jh220050

Implementation and Application of High-Performance Empirical Dynamic Modeling

Keichi Takahashi (Tohoku University)

Abstract

This research aims at developing a high-performance implementation of Empirical Dynamic Modeling (EDM), an emerging framework for non-linear time series analysis. EDM enables a variety of analyses such as short-term forecasts, quantification of non-linearity, and causal inference. Although EDM is a generic modeling method for time series data, it was originally developed in the field of ecology, where available datasets are relatively small. Thus, the current libraries for EDM are not designed with performance in mind, and the scale of datasets that can be analyzed are limited. To enable large-scale analysis using EDM, we have been developing a high-performance EDM implementation. In this research, we continue this effort by (1) porting time-consuming kernels in EDM to the SX-Aurora TSUBASA Vector Engine (2) enhancing the scalability of EDM by utilizing approximate algorithms (3) analyzing neural activity datasets to evaluate the performance of the ported implementation.

1 Basic information

1.1 Collaborating JHPCN centers

Tohoku University

1.2 Theme area

Data science/data usage area

1.3 Research area

- Very large-scale data processing
- Very large-scale numerical computation

1.4 Project members and their roles

- Keichi Takahashi, Tohoku University, Japan (Administration and code development)
- Gerald M. Pao, Salk Institute for Biological Studies, USA and Okinawa Institute of Science and Technology, Japan (Algorithm design and analysis of results)
- Wassapon Watanakeesuntorn, Nara Institute of Science and Technology, Japan (Performance evaluation)

2 Purpose and significance of the research

This research aims at developing a high-performance implementation of *Empirical Dynamic Modeling (EDM)*, an emerging framework for non-linear time series analysis, and applying it to large-scale datasets. EDM enables a variety of analyses such as short-term forecasts, quantification of non-linearity, and causal inference. These analyses are achieved by reconstructing the latent

dynamics behind the data without assuming a parametric model or using prior knowledge (Figure 1).

Although EDM is a generic modeling method for time series data, it was originally developed in the field of ecology. *De facto* standard tools and libraries for EDM analysis are therefore designed to target a small number of short time series. While recent studies have successfully applied EDM to diverse datasets, the lack of a high-performance implementation is limiting the scale of datasets that can be analyzed. To enable large-scale analysis using EDM, we have been developing a high-performance EDM implementation. In this research, we continue this effort by tackling the following three challenges:

1. Porting our EDM implementation to the SX-Aurora TSUBASA Vector Engine.
2. Enhancing the scalability of EDM by utilizing approximate algorithms.
3. Analyzing neural activity datasets to evaluate the performance of the ported implementation.

3 Significance as JHPCN Joint Research Project

This research closely aligns with the goals of JHPCN joint research. It is carried out by an international and interdisciplinary team of scientists. The representative has been working with HPC infrastructure and HPC application optimization, while the deputy representative has long experience in the field of quantitative biology, neuroscience, and nonlinear dynamics. In particular, Pao has been collaborating with George Sugihara, one of the original developers of EDM.

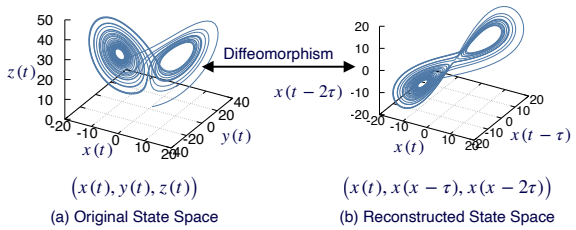


Fig. 1: State Space Reconstruction

4 Outline of Research Achievements up to FY2021

N/A

5 Details of FY2022 Research Achievements

5.1 Porting EDM to the SX-Aurora TSUBASA Vector Engine

5.1.1 Overview

We ported the most time-consuming and critical component of the EDM algorithm, which is the k-nearest neighbor (k-NN) search in the state space, to the SX-Aurora Vector Engine and conducted a preliminary performance evaluation. We implemented an exact k-NN search using a brute-force approach. That is, we first (1) compute the pairwise distance matrix between all points in the state space, and then (2) sort the distance matrix to find the nearest neighbors for each point. The outcome of this section has been presented in [2] and [4].

5.1.2 Compared algorithms

Sine the pairwise distance calculation achieves high performance comparable to that of GPUs, and shows both high vector-

ization rate and average vector length, we focused on the sorting of the distance matrix and compared the following five approaches:

1. C++ STL sort (introsort)
2. C++ STL partial sort (heap)
3. NEC ASL sort (radix sort, vectorized)
4. LSD radix sort (radix sort, vectorized)
5. MSD radix partial sort (vectorized)

C++ STL sort (introsort): This is the sorting routine provided by C++ Standard Template Library (STL) as `std::sort`. It uses the introsort algorithm, where the input array is recursively partitioned using quicksort, and once each partition becomes smaller than a threshold, it is sorted using insertion sort.

C++ STL partial sort (heap): This the partial sorting routine provided by C++ STL as `std::partial_sort`. The algorithm works by scanning over the input array and incrementally updating a max-heap of size k. If an array element is smaller than the maximum element in the heap, the maximum element is element is removed from the heap and the new element is inserted into the heap. If the array element is larger than the maxmium element in the heap, it is ignored.

NEC ASL sort (radix sort, vectorized): This a vectorized sort algorithm provided by NEC's proprietary ASL library (`https://sxauroratsubasa.sakura.ne.jp/documents/sdk/SDK_NLC/UsersGuide/asluni/f/en/index.html`). The documentation provided by NEC mentions it uses radix sort, but the detailed algorithm is not described and unknown.

LSD radix sort (radix sort, vectorized): This is a vectorized radix sort implemented by NEC and open-sourced on GitHub (`https://github.com/SX-Aurora/radixsort_vec`). The algorithm looks at a block of contiguous bits and from the most significant digit to the least significant digit. At each iteration, a "digit" (a contiguous blocks of bits) in the binary representation of each array element is extracted, and the array elements are put into bins corresponding to different digits. The elements are then reorganized using a stable counting sort. This is repeated until all digits are processed.

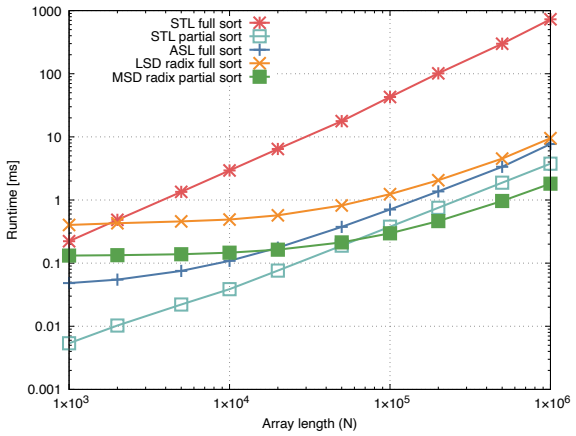


Fig. 2: Partial sort runtime on VE Type 20B

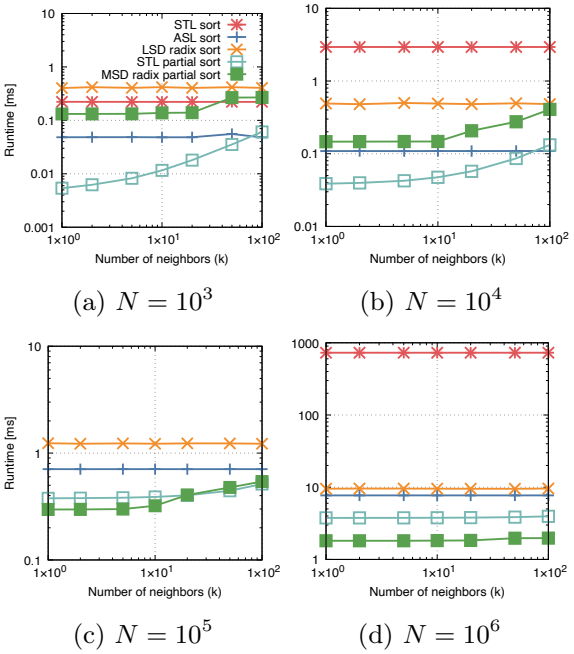
MSD radix partial sort (vectorized): This is our implementation of a vectorized radix partial sort. This is similar to the LSD radix sort, but the digits are scanned from the most-significant to the least-significant bits. Furthermore, only the elements in the bins that contain the top k elements are carried over to the next iteration. This way, the number of elements to be processed is reduced at each iteration.

5.1.3 Evaluation results

Figure 2 compares the runtime of the five algorithms with respect to the array length. Clearly, STL sort is the slowest one among all algorithms, because it is not vectorized at all. The next slowest is LSD radix sort. Although this implementation is vectorized and also provided by NEC, it is slower than any of the vectorized algorithms. The fastest one among algorithms that sort the whole array (*i.e.,* not partial sort) is the ASL sort. This algorithm is consistently faster than the LSD radix sort, and even faster than MSD radix partial sort if the array length is below 20,000.

Surprisingly, the unvectorized STL partial sort is faster than the vectorized MSD radix partial sort for shorter arrays. Specifically, the STL partial sort is the fastest if the input array is shorter than 100,000. This is likely because the MSD radix sort requires multiple passes over the input array, while the STL partial sort only requires a single pass over the input. As the array becomes longer, vectorization pays off. Therefore, a hybrid approach to use STL partial short arrays and MSD radix sort for long arrays seems to be promising.

Figure 3 compares the runtime of the algorithms with respect to the number of nearest neighbors to find. As expected, the runtime of STL sort, ASL sort and LSD radix sort are constant because they sort the entire array regardless of k. The runtime of both the STL partial sort and MSD radix partial sort increase with respect to k. However, the increase in runtime decreases as the array becomes longer. When the array length is 10^6, we see almost no increase in k.



(a) $N = 10^3$ (b) $N = 10^4$

(c) $N = 10^5$ (d) $N = 10^6$

Fig. 3: Top-k sorting runtime on VE Type 20B (varying k)

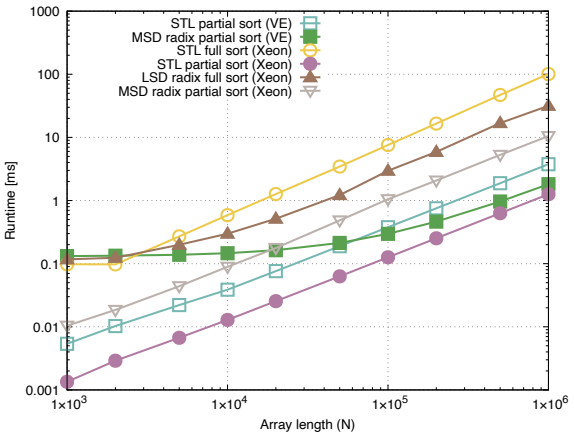


Fig. 4: Comparison of partial sort runtime on VE Type 20B and Xeon Silver 4208

Lastly, Figure 4 compares the runtime of sorting algorithms on VE Type 20B and Xeon Silver 4208. The plot shows that the sorting performance on VE is generally poor

compared to Xeon, none of the sorting algorithms on VE surpassed the performance of STL partial sort on Xeon. We are therefore considering to run the distance calculation on the Vector Engine and the sorting on the CPU.

5.2 Applying approximate algorithms to EDM

5.2.1 Overview

Even if accelerators such as VE and GPU are employed, brute-force k-NN search is not scalable since it has quadratic time and space complexity with respect to the number of points. We therefore investigated the feasibility of using approximate k-NN search algorithms to make EDM scalable. Specifically, we integrated the Faiss[*1] vector similarity search library into our implementation and evaluated the runtime and accuracy trade-off. The outcome of this section is currently under review [1].

We compared following three different approximate k-NN search algorithms in addition to brute force search on CPU and GPU:

1. Inverted File Index (IVF): This algorithm first clusters the data points into multiple clusters. Then, a brute-force search is executed within the cluster closest to the query point.
2. k-dimensional Tree (k-d Tree): This algorithm recursively partitions the space into cells using hyperplanes. Given a query point, the tree is traversed and only points contained in a small number of cells are searched to find the nearest neighbors.

[*1] `https://faiss.ai/`

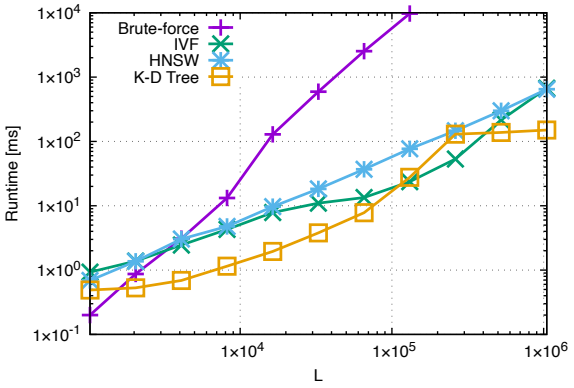


Fig. 5: Simplex projection runtime with different approximate k-NN search algorithms ($E = 1$)

3. Hierarchical Navigable Small World (HNSW): This is a state-of-the-art graph-based approximate k-NN search algorithm based on the Navigable Small World (NSW) graphs. NSW graphs are graphs that contain both long-range and short-range edges, and there exist a path between any two vertices with a polylogratihmic number of edges with respect to the number of vertices.

5.2.2 Evaluation results

We integrated the above-mentioned three approximate k-NN search algorithms into our implementation of EDM [3], and measured the runtime of Simplex projection, a short-term forecast method, under different conditions.

Figure 5 shows the runtime of Simple projection when the embedding dimension (the dimension of the reconstructed state space) is one. Evidently, the runtime of brute-force k-NN search grows rapidly due to its quadratic time complexity and becomes quickly pro-

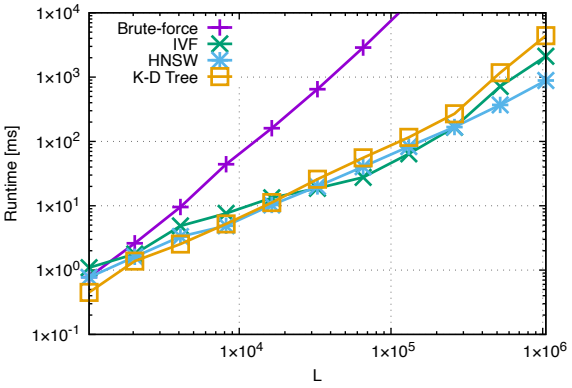
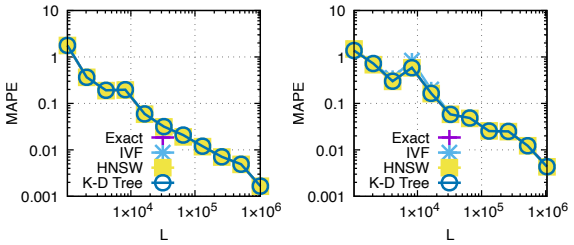


Fig. 6: Simplex projection runtime with different approximate k-NN search algorithms ($E = 20$)



(a) $E = 1$ (b) $E = 20$

Fig. 7: MAPE of Simplex projection with different approximate k-NN search algorithms

hibitive. Out of the approximate algorithms, k-d Tree is the fastest for most time series length. The speedup of k-d Tree over brute-force reaches 3406× when the time series length is 2^{20}. Figure 6 shows the runtime when the embedding dimension is 20. Here, k-d Tree is non longer the fastest and falls behind IVF and HNSW. This is a known property of k-d Tree that it suffers with high-dimensional data. The speedup of HNSW over brute-force reaches 775× when the time series length is 2^{20}. In conclusion, using k-d Tree for low embedding dimension and HNSW for high embedding dimension seems to be reasonable choice.

We also tested the accuracy of Simplex projection when using approximate k-NN search algorithms instead of exact k-NN search. Figure 7 shows the Mean Absolute Percentage Error (MAPE) of Simplex projection with different k-NN search algorithms. Surprisingly, the change in MAPE is almost ignorable even when an approximate search is used. This indicates that Simplex projection is highly robust to incorrect neighbors.

5.3 Analyzing neural activity datasets

We obtained several neural activity datasets from our collaborators. These datasets contain whole brain scale neural activity at single-neuron resolution. Specifically, they were recorded from a genetically modified larval zebrafish using light sheet fluorescence microscopy. The datasets include up to 10,000 time steps and 100,000 time series. We applied Convergent Cross Mapping (CCM), a causal inference algorithms in EDM, to these datasets and obtain a map of causal interactions between every pair of neurons. The outcome of this section has not been published yet.

6 Self-review of Current Progress and Future Prospects

As for the first challenge, we were able to port the performance-critical k-NN search kernel to the Vector Engine. However, the sorting performance was unsatisfactory even though we implemented several highly vectorized sorting algorithms. We will continue

to analyze the reasons behind the efficiency and optimize them. Furthermore, we will investigate whether we can apply state-of-the-art sorting algorithms for GPUs. We will also explore a hybrid approach to run tine distance calculation on the Vector Engine and the sorting on the CPU.

As for the second challenge, we integrated several approximate k-NN search algorithms into our implementation of EDM [3] and evaluated the runtime and accuracy of Simplex projection. We showed that the use of approximate k-NN search can offer up to 3406× speedup with minimal loss in accuracy. This result opens up possibilities to apply EDM to extremely large-scale datasets that were intractable in the past. We plan to open-source the implementation in the near future.

As for the third challenge, we applied the Convergent Cross Mapping (CCM) causal inference algorithms to whole brain scale neural activity datasets and obtained a causal map of all neurons. We plan to apply the result of this project to other datasets in the biology field, such as spatiotemporal gene expression datasets.

7 List of publications and presentations

Journal Papers (Refereed)

[1] Keichi Takahashi, Kohei Ichikawa, Joseph Park, and Gerald M. Pao (+), "Scalable Empirical Dynamic Modeling with Parallel Computing and approximate k-NN Search," *IEEE Access*, Jan. 2023. (under review)

Proceedings of International Conference Papers (Refereed)

N/A

Presentations at International conference (Non-refereed)

[2] Keichi Takahashi and Gerald M. Pao (+), "Challenges in Scaling Empirical Dynamic Modeling," *The 34th Workshop on Sustained Simulation Performance (WSSP 34)*, Oct. 2022.

Presentations at domestic conference (Non-refereed)

N/A

Published open software library and so on

[3] Keichi Takahashi, kEDM, `https://github.com/keichi/kEDM`, 2023.

Other (patents, press releases, books and so on)

[4] Keichi Takahashi, Kohei Ichikawa, and Gerald M. Pao (+), "Toward Scalable Empirical Dynamic Modeling," *Sustained Simulation Performance 2022*, 2023. (in print)