Access Pattern Learning with Long Short-Term Memory for Parallel Particle Tracing

Fan Hong[1] Jiang Zhang[1] Xiaoru Yuan[1,2*]

1) Key Laboratory of Machine Perception (Ministry of Education), and School of EECS, Peking University, Beijing, China
2) Beijing Engineering Technology Research Center of Virtual Simulation and Visualization, Peking University, Beijing, China

Abstract

In this work, we present a novel access pattern estimation approach for parallel particle tracing in flow field visualization based on deep neural networks. With strong generalization ability, we develop a Long Short-Term Memory (LSTM)-based model, which is capable of learning accurate access patterns with only a few training samples and representing the learned patterns with small storage overhead. Equipped with prediction and prefetching functions driven by the developed model, our parallel particle tracing framework employs CPUs and GPUs together for particle tracing tasks. We demonstrate the accuracy and time efficiency of our approach with various flow visualization applications in three different flow datasets.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Scientific visualization; Computing methodologies—Machine learning—Machine learning approaches—Neural networks

1 Introduction

In flow visualization and analysis, particle tracing is one of the most fundamental techniques. Many applications require massive tracing of field lines, such as source-destination analysis [19], finite-time Lyapunov exponent (FTLE) computation [15], field line or surface rendering [8], etc. However, particle tracing is very computational-expensive, especially for large and complex unsteady flow fields. When the data cannot fit into the machine memory, tracers have to either load data from external storage or communicate with other tracers to exchange data. Existing studies have shown that I/O costs could take up to 90% of the computation time, due to the complicated and intricate data access patterns which are hard to predict.

To reduce I/O costs in particle tracing, one key idea is to improve data locality while advecting seeds [4–6]. Data access pattern modeling is one type of solutions for this purpose. Access dependencies between data blocks are usually considered, that is when a particle has already visited a block or some blocks, which block it probably moves to. After estimating the access dependencies, data locality can be improved with feasible techniques, such as file layout reorganization [4, 5], finer data management [14], data prefetching [29], etc. However, it is a non-trivial task to obtain an accurate estimation of the access dependencies.

Among the existing methods [4, 14, 29], frequency-based approaches are usually used. In the simplest case, these approaches assume the probability of accessing one data block only depends on the previously visited block, which is named as 1st-order dependencies [29] or 1-hop transitions [4]. When advecting a particle, 1st-order dependencies is capable of predicting the next visiting block based on the probability distribution bound to its current resided block. To make the prediction more accurate, high-order dependencies are considered, i.e. the probability of accessing one block depends on more than one passed blocks. Due to the lack of generalization, all valid cases have to be recorded with their transition probabilities, which causes a large storage overhead. As reported, the storage of 6th-order dependencies can be larger than the original data in certain flow fields [29].

In this work, we propose a new approach to model data access patterns with Long Short-Term Memory (LSTM) which has strong generalization capability. LSTM is a kind of recurrent neural networks (RNNs) [10, 17] which allow forward and backward connections between neurons. General RNNs are not suitable for handling long-term dependencies because of the gradient vanish problem [1]. Instead, LSTM architecture remembers values over arbitrary intervals. LSTM aims at learning representation with long-term dependencies from sequences for tasks such as speech recognition [30], machine translation [7], and text generation [27], and has already achieved incredible success. As for the access pattern estimation task in particle tracing, LSTM can naturally involve all historic footprints of particles as sequences and estimate their access dependencies. Our LSTM-based model can learn access dependencies with fewer sampled data and represent them with a neural network of smaller storage.

We further develop a parallel particle tracing framework with prediction and prefetching functions driven by our model. We sample several pathlines by placing seeds in the domain, and feed them to train our LSTM-based model. Data access dependencies are learned through the training process. In parallel particle tracing, our model is employed for data block prefetching to reduce I/O costs. Given the passed blocks of particles, the next blocks are predicted with a high accuracy and are prefetched in advance. In addition, since the trained model are deployed in GPUs, our framework is a new manner of combining CPUs and GPUs together for particle tracing tasks.

Although the basic idea of employing LSTM to learn from sequences is natural, the concrete solution for particle tracing tasks is non-trivial. Deep learning models are famous for their high performance in various applications, but they are built upon the good quality of training data and suitable network architectures. These challenges also apply to our tasks. First of all, caution must be taken to prepare our data by transforming the original particle trajectories, i.e. spatiotemporal coordinates, to the sequences fed to the model. It is desirable that more original information is directly fed into the model. But it requires a larger capacity of the model, which could make the training time unendurable or the network unallocable. In our approach, we derive the sequences of movements between blocks instead of original blocks for usage, which greatly reduces the size of the model.

Furthermore, the network architecture needs careful design as well as its hyper-parameters. The network is built around the LSTM layer to accept transformed sequences and to generate probabilities of the next movements. We conduct experiments to explore the design space of model hyper-parameters and give an adequately excellent configuration. Our experiments with various flow visu-

*Xiaoru Yuan is the corresponding author. e-mail: {fan.hong, jiang.zhang, xiaoru.yuan}@pku.edu.cn.

alization applications have shown that the data access patterns can be learned with high accuracy. The learned patterns then drive effective prefetching and improve time efficiency in parallel particle tracing.

In summary, the novelty and contributions of our paper include:

- An LSTM-based model to learn access patterns in particle tracing, which obtains comparable accuracy to previous approaches but using fewer training samples and smaller storage overhead;
- A time-efficient particle tracing framework with prefetching driven by our model, which is also a novel form of CPU/GPU collaboration for particle tracing tasks.

In the remainder of this paper, we review the background of this work in Section 2. In Section 3, we introduce our LSTM-based model in detail and evaluate its hyper-parameters. Section 4 describes the parallel particle tracing framework with the prediction and prefetching functions driven by our model. Results are shown in Section 5 to demonstrate the effectiveness and efficiency of our approach. At last, we give some discussions on our approach and present our conclusion.

2 Related Work

We first review literature related to particle tracing and data access patterns in flow fields. Then we introduce the background concerning deep learning and LSTM.

2.1 Particle Tracing and Data Access Patterns

Particle tracing is a fundamental technique to retrieve integral curves in flow fields visualization and analysis. As the data increases to a large scale, particle tracing becomes an inevitable bottleneck [18]. Parallel computation is an efficient solution. Three parallel strategies are usually adopted, i.e. task-parallelism, data-parallelism, and hybrid-parallelism. Task-parallelism mainly focuses on the workload balance across different tracers [22, 24, 26], while data-parallelism focuses on the data partition [6, 23]. More recently, hybrid-parallelism becomes popular [2, 21]. DStep [19] is a MapReduce-like particle tracing framework, which has become one of the most scalable methods. Some flow analysis methods are built on DStep [12, 13] to improve their scalability and efficiency. There are a few works combining CPUs and GPUs for tracing computation, whose idea is to involve GPUs for the integration computation [3, 25]. Camp et al. [3] conducted a study to access the benefits and limitations brought by GPUs.

