-R}, q_{i+R})$, $l_i = min(q_{i-R}, q_{i+R})$,
  $$LB_{Keogh}(Q, C) = \sqrt{\sum_{i=1}^{n} \begin{cases} (c_i - u_i)^2 & \text{if } c_i > u_i \\ (c_i - l_i)^2 & \text{if } c_i < l_i \\ 0 & \text{otherwise} \end{cases}}$$
  Complexity: $O(n)$.
- $LB_{KeoghEC}$
  Sequences U and L are constructed for subsequence C
  $u_i = max(c_{i-R}, c_{i+R})$, $l_i = min(c_{i-R}, c_{i+R})$,
  $$LB_{Keogh}(Q, C) = \sqrt{\sum_{i=1}^{n} \begin{cases} (q_i - u_i)^2 & \text{if } q_i > u_i \\ (q_i - l_i)^2 & \text{if } q_i < l_i \\ 0 & \text{otherwise} \end{cases}}$$
  Complexity: $O(n)$.

Parallelization Roadmap

I Parallel Algorithm for CPU
- Parallelize UCR-DTW using OpenMP
- Run parallel application on Xeon Phi only using *native* mode

II Parallel Algorithm for CPU and Coprocessor
- Parallel algorithm, combining CPU and Xeon Phi
  - coprocessor computes DTW
  - CPU prunes dissimilar subsequences and sends rest subsequences to the Xeon Phi
- Run parallel application on CPU and on coprocessor using *offload* mode

Splitting Time Series Among Threads



- T is partitioned into H equal-length *segments*

$$H = \lceil \frac{N}{P \cdot S} \rceil \cdot P$$

  where
  P is the number of OpenMP-threads,
  S is a max length of segment (parameter of the algorithm, e.g. $S = 10^6$),
  $n \ll S < N$

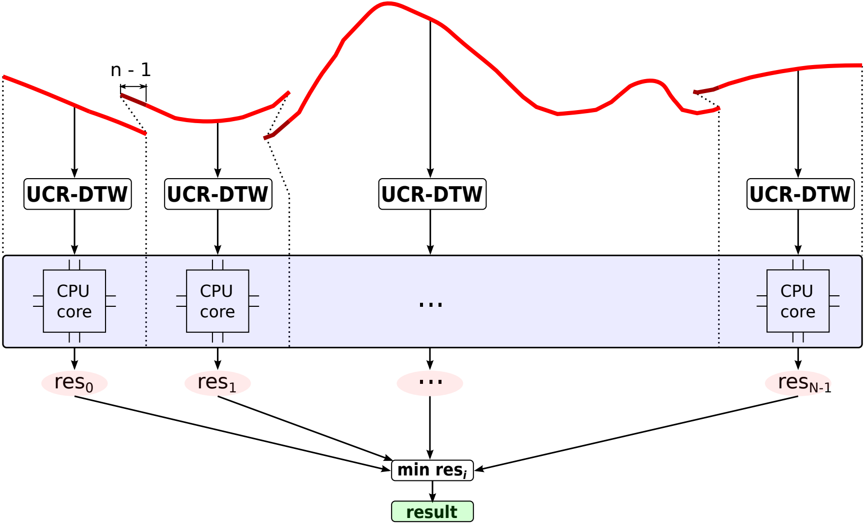
- k-th segment, $0 \le k \le H-1$, is a subsequence T_{sl}

$$s = \begin{cases} 1 & , k = 0 \\ k \cdot \lfloor \frac{N}{H} \rfloor - n + 2 & , else \end{cases}$$

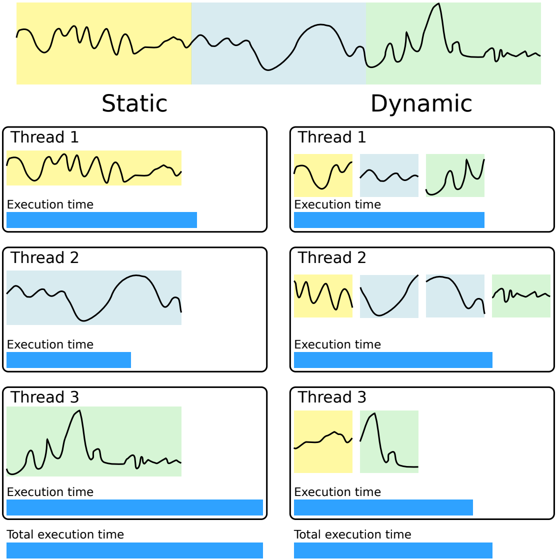
$$l = \begin{cases} \lfloor \frac{N}{H} \rfloor & , k = 0 \\ \lfloor \frac{N}{H} \rfloor + n - 1 + (N \ mod \ H) & , k = H-1 \\ \lfloor \frac{N}{H} \rfloor + n - 1 & , else \end{cases}$$

  where n is length of the query

Parallel Algorithm for CPU



Dynamic vs Static Distribution

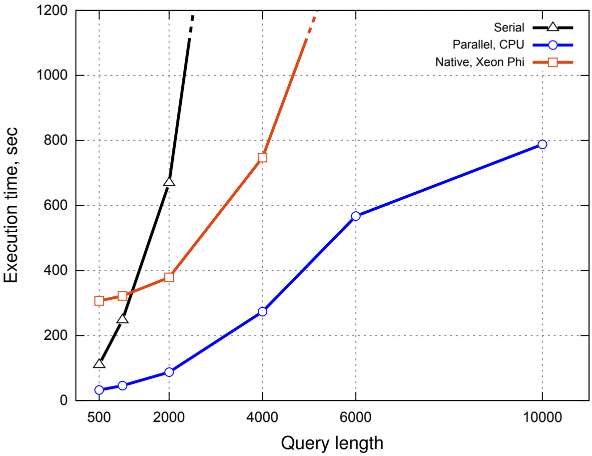


Performance of the Parallel Algorithm for CPU

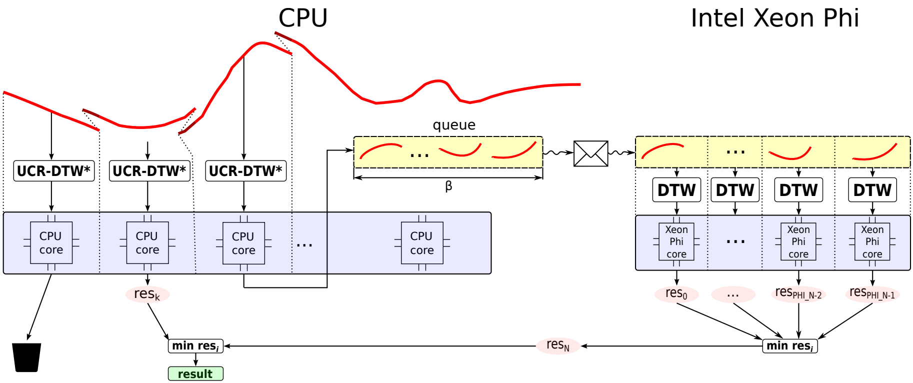
LB_Kim	$O(1)$
LB_Keogh	$O(n)$
LB_KeoghEC	$O(n)$
DTW	$O(n^2)$

Time of loading data from disk into memory of Intel Xeon Phi: $\approx 300\ s$

Data set: RANDOM WALK, 10^8 datapoints



Parallel Algorithm for CPU and Coprocessor



Experiments: Hardware

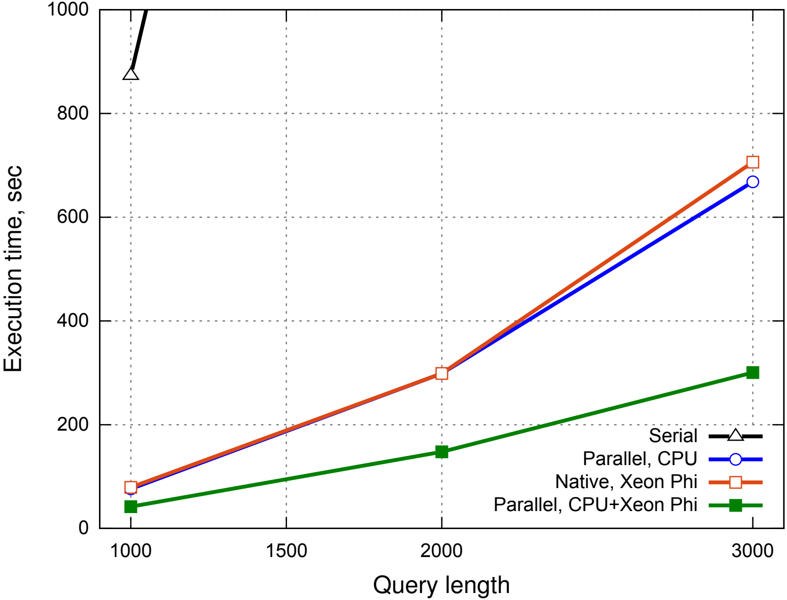
Specifications	Processor	Coprocessor
Model	Intel Xeon X5680	Intel Xeon Phi SE10X
Cores	6	61
Frequency, GHz	3.33	1.1
Threads per core	2	4
Peak performance, TFLOPS	0.371	1.076
Memory, Gb	24	8
Cache, Mb	12	30.5

Experiments: Data Sets

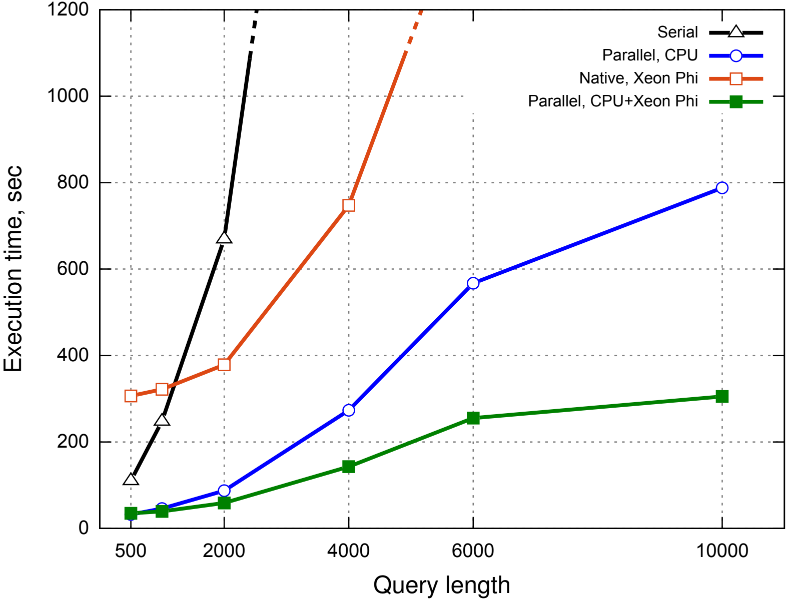
Time series	Category	Length
PURE RANDOM	synthetic	10^6
RANDOM WALK	synthetic	10^8
ECG*	real	$2 \cdot 10^7$

* Rakthanmanon T., et al. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping // ACM SIGKDD, 2012. P. 262–270.

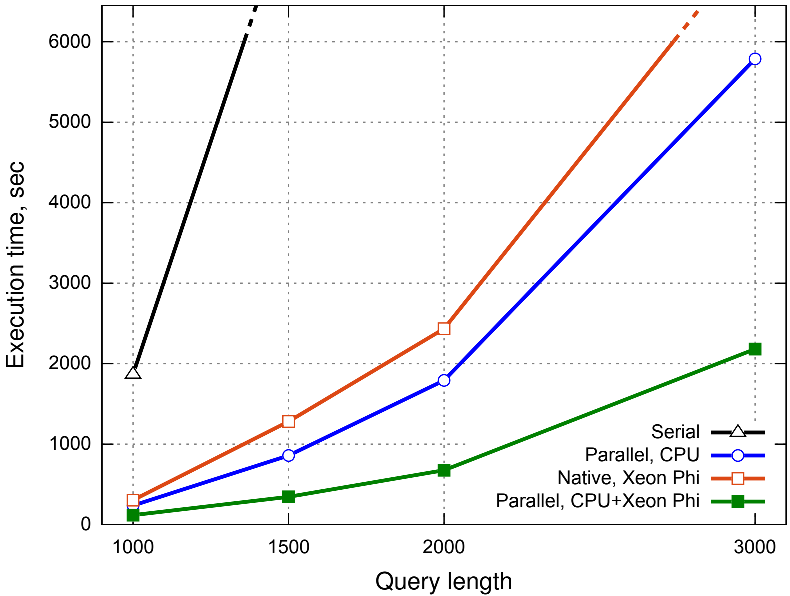
Performance – PURE RANDOM



Performance – RANDOM WALK



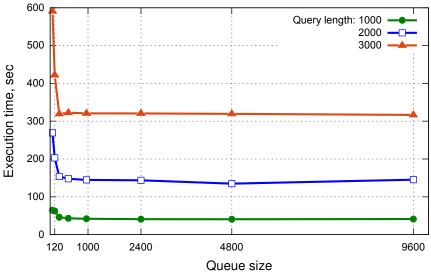
Performance – ECG



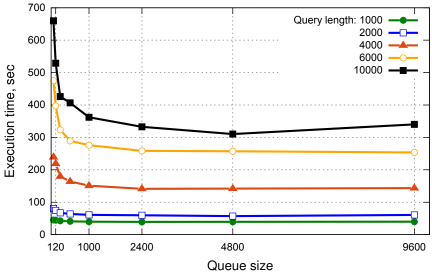
Impact of Queue Size on the Speedup

Queue size $= C \times h \times W$
where
C — the number of available cores of the coprocessor,
h — hyperthreading factor of the coprocessor,
W — the number of candidates to be processed by a coprocessor's thread.

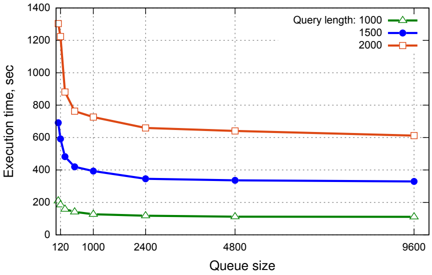
Impact of Queue Size on the Speedup



(a) PURE RANDOM

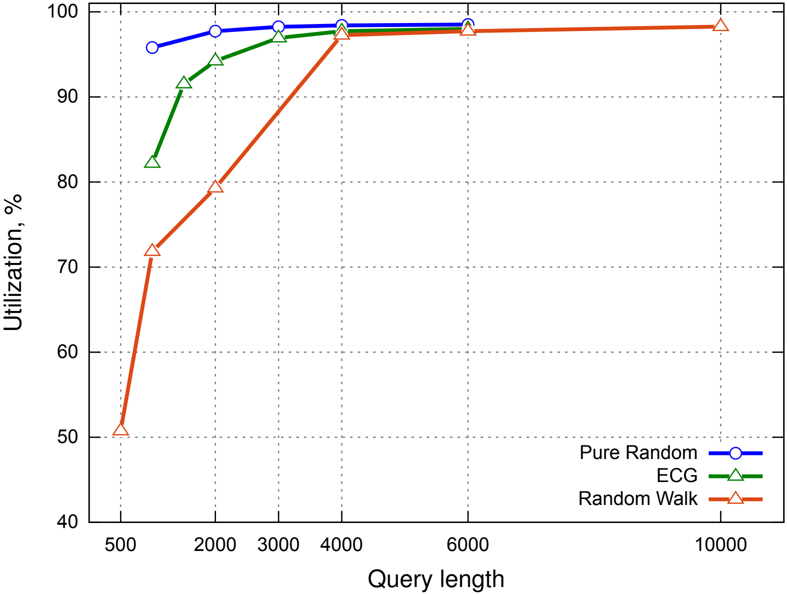


(b) RANDOM WALK

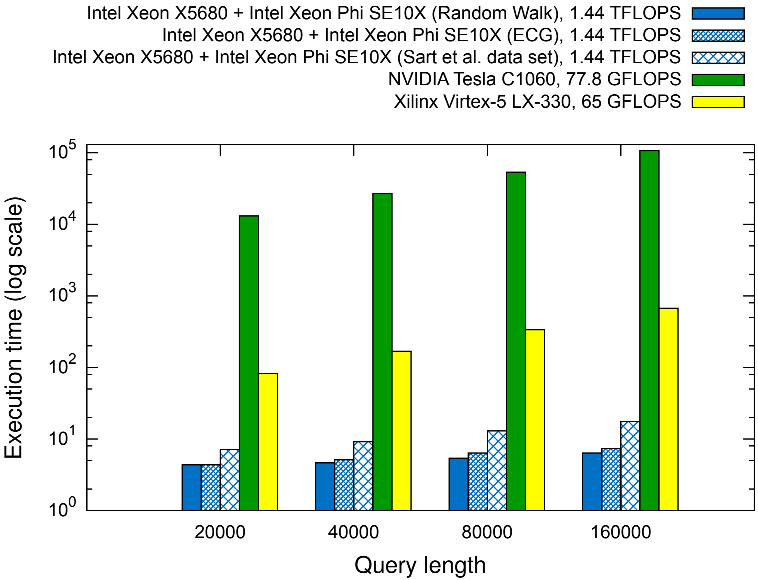


(c) ECG

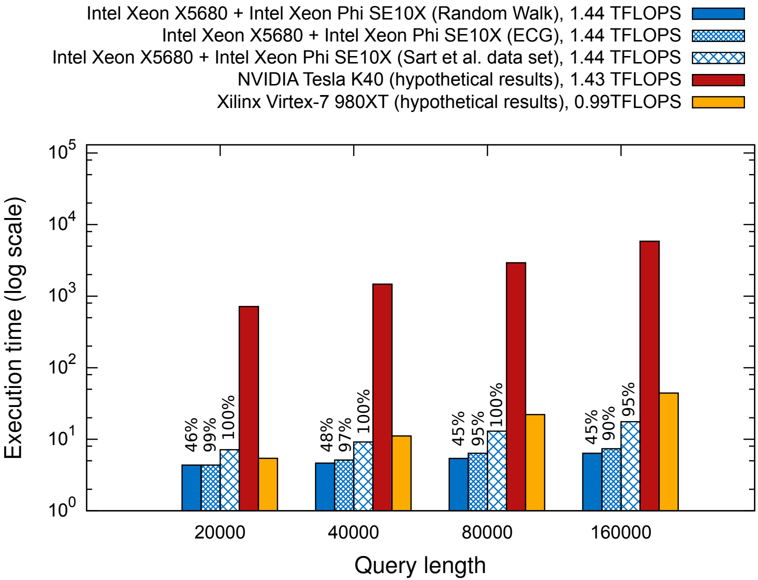
Utilization of Coprocessor



Comparison with Analogues



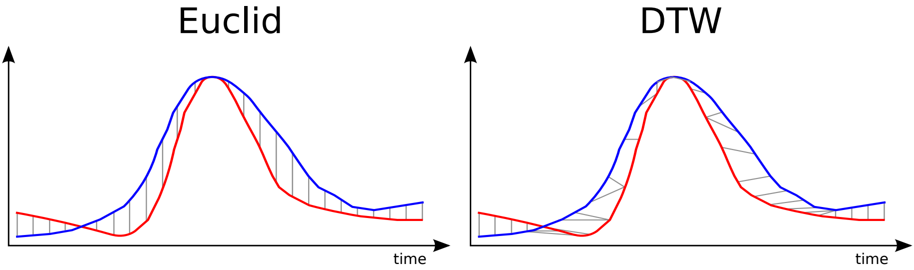
Comparison with Analogues



Conclusion

- A parallel algorithm for best-match time series subsequence search under DTW distance on the Intel Many Integrated Core has been presented.
- The algorithm combines capabilities of CPU and the Intel Xeon Phi
  - the coprocessor is exploited only for DTW computations;
  - CPU performs lower bounding, prepares subsequences for the coprocessor;
  - CPU supports a queue of candidate subsequences and the coprocessor computes DTW for each candidate.
- Experiments have shown that the algorithm does not concede to analogous algorithms for GPU and FPGA on performance.
- Future work: extend the algorithm for the following cases:
  - implement modified DTW, based on the wavelet transform;
  - several the Intel Xeon Phi coprocessors;
  - cluster computing system with nodes equipped with a the Intel Xeon Phi coprocessor(s).

How to Compute DTW

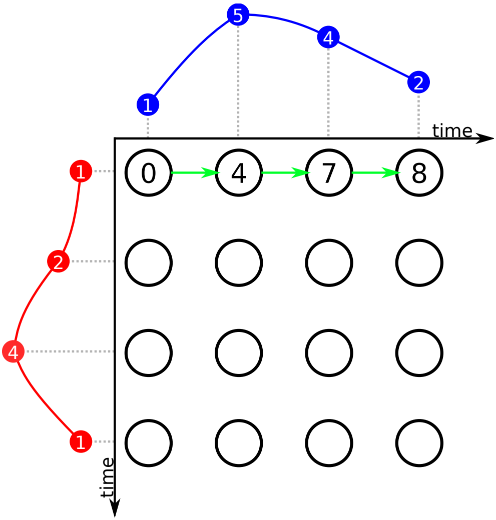


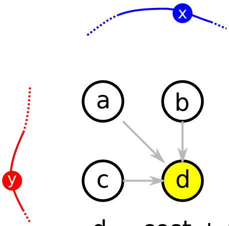
$$DTW(X,Y) = d(N,N),$$

$$d(i,j) = |x_i - y_j| + min \begin{cases} d(i-1,j) \\ d(i,j-1) \\ d(i-1,j-1), \end{cases}$$

$$d(0,0) = 0; d(i,0) = d(0,j) = \infty; i = 1,2,\ldots,N; j = 1,2,\ldots,N.$$

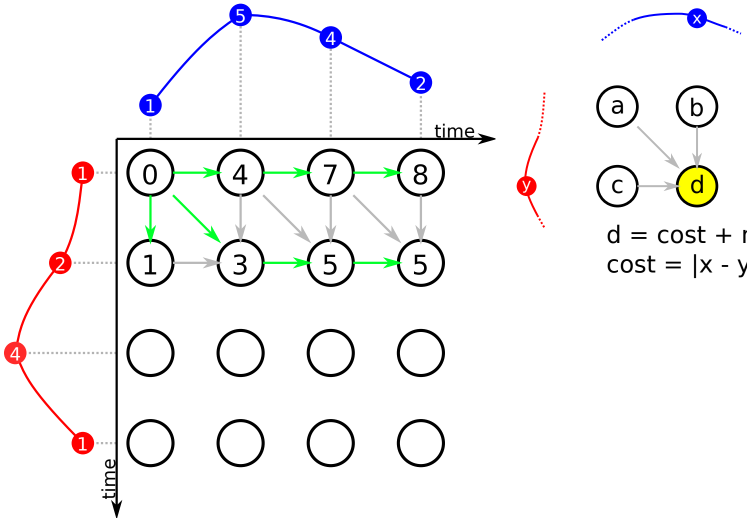
How to Compute DTW





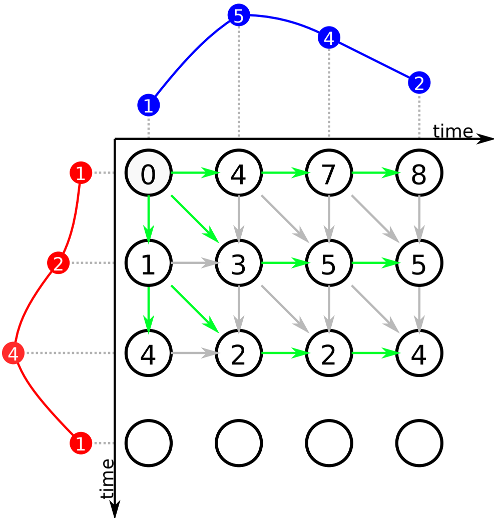
d = cost + min(a, b, c)
cost = |x - y|

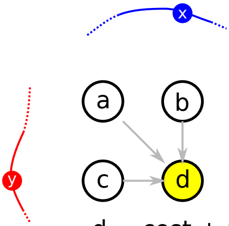
How to Compute DTW



d = cost + min(a, b, c)
cost = |x - y|

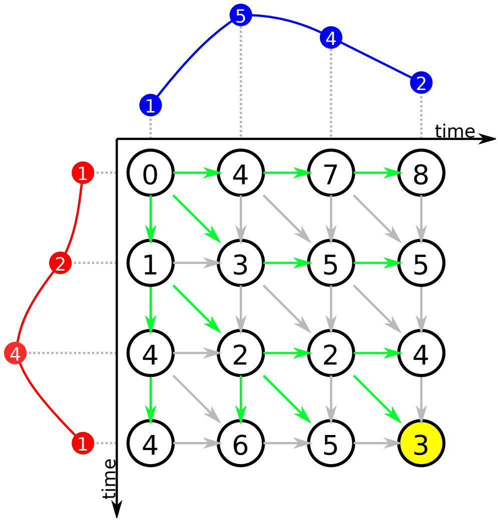
How to Compute DTW

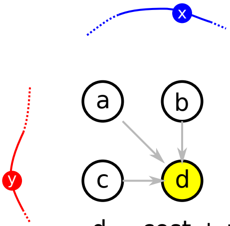




d = cost + min(a, b, c)

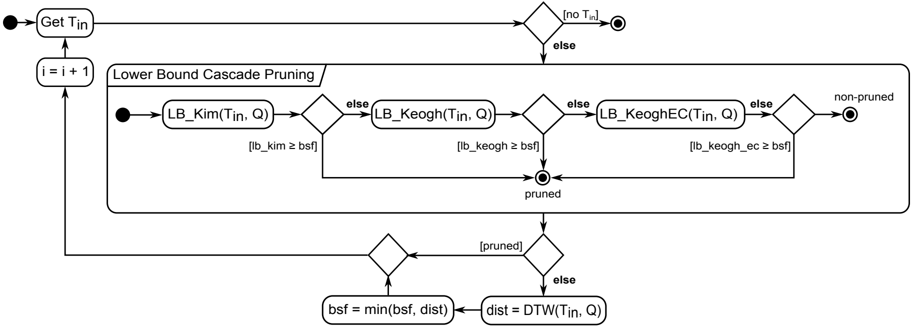
cost = |x - y|



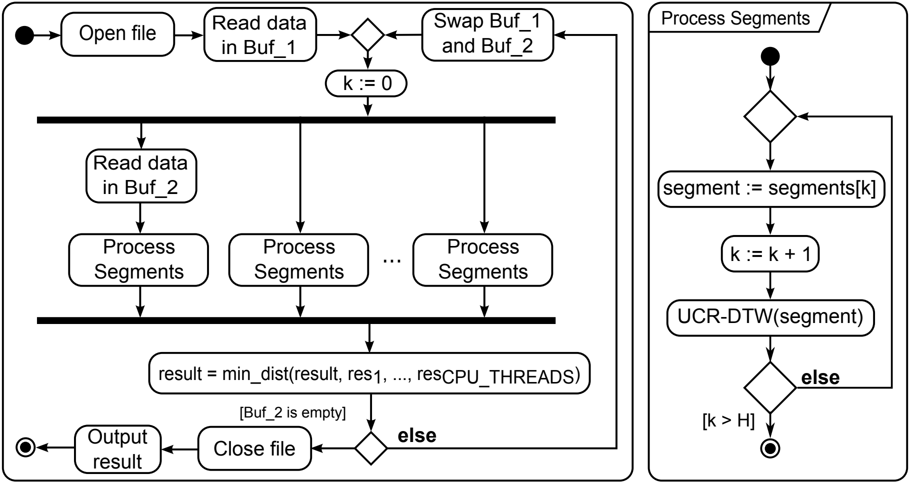


d = cost + min(a, b, c)
cost = |x - y|

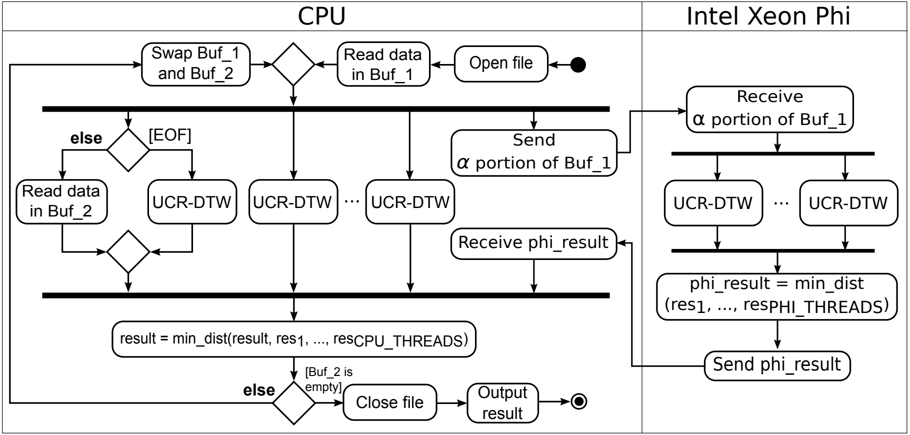
Serial Algorithm



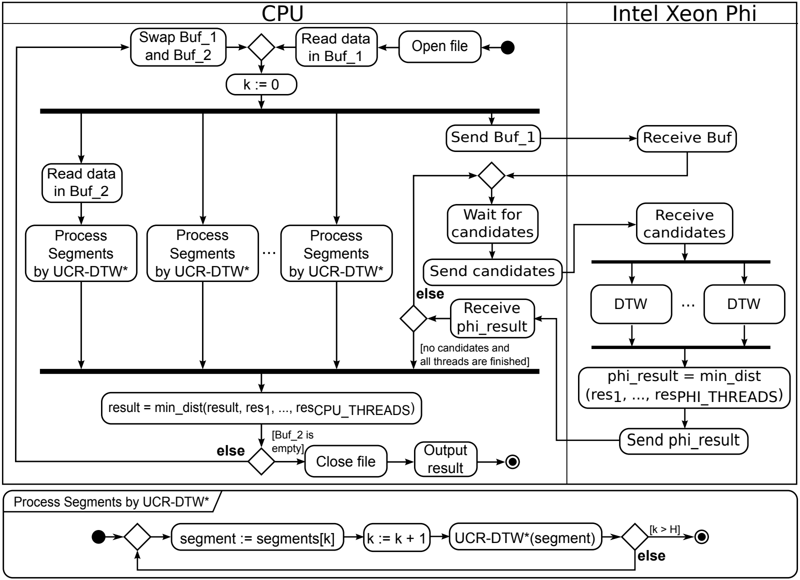
Simple Algorithm



Naïve Algorithm



Advanced Algorithm



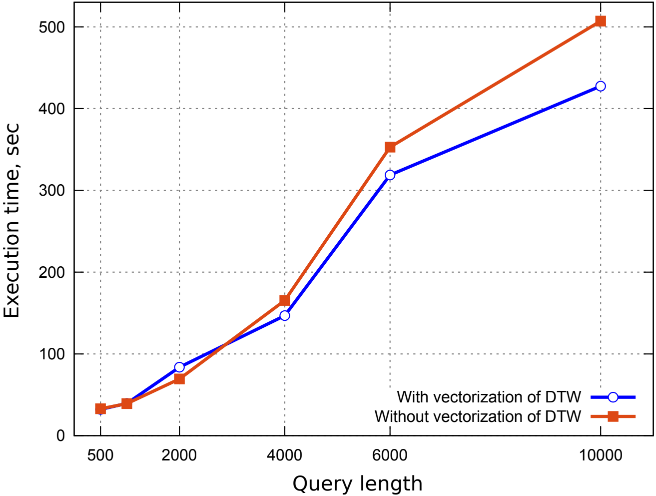
Before vectorization of DTW

```
double DTW(a: array [1..m], b: array [1..m], r: int) {
   cost := array [1..m]
   cost_prev := array [1..m]

   for i := 1 to m
      cost[i] = infinity
      cost_prev[i] = infinity

   cost_prev[1] = dist(a[1], b[1])

   for j := max(2, i-r) to min(m, i+r)
      cost_prev[j] := cost_prev[j-1] + dist(a[1], b[j])

   for i := 2 to m
      for j := max(1, i-r) to min(m, i+r)
         c := d(a[i], b[j])
         cost[j] := c + min(cost[j-1], cost_prev[j-1], cost_prev[j])
      swap(cost, cost_prev)

   return cost_prev[m]
}
```

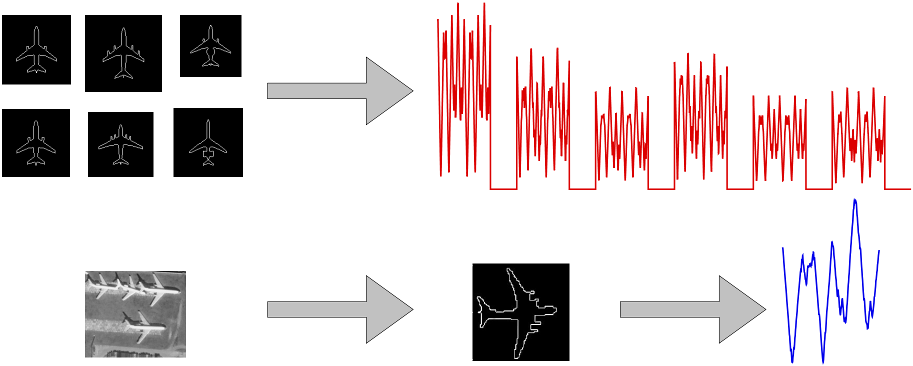
After vectorization of DTW

```
double DTW(a: array [1..m], b: array [1..m], r: int) {
   cost := array [1..m]
   cost_prev := array [1..m]
   for i := 1 to m
      cost[i] = infinity
      cost_prev[i] = infinity
   cost_prev[1] = dist(a[1], b[1])
   for j := max(2, i-r) to min(m, i+r)
      cost_prev[j] := cost_prev[j-1] + dist(a[1], b[j])
   for i := 2 to m
      for j := max(1, i-r) to min(m, i+r)
         cost[j] = min(cost_prev[j-1], cost_prev[j])
      for j := max(1, i-r) to min(m, i+r)
         c := dist(a[i], b[j])
         cost[j] := c + min(cost[j-1], cost[j])
      swap(cost, cost_prev)
   return cost_prev[m]
}
```

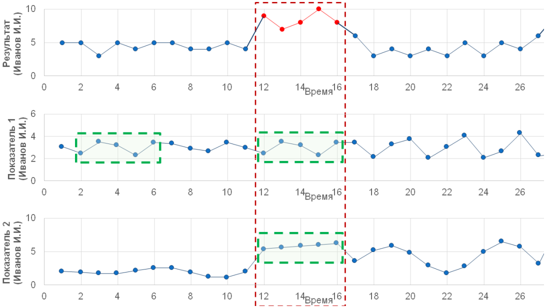
Impact of vectorization of DTW



Classification of Contours



MedMining Project



Data mining of physiological studies of professional athletes