Besides parallel computation, access patterns estimation is an effective way to improve the performance of particle tracing. It can work with or without parallel computation. In particle tracing, block access dependencies are usually considered as the access patterns, i.e. the behavior of particles moving from blocks to other blocks. Access dependency graph is widely used to optimize data storage [5], to provide sparse data management [14], to prefetch data [29], or to dynamically balance workload [24]. Their purpose is to improve data locality in particle tracing, so that I/O cost is reduced.

In this work, we adopt a neural network model to learn data access patterns in flow fields. Our results show that our approach has achieved an excellent estimation for access dependencies. We further employ our model to drive prediction and prefetching functions for out-of-core particle tracing to reduce I/O costs. Our approach also embodies a new form of CPU/GPU collaboration, where the learning model is deployed on GPUs for access pattern predictions.

2.2 Deep Learning and LSTM

Deep learning is part of machine learning methods aimed at learning data representations with deep neural networks. Deep learning architectures have been applied to various fields, and have achieved results which are comparable or even superior to human experts. In this work, we are interested in a special type of deep learning architectures, named recurrent neural networks (RNNs) [20]. Compared to classical feed-forward neural networks, RNNs allow connections between neuron units to form a directed cycle, which enables them to represent dynamic temporal behaviors. Long short-term memory (LSTM) is one type of RNNs, which has achieved state-of-the-art performance in speech recognition [30], machine translation [7], text generation [27], etc. The key part in LSTM is the memory unit, which is capable of keeping values for arbitrary time periods without iterative modification [10, 17]. This property gives LSTM models the ability of remembering, which makes it very suitable for learning from temporal sequences. There exist some variations of LSTM, like peephole LSTM [9] and Gated Recurrent Unit (GRU) [7]. However, none of the variants can improve upon the standard LSTM architecture significantly [11]. There is also some sophisticated usage of LSTM being proposed, such as seq2seq [28] and Encoder-Decoder [7]. We do not use these models mainly for two reasons: our target problem does not involve the "translation" between two different sequences, instead, only prediction tasks in one sequence; and the intermediate representation could cause information loss compared to the direct usage.

In this work, we make a novel adoption of LSTM for the estimation of access patterns in particle tracing. Exploration on the mapping from particle trajectories to sequences data and the model architectures are made, without prior experience in this problem. We believe it is the first work which employs deep learning techniques to tackle the challenges in particle tracing.

3 LSTM-based Learning Model for Data Access Dependencies

In the following, we present the technical details of our LSTM-based model for access patterns estimation. We first present how we formulate the access pattern estimation problem into a classification problem that could be effectively handled by machine learning models, and the according data transformation. We discuss the rationality behind our choices at the same time. Then, we introduce the basics of Long Short-Term Memory (LSTM) and technique details of our neural network model built upon the LSTM layer. We provide a description of the model architecture, and how we train and test it. At last, we conduct experiments to evaluate hyper-parameters related to our model.

3.1 Mapping from Particle Trajectories to Sequences

The particle trajectories are naturally sequences of real-valued coordinates, but they are infeasible to be directly fed to a machine learning model. There are a few choices must be made when preparing our data for the model. In the following, we present how the particle trajectories are mapped to sequences of movements and the rationality of our choices. Meanwhile, the access patterns estimation problem is formalized as a classification problem.

3.1.1 Continuous or discrete data type?

The first choice we have to make is the type of data fed to the model. For the data access patterns problem in flow fields, we usually consider the access dependencies between blocks. Previous approaches also manipulate data from the level of blocks, either for file layout reorganization, data sparse management, or for data prefetching. Therefore, we expect our learning model to process blocks directly. Under such precondition, the access pattern estimation problem becomes a classification problem: for one particle, given the history of its passed blocks, we expect our model to predict which block the particle will enter next. Although working with real-valued coordinates may also be used to predict blocks, it demands higher precision for the model. For example, if the predicted coordinates have small errors near the borders between blocks, the prediction of blocks could be totally wrong. Therefore, we decide to transform the coordinates sequences into block sequences, in which blocks are identified using block indices. In a 3D unsteady flow field, the block indices are tuples of 4 integers, where 3 for spatial indices and 1 for timestep.

3.1.2 Sequences of blocks or movements?

The block sequences still have difficulty being directly processed by a learning model, because the number of unique blocks in an unsteady flow field data is usually too large. For example, in the Hurricane Isabel dataset with a domain size of $500 \times 500 \times 100$ and 48 timesteps, there are $1,200,000$ blocks if we set the block size to $10 \times 10 \times 10$. It means a classification problem of $1,200,000$ classes, which is rarely seen in classical machine learning problems. Such a huge number will make the predictions very difficult and turn the number of parameters in the model to be very large. A large number of parameters will make it formidable to train such a model, due to either unendurable training time or large GPU memory occupation.

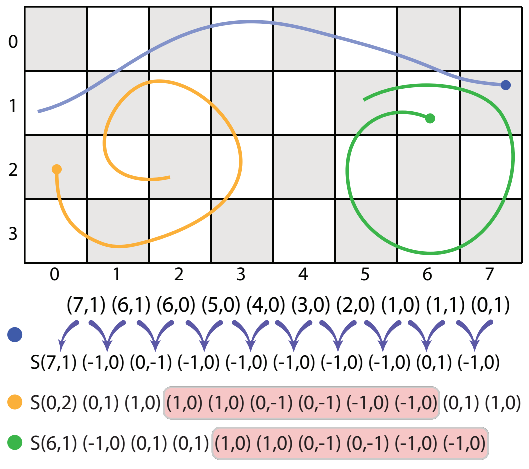
0
1
2
3
0 1 2 3 4 5 6 7
(7,1) (6,1) (6,0) (5,0) (4,0) (3,0) (2,0) (1,0) (1,1) (0,1)
● S(7,1) (-1,0) (0,-1) (-1,0) (-1,0) (-1,0) (-1,0) (-1,0) (0,1) (-1,0)
● S(0,2) (0,1) (1,0) (1,0) (1,0) (0,-1) (0,-1) (-1,0) (-1,0) (0,1) (1,0)
● S(6,1) (-1,0) (0,1) (0,1) (1,0) (1,0) (0,-1) (0,-1) (-1,0) (-1,0)

Figure 1: An illustration of the mapping from particle trajectories to sequences. The passed block sequence of the blue particle is transformed to a sequence of movements prepended by its seeding block. The passed blocks of the yellow and green particles are totally different, but their movement sequences share a common subsequence highlighted in red boxes.

In our approach, we transform the block indices sequences into movement directions sequences. We consider how particles move from one block to its neighboring blocks. Figure 1 gives an illustration of our approach using a 2D flow field without the time dimension. One particle moves from one block to one of its neighboring 8 blocks, if the current block is not on the boundary of the domain. The differences of the block indices are used to recognize this movement. There are only 8 different cases in normal situations. In a 3D domain, 26 different movements happen. If time dimension involved, the number is doubled plus one, since the particle either stays in the same timestep or proceeds to the next one. There are rare cases where a particle moves so fast to pass through neighboring blocks and reaches other blocks, due to a large integration step and a small block size. The number of different movements could increase to hundreds. Yet, it is still a significant reduction compared to original block indices-based representation.

One additional benefit of the movement representation is that indirect access dependencies of blocks could be learned. For example, in Figure 1, yellow and green particles start from different blocks, but show similar swirling movements. In their movement representations, they share a common subsequence (highlighted in red boxes) representing such swirling patterns. Existing works have shown that LSTM-based models can learn these patterns effectively, since they are explicitly exposed in the sequences. In the prediction phase, if an unseen particle shows similar swirling behaviors, although not exactly in the same blocks, the model can still predict its movement well. In contrast, in previous approaches, the access patterns are strictly bound to block indices. It is impossible for them to transfer the access dependencies learned from particles in one block to those in another block.

3.1.3 All information preserved?

Although the movement representation described above can greatly reduce the storage or time cost of the model, not all information of particles is preserved, if compared with the block indices-based representation. From only the movement sequence of one particle, it is not possible to restore the whole block indices sequence of its history. To overcome this limitation, we prepend the indices of seeding blocks of particles to their movement representations. In the Figure 1, the seeding blocks are represented using "S" plus the block indices. The passed blocks of one particle can be fully restored by accumulating the movements on its seeding block sequentially. This makes sure that no information is lost after our mapping compared to the original block sequences.

Under the above choices we have made, particle trajectories are mapped to movement representations for the model training and inference phases.

3.2 Basics of Long Short-Term Memory (LSTM)

LSTM is one kind of recurrent neural networks (RNNs), which is very unlike feed-forward neural networks. In feed-forward neural networks, information flows only in one direction, from the input neurons, passes through hidden neurons if any, to the output neurons. There is no direct relations between the outputs from different inputs. In contrast, RNNs are able to maintain certain information by forming cycles between neurons. However, general RNNs are not suitable for handling long-term dependencies because of the gradient vanish problem [1]. LSTM is one type of RNN architectures, in which a special memory unit is used to maintain information extracted from previous inputs. The stored values are not modified as the learning proceeds, so LSTM can learn long-term dependencies from the sequences. The maintained information or the stored values are denoted as hidden states in our following description. As hidden states involved, when the items of a sequence are fed to an LSTM network sequentially, the outputs are actually determined by not only the current item but also all items prior to it.

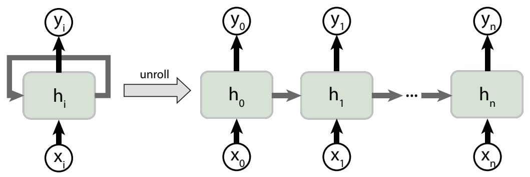

Figure 2: Illustration of an LSTM network, and its unrolled view.

Figure 2 gives the illustration of an LSTM network and its unrolled view. The unrolled view shows how an LSTM processes a sequence. Inside an LSTM, there are complex structures, such as input, output, forget gates, and their computations, to enable the functions of remembering and forgetting. These structures are simplified as the hidden states h_i of LSTM units in the illustration, which we will not give mathematic details. We introduce the behaviors of LSTM from the high level.

We denote the input sequence as $\mathbf{x} = \{x_0, x_1, \cdots, x_n\}$. At the beginning, the hidden states of the LSTM are randomly initialized or just set to $\mathbf{0}$. When it accepts input item x_0, its hidden states change to h_0. Then, as the input item x_1 comes, the hidden states update to h_1. This procedure continues until all items in the sequence are processed. We obtain a sequence of hidden states $\mathbf{h} = \{h_0, h_1, \cdots, h_n\}$. The update mechanism, i.e. the parameters of LSTM, is learned through the training process. From the hidden state sequence $\mathbf{h}$, the output sequence is derived using certain functions, which we denote as $\mathbf{y} = \{y_0, y_1, \cdots, y_n\}$. Therefore, each y_i is decided together by the inputs $x_0, x_1, \cdots, x_i$, and the initial hidden states h_0. Note that $x_{(.)}, y_{(.)}$ and $h_{(.)}$ are high-dimensional vectors with the same size.

The output sequence $\mathbf{y}$ can further be processed by other neural layers for specific tasks. For example, if we want to adopt y_i for classification tasks, we can add several layers to transform y_i to probability distributions over the set of classes.

3.3 Model Architecture

With the data preparation discussed in the previous subsection, the particle trajectories are formulated as sequences of movements plus seeding block indices. All possible movements and seeding blocks are denoted as D_m and D_s respectively. They make up a whole set of possible elements in the sequences, which is called D. Remark that, in our proposed learning model, the exact values of movements or seeding blocks do not contribute to the computation. They are treated as categorical values, so the model only needs to determine whether two elements are identical or not.

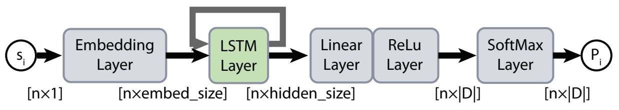

Figure 3: LSTM-based network architecture used in our approach. Sizes of input and output data are shown between layers.

Figure 3 gives an illustration of the network architecture of our LSTM-based model. The input is a movement sequence $\mathbf{s} = \{s_0, s_1, \cdots, s_{n-1}\}$ with length n. At the beginning, every element in the sequence is transformed into a real-valued vector through an embedding layer. This is a necessary step because most neural layers handle numerical values only rather than categorical values. In the embedding layer, all elements in the dictionary D are projected into another space of *embed_size* dimensions. However, the projection conducted here is different from those in other techniques like principal component analysis (PCA) or multidimensional scaling (MDS). In the embedding layer, the learned projection parameters form a direct mapping from categorical values to real-valued vectors, while PCA or MDS learn a transformation for original high-dimensional data. Moreover, the embedding parameters are learned together with other parameters in the model, rather than by analyzing statistical features of data.

After the embedding layer, the sequences of high-dimensional vectors are fed into an LSTM layer. The number of dimensions of the hidden states in the LSTM layer is denoted as *hidden_size*, which is equal to *embed_size*. As we have described in the previous subsection, the LSTM layer outputs a sequence of vectors of *hidden_size* dimensions corresponding to each item in the input sequence.

To transform the output of the LSTM layer for the classification task on D, several layers are appended. A full-connected linear layer transforms every item in the output sequence to a vector of length $|D|$. Then a ReLU activation function, i.e. $f(x) = \max(0, x)$, is further used to rectify those negative values to 0. Till now, the vectors convey non-negative values, which can be treated as "weights" for each element in D. At last, a softmax layer follows to rescale these weights, so that they all lie in the range $(0, 1)$ and sum to 1. Softmax is defined as $f_i(x) = e^{x_i} / \sum_j e^{x_j}$, where x_i is the component in the original vector, and $f_i(x)$ is its transformed value. Therefore, for each input item, its corresponding output vector is the probability distribution P_i over all elements in D.

The size of the model can be further reduced to decrease the training time and storage size. Since the next movements of particles are chosen from D_m only, we can calculate the probabilities only for them. The output size of the linear layer and following ReLU layer and SoftMax layer can reduce to $n \times |D_m|$. But for the embedding layer, we still need to process $|D|$ elements.

In summary, our model accepts a sequence of the movement representation $\mathbf{s} = \{s_0, s_1, \cdots, s_{n-1}\}$, and outputs a sequence of probability distributions $\mathbf{P} = \{P_0, P_1, \cdots, P_{n-1}\}$. Due to the property of the LSTM layer, each P_i is determined not only by s_i, but also by $s_0, s_1, \cdots, s_{i-1}$. Therefore, under our mapping, the P_i stands for the probability distribution of next movement conditioned by all previous movements and the seeding block. These probability distributions are further used in our parallel particle tracing framework for data block prefetching.

3.4 Model Training and Testing

To train the model, a few particles are sampled from the original flow field data. Their trajectories are transformed to sequences of movements and seeding blocks. Since we only need the blocks/movements sequences rather than the original coordinates for the training, these trajectories do not need to be very accurate. They are generated with a relatively larger integration step and less accurate integration method like 1st-order Runge-Kutta, so the computation of the training samples is fast. We denote the sequence of one particle as $\mathbf{s} = \{s_0, s_1, \cdots, s_n\}$, where $s_0 \in D_s$ is the seeding block index, while other $s_i \in D_m$ represent the movements. The model outputs vectors of probabilities of next movement, i.e. $\{P_0, P_1, \cdots, P_n\}$, where P_i is the probability distribution over all possible movements D_m. For each movement s_i where $0 < i \leq n$, $P_{(i-1)}(s_i)$ is the probability of right prediction. Usually, negative log-likelihood is used as the loss function, i.e. $-\log(P_{(i-1)}(s_i))$. Then, for sequence $\mathbf{s}$, the total loss is

$$Loss(S) = \sum_{0<i\leq n} -\log(P_{(i-1)}(s_i)).$$

Note that P_n is not involved into the loss function calculation, because no ground truth s_{n+1} provided.

In the training process, the *optimizer* calculates the sum of losses using a specific set of training samples and derives its gradients on model parameters. The *learner* then updates parameters based on gradients with certain strategies. The total loss is expected to be minimized after multiple iterations. Thanks to the highly developed deep learning frameworks, we no longer need to implement the backward propagation algorithm, as well as its optimizer and parameter learner. Instead, we only need to select suitable ones from various options.

For the optimizer, mini-batch gradient descent is chosen, in which only a small subset is chosen for total loss calculation. Compared to stochastic gradient descent, where only one randomly picked example is chosen in each iteration, fewer updates apply to the model to keep computational efficiency. While compared to the full-batch approach, the model's update frequency is higher, allowing a more robust convergence to avoid local minima. So mini-batch gradient descent is a compromise of full-batch and stochastic approaches. In our training process, the batch size is set to 200 to balance convergence and training time. The number of epochs is set to 10, that is the model goes through the whole training set 10 times. The current setting is sufficient to achieve good convergence compared with smaller batch sizes or more epochs.

Together with mini-batch gradient descent, RMSProp (Root Mean Square Propagation) is chosen as the learner for model parameters because of its recognized good performance. RMSProp can automatically adjust the learning rate so that it keeps on the same scale of the gradients. It has shown excellent adaptation of learning rate in different applications.

In the testing process, we calculate the prediction accuracy through a validation dataset. Negative log-likelihood is not used as the accuracy measure, although it is suitable for model training. In the application scenario of data prefetching, it is possible that multiple blocks are prefetched once a time based on the prediction results. The probability of the hit is thus increased as unexpected data fetching avoided. Therefore, from the probability distribution P_i, one or multiple movements with top probabilities are extracted, which we denote as the set $pred_i$. We then test if the next movement is included in $pred_i$, i.e $s_i \in pred_{i-1}$. For each sequence $\mathbf{s}$, the hit ratio is defined as

$$hit\_ratio = \frac{\sum_{0<i\leq n} I[s_i \in pred_{i-1}]}{n}, \quad (1)$$

where $I[\cdot]$ is the indicator. For the whole test dataset, we take the average of hit ratio as the measurement of the model performance.

3.5 Parameter Evaluation

There are lots of factors affecting the model performance, including the hyper-parameters of the model, the quality and size of training dataset, block size, and so on. Fully exploration of all possibilities to obtain an optimal one is usually unrealistic. In classical deep learning tasks, like image classification or object identification, researchers have accumulated abundant experience for hyper-parameter configurations. For us, learning access dependencies in flow fields is a totally new scenario, where no previous works exist. It is unknown whether existing experience or rules of LSTM-based models or general deep learning models still apply to our scenario. We extensively studied the influences of hyper-parameters, sizes of training samples, and block sizes under our current network architecture, and report our results in the following. The accuracy is measured using *hit_ratio* in Equation 1. We use the Hurricane Isabel dataset as an example to demonstrate our evaluation process. Isabel data has a domain of size $500 \times 500 \times 100$ and 48 timesteps. We only consider particle trajectories seeding at the first timestep.

3.5.1 Number of training samples

Usually, a larger training dataset could give better performance, but may also cause overfitting. As in our application scenario, since our overall goal is to accelerate particle tracing tasks, using a large number of samples is undesirable in the preprocessing stage. We intend to find a balance between these two aspects. We sample two sets of particles trajectories from flow fields. One set contains particle trajectories with uniformly located seeds. For the Hurricane Isabel dataset, we partition it into blocks of size $10 \times 10 \times 10$, and use the trajectories seeding at the center of these blocks. There are $25,000$ trajectories in total. It makes sure that the training samples cover the domain roughly to avoid uncovered regions. The other set of trajectories are randomly sampled in the whole domain without duplicates to increase the number of training samples. In our experiments, the model performance is tested under different numbers of samples.

The results are shown in Figure 4(a). We observe from the results that with more training data, the hit ratio is not guaranteed to improve. Although with $1,025,000$ samples the model obtains the best hit ratio, the accuracies of other options are very close. Only for $525,000$ samples, the model performs slightly worse. One reason could be the similarities between training samples. Classical machine learning applications usually require the training samples to be as diverse as possible to improve the model performance. While in our scenarios, the diversity of particle trajectories are limited due to the spatial coherence. The results indicate that modification of the network architecture is required for further extracting

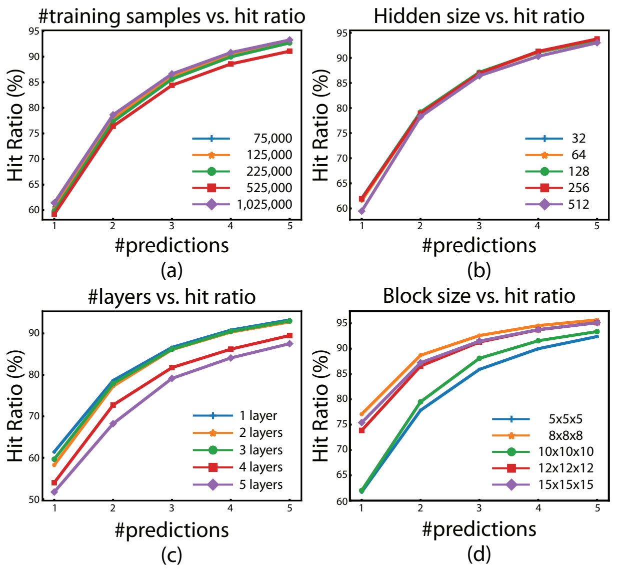

Figure 4: Hit ratios under different numbers of predictions using the model of different (a) numbers of training samples, (b) hidden sizes of the LSTM layer, (c) numbers of stacked LSTM layers, and (d) block sizes.

useful information from large training sets. Considering both the experiment results and the generalization ability, though smaller training sets can also give good results, we decide to use $1,025,000$ sampled trajectories for future experiments. It is only 32% of the training samples used in high-order work [29], but can achieve comparable accuracy as shown in Section 5.

3.5.2 Width and depth of LSTM layer

In neural network architectures, especially convolutional neural networks (CNNs) in image-related tasks, wider and deeper networks usually brings better performance, though sophisticated designs are required. We would like to study whether the width and depth affect our network performance. In the LSTM layer, the width denotes the size of hidden states, i.e. *hidden_size*, of the LSTM layer. A larger *hidden_size* means higher-dimensional vectors are used to store the information extracted from the data. At the same time, multiple LSTM layers can also be stacked to increase its depth. The output sequences of one LSTM layer are not directly used for tasks. Instead, they are fed to another LSTM layer as inputs. It is possible that additional layers learn higher levels of abstraction of the data. But as for our tasks, it is still unknown what a wider and deeper LSTM layer could bring to the performance.

Figure 4(b) and (c) show results of the hit ratio of different sizes of hidden states and different numbers of stacked LSTM layers. As for the hidden sizes, we observe there are no significant differences between the options. A hidden size of 256 is slightly better, while smaller hidden sizes give almost the same performance. When the hidden size increases to 512, the hit ratio starts to drop visibly. From the accuracy of training and validation sets, we think the reason could be the overfitting, which is a common issue of deep learning architectures with wide layers. Since the difference is not significant in our testing, we just choose the option 256. For the numbers of layers, we can observe the hit ratio drops when more layers stacked. Especially, when stacking 4 or 5 layers, there is almost 10% drop compared to using 1 layer. Overfitting could still be the reason, as the training samples may not provide more information for the network to extract due to their limited diversity. Even in classical applications, like image recognition, some works [16] have reported that deeper networks do not always bring better performance, instead, degradation could be introduced. Based on the above results, we decide to choose 1 LSTM layer. The shorter training time is also its advantage.

3.5.3 Block size

Though the block size is not a direct hyper-parameter of the deep learning model, it is an important factor in our overall approach. Previous works [14, 29] have shown that block sizes influence the final tracing performance greatly. The influence of the block size to our model is even more complicated, since it affects not only the precision of the model itself but also its input data and the usage of its outputs. Modeling the trajectories with a small block size preserves more precise information, so the model could be more accurate. But a small block size requires the model to make more accurate predictions, even with movement representations as particles could move beyond neighboring blocks. Besides, the input sequences are also changed with different block sizes. Therefore, it is necessary to investigate the influence of different block sizes

Figure 4(d) shows accuracy testing results under different sizes of blocks. Different options produce larger differences than previous hyper-parameters. We find the block size of $8 \times 8 \times 8$ gives better accuracy. A large block size of $12 \times 12 \times 12$ and $15 \times 15 \times 15$ give very close results to the best option. But $10 \times 10 \times 10$ gives very bad accuracy, especially when the number of predictions is less than 3. Overall, the results show no obvious correlation between the block sizes and accuracies, because their relations are complicated as we have discussed in the last paragraph. For now, we use a block size of $8 \times 8 \times 8$ for the following usage.

From above experiments, we found that the block size influences the performance most, the number of stacked LSTM layers takes the second place, and other factors least. But for now, the network architecture is fixed for simplicity. A complete design study involving variations of model architectures is desirable to obtain a better understanding of deep learning models in our tasks. Right now, our current configuration could reach a hit ratio of $\sim 76\%$ with 1 prediction for the Hurricane Isabel dataset. It matches the accuracy achieved by the high-order work [29], but with fewer training samples and smaller storage. When the number of predictions increases to 3, more than 90% of the particles' movements are in the prediction, which could avoid lots of unexpected data fetching in particle tracing tasks.

4 Deep Learning Model-Driven Prefetching for Parallel Particle Tracing

Based on the proposed deep learning model, we further develop a prefetching component and employ it for parallel particle tracing computation. The parallel computation part is based on Zhang et al.'s work [29] and Guo et al.'s work [14]. In Guo et al.'s work, a two-layer cache scheme is proposed for task-parallel out-of-core particle tracing with limited computation resources. The cost of I/O requests is reduced greatly by the cache mechanism and the prefetching based on 1st-order block access dependencies. In the following Zhang et al.'s work, high-order dependencies, i.e. more than one historical visited blocks, are used to predict and prefetch future possible blocks. The data usage of prefetched blocks is improved, so as the time efficiency.

Compared to the high-order work [29], our framework has two major differences. The prediction part is now driven by our LSTM-based model, which could involve the whole history of particles for prediction. In addition, the learned patterns are represented and stored with smaller storage overhead, so they can reside in (GPU) memory in long-term. While in the high-order work, tracers have to request disks frequently to retrieve high-order dependencies bound to blocks. In addition, our framework combines CPUs and GPUs for particle tracing, because the deep learning model is deployed on GPUs to make predictions for particles. It is a new and potential form of CPU/GPU collaboration for particle tracing tasks, where only a small amount of data transferred between the main memory and GPU memory.

4.1 Workflow

The overall workflow of our approach is similar with Zhang et al's work, so our introduction to the tracing framework is concise. The workflow of our model-driven particle tracing framework is illustrated in Figure 5. The raw data is partitioned into blocks indexed by spatiotemporal locations. In the preprocessing stage, a small set of pathline samples are generated to train our LSTM-based model. The configuration of the network and training process is chosen based on the experiments described in the previous section. In the application stage, when advecting particles, our model can predict next blocks of particles based on their histories of passed blocks. Tracers then prefetch blocks based on predictions. Since the I/O requests and latency are reduced and hidden, the time efficiency of parallel particle tracing is improved.

4.2 Runtime Particle Tracing with Prefetching

In the framework, particles are traced in a task-parallel way, i.e. the particles are assigned to tracers evenly and are advected. In the resource-limited scenario, data of blocks are loaded from disks and cached into main memory when needed or prefetched, and purged with a least-recently-used (LRU) policy. When one particle enters a block whose data is not in the cache, a prefetch request is issued. The historically passed blocks of this particle are transformed to sequences and fed into our model as described previously. Our model then makes predictions of the next movement, and gives several candidate blocks. These blocks are prefetched from disks together at a time. Unlike high-order approaches, whose dependencies are too large to be loaded into main memory all at once, our approach needs no additional disk read operations for dependencies.

4.3 Implementation and Deployment of Our Model

The original particle tracing framework is implemented using C/C++ language. While our LSTM-based model is implemented in Python language using PyTorch[1]. To utilize our model for prediction and prefetching in our framework, we glue them using the C/C++ embedding feature of Python.

Our LSTM-based model is deployed on GPUs, where PyTorch handles the data transfer between CPUs and GPUs. In our usage, the transferred data only includes movements sequences (CPUs to GPUs) and predicted movements (GPUs to CPUs), whose sizes are very small. We can further save the transferred data as well as the prediction time thanks to the property of LSTM. After our model makes predictions for one particle, the hidden states of the model are saved. When the particle needs prefetching the second time, it surely passes more blocks. Thus, the model only needs to process those new movements starting from the saved hidden states. Both transferred data and prediction time is greatly reduced. Note that these saved states still reside in GPU memory, so there is no data transfer for them. To avoid the GPU memory to be filled by the saved states, we purge those recently unused states every few seconds.

5 Results

In this section, we present the evaluation results of our approach on various flow visualization applications. Three datasets are used in the evaluation (Figure 6): Hurricane Isabel Dataset, GEOS-5 Simulation Dataset, and Ocean Simulation Dataset. The Isabel data is a

[1]http://pytorch.org/

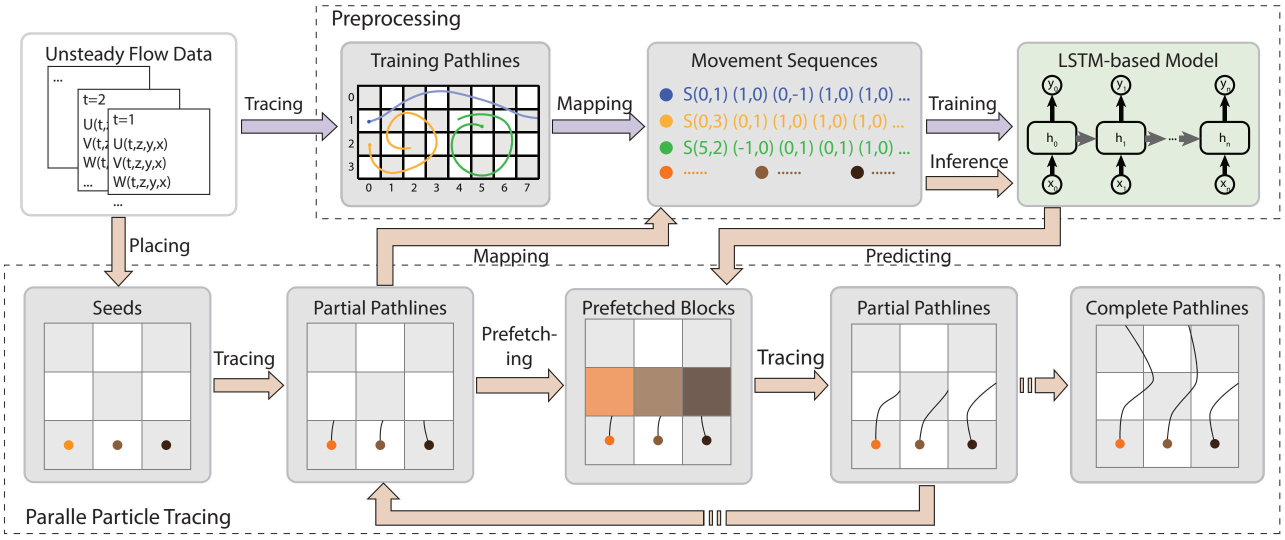

Figure 5: Workflow of our parallel particle tracing framework with deep learning model-driven prediction and prefetching. The processing steps in preprocessing and tracing stages are distinguished by colors of arrows.

hurricane simulation from the National Center for Atmospheric Research in the United States. Its spatial resolution is $500 \times 500 \times 100$. There are 48 timesteps, which are hourly saved in separate files. The GEOS-5 simulation is an atmospheric model from the NASA Goddard Space Flight Center. The spatial resolution is $1^\circ \times 1.25^\circ$ with 72 vertical pressure levels contained. The data consists of 24 monthly averaged simulation results from January 2000 to December 2001. The Ocean data comes from global ocean simulation, which has a very high spatial resolution of 1801×842 and 55 depth levels. The temporal resolution is 1 day. We use 28-day of data for our testing, whose size is about 27GB.

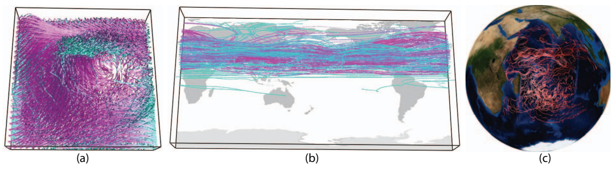

Figure 6: Three datasets used in our performance testing: (a) Hurricane Isbael, (b) GEOS-5 Simulation, and (c) Ocean Simulation.

We test the runtime performance on a symmetric multiprocessing node, with two Intel Xeon CPU E5-2650 v4 CPUs operating at 2.4 GHz supporting at most 24 processes or 48 threads with hyper-threading. The node is equipped with 8 NVIDIA M6000 graphics cards, with 3072 CUDA cores and 12GB memory each. The data is stored in a local disk with 2TB capacity.

In our performance testing, we compare our approach with two baseline methods, particle tracing without prefetching, and prefetching with high-order dependencies [29], which we call it high-order work for short in the following. For the high-order work, 4th-order dependencies are used for comparison, and we follow the parameters reported in the paper. We choose 4th-order dependencies only because they show closer runtime performance to our approach. But the required storage of 4th-order dependencies is much larger ($50\times \sim 230\times$) than our approach. 6th-order dependencies can be even larger than the original data. In case the flow data is very large, high-order approach could be prohibited due to its storage cost. If we require the high-order work to use the same storage as ours, its performance will be much worse than ours.

Our framework is designed for parallel computing environment where each process obtains limited computing resources, i.e. main memory and GPU memory. Unfortunately, we do not possess a cluster of computation nodes, where each node is equipped with a high-end graphics card, for our framework to deploy on. Therefore, a symmetric multiprocessing node is employed, which could reduce the communication and data read costs considerably. To make the testing fair for two baseline methods, they are deployed on the same platform for testing.

5.1 Preprocessing

In preprocessing, the hyper-parameter selection is conducted as described in Section 3.6 for each dataset. The results is reported in Table 1. In the test, more training samples are used for Isabel dataset and Ocean dataset, because of their high spatial resolution. However, the average sequence length, which represents the average blocks particles visit, in GEOS-5 dataset is the almost double of that in Isabel dataset, because these two datasets have very different time extents and time scale. Ocean dataset has smaller average sequence length, due to the large block size which comes from its large spatial resolution. Above two factors make the differences of the storage and training time of the model. Comparing with the high-order work, our method requires much fewer training samples and costs much smaller storage overhead. For example, for Isabel dataset, our trained model only occupies 53MB, while 4th-order dependencies need 5.7GB, and 6th-order occupies 20GB, which is even larger than the raw data. At the same time, our model only uses about one third of the training samples in the high-order work to obtain comparable accuracy.

5.2 Runtime Performance

In our experiments, we test both local- and full-range analyses on three datasets using our model-driven parallel particle tracing framework as shown in Figure 6. We measure the performance from two aspects: running time and data usage. The data usage is defined as the percentage of prefetched data really used. Unlike the hit ratio defined previously, the data usage drops when using more predictions, because only one block is expected to be really used by the corresponding particle. However, the prefetched blocks

Table 1: Preprocessing results of Isabel, GEOS-5 and Ocean datasets.

Dataset	Dimensions	Data Size	Block Size	#Training Samples	Ave. Seq. Length	Storage			Training Time
						Ours	4th-order [29]	6th-order [29]	
Isabel	$500 \times 500 \times 100 \times 48$	13.4GB	$8 \times 8 \times 8$	1,025,000	51.1	53MB	5.7GB	20GB	6030.0s
GEOS-5	$288 \times 181 \times 72 \times 24$	1.34GB	$8 \times 8 \times 8$	458,278	92.8	9.6MB	388MB	1.2GB	8148.9s
Ocean	$1801 \times 842 \times 55 \times 28$	27GB	$20 \times 20 \times 10$	1,134,017	25.6	16MB	3.6GB	12GB	5999.8s

can be shared by other particles, which increases the data usage. In the testing, we evaluate how the number of predictions influences the running time and data usage. At the same time, we test how the performance changes with increasing number of processes and particles to investigate its scalability. In the evaluation, the cache size of every process is set to 1GB, and the GPU memory for saved hidden states is limited to 512MB per process.

5.2.1 Local-range Analysis

In the local-range analysis, seeds are densely placed in a local region for visualization and analysis of flow fields. Source-destination queries [19] is one typical local-range analysis. Since the seeds are spatially neighbors, their visited blocks are expected to be similar. In our experiments, we select a region and trace 500,000, 87,500, and 800,000 pathlines for three datasets respectively.

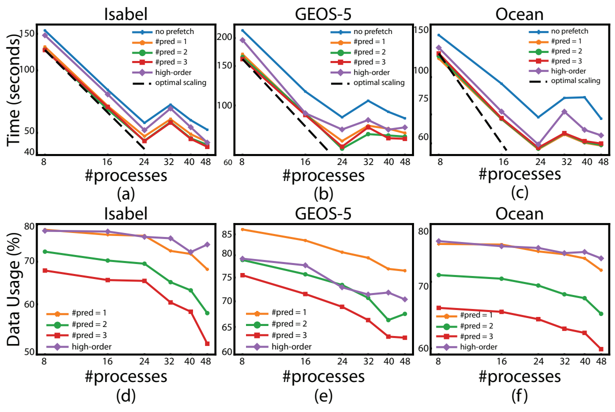

Figure 7: Local-range analysis results of running time (a-c) and data usage (d-f) under different #processes.

Figure 7 shows the results of running time and data usage under different #processes. From Figure 7(a-c), we can clearly see that our approach shows significant performance improvement compared to tracing without prefetching. Compared with the high-order work, our approach shows slightly better time-efficiency. From Figure 7(d-f), the data usage of high-order approach is close to that of our approach with 1 or 2 predictions in different datasets. Generally, higher data usage usually gives shorter running time, but the prefetching mechanisms also influence the performance. High-order work could be slowed down by more disk read operations. We notice that there is a sharp turning of running time using 24 and 32 processes in our method and two baseline methods. This is caused by the configuration of our computation node, as the hyperthreading technique is used to support more than 24 processes. When the number of threads is not large enough, the benefits brought by hyperthreading do not overcome its costs. However, we can observe, except these points, our approach actually shows good scalability.

At the same time, we observe, with more predictions and prefetched blocks once a time, the improvement of our approach is not significant. The reason comes from the property of local-range analysis, where the consistency of access patterns between particles is already high. The data fetching costs cancel the benefits brought by multiple predictions. From current results, 2 and 3 predictions almost coincide with each other in terms of running time.

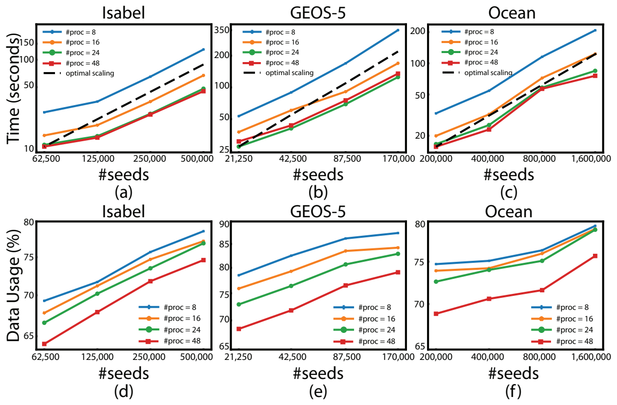

Figure 8: Local-range analysis results of running time (a-c) and data usage (d-f) under different #seeds.

In Figure 8, we further show the running time and data usage of our approach under different #seeds. Because of the hyperthreading issue, we skip the results with between 24 and 48 processes. From Figure 8(a-c), linear or superlinear scaling is observed for all datasets, even when the number of seeds is large. With more seeds, prefetched blocks are reused by other seeds in a higher probability, which leads to better acceleration. The data usage plots in Figure 8(d-f) give support to this explanation. As the number of seeds raising, the data usage increases stably.

5.2.2 Full-range Analysis

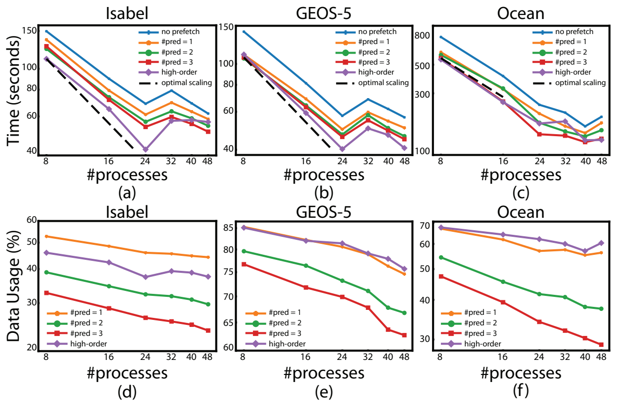

Figure 9: Full-range analysis results of running time (a-c) and data usage (d-f) under different #processes.

In the full-range analysis, seeds are uniformly placed in the whole domain with strides to give an overview distribution of the flow fields. Since the spatial locations of seeds are widely distributed, their visited blocks are more different from each other than in the local-range analysis. Reuse of prefetched blocks is thus weakened under limited memory. In our experiments, seeds are placed with strides of $20 \times 20 \times 10$, $4 \times 4 \times 4$, and $10 \times 10 \times 10$ in three datasets respectively.

Figure 9 shows the running time for all datasets and corresponding data usage. From Figure 9(a-c), it it clear to see that our fetching function could improve the time efficiency greatly compared to tracing without prefetching. Moreover, unlike local-range analysis, with more predictions and prefetched blocks, the performance continues increasing. Although, from Figure 9(d-f), the data usage starts at lower values and drop faster compared to those in local-range analyses, the prefetched data still brings considerable improvement of running time. This is because the full-range analysis is a more difficult task, so prefetching could bring greater benefits. When using more processes, our approach still suffers from the limitations of hyperthreading. However, the scalability is still nearly linear, if not considering those points. As for the high-order work, they give slightly better performance than ours in most cases. But their data usage is lower than ours with 1 prediction in Isabel data, and almost the same in GEOS-5 and Ocean data. The computation cost of model inference could increase the running time.

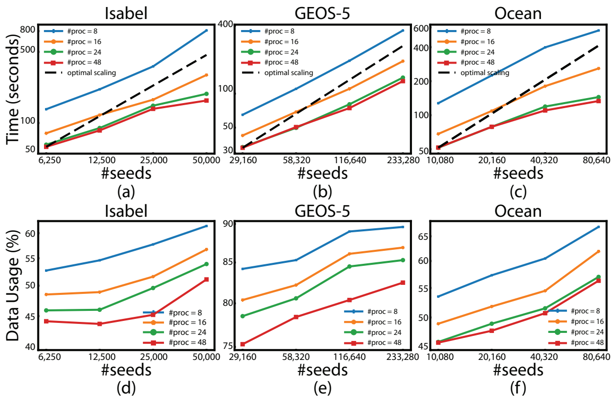

Figure 10: Full-range analysis results of running time (a-c) and data usage (d-f) under different #seeds.

We further test the running time and data usage of our approach under different #seeds. As shown in Figure 10(a-c), in all datasets, our approach shows excellent scaling. For Isabel dataset, the acceleration is even more significant with 24 and 48 processes, and so as the Ocean dataset. From the corresponding data usage plot in Figure 10(d), we find that the increase of data usage helps a lot to the acceleration. The reuse of prefetched blocks increases due to more visiting blocks shared between particles, since seeds are placed more densely. For the GEOS-5 dataset, it also shows superlinear scalability. However, its acceleration becomes attenuated, as well as the data usage shown in Figure 10(e). This is because the data reuse is already high enough due to their data properties.

Comparing the results of local and full range analysis, we find our approach shows slightly better performance in local-range analysis, while high-order outperforms a bit in full-range analysis. Since the configurations of both approaches are fixed for each dataset, this difference should come from the nature of both approaches. The generalization nature of our learning model makes it more suitable for a bundle of neighboring particles, i.e. the situation in local-range analysis, as it tends to learn and predict the common behaviors of particles. While the honest recording nature of high-order approach makes it predict more accurately for more separated particles, i.e. the case in full-range analysis.

In summary, the experiments above have demonstrated that our framework driven by the deep learning model can improve the time efficiency significantly compared to tracing without prefetching. Especially, the excellent scalability under different #seeds brought by our prefetching function is very impressive. In addition, compared with high-order work with 4th-order dependencies, our approach acquires adequate prefetching accuracy and data usage. Because of different prefetching mechanism, our approach achieves better performance in local-range analysis, while high-order work is slightly better in full-range analysis. However, our model can achieve such high accuracy and improvements with fewer training samples and much smaller storage overhead.

6 Discussion

In this section, we mainly compare our LSTM-based model with previous literature concerning data access patterns, and discuss merits and limitations of our approach. We also give a discussion about our approach from the aspects of the deep learning model.

The most significant difference is that our approach learns the data access patterns through an LSTM-based deep neural network. The learning ability means the capability of generalization. Our approach benefits from the generalization from two aspects: fewer training samples and small storage overhead, as shown in previous sections. The generalization also brings some limitations, that is, to avoid overfitting, the access patterns are not recorded honestly. This property makes it achieve better performance when predicting for groups of neighboring particles, such as the scenarios of local-range analysis. Yet, we still believe with finer tuning, our model is able to achieve better performance in the future.

Right now, the generalization of our LSTM-based model is limited to single datasets. Models are trained for different datasets separately. The main reason is the seeding block indices, which implicitly convey unique positional information of each dataset, involved in the movement representation. A more universal representation of initial conditions is desired if we want to train a unified model. The overfitting problem we currently encounter is also expected to be alleviated with more diverse datasets.

Our framework is also novel in terms of its parallel mechanism, i.e. a new way of CPU/GPU collaboration for the particle tracing task. The idea of previous approaches [3, 25] is to move the expensive integration computation from CPU cores to GPU cores. However, massive data transfer is required from main memory to GPU memory for integration. The poor data locality makes the I/O costs even higher, since a same block of data could be transferred to GPUs multiple times. While in our approach, only movement sequences and predictions are transferred to GPUs, which alleviates the data transfer burden greatly. We believe our approach can inspire more time-efficient CPU/GPU collaboration in the future.

As the model itself, it is the first trial which adopts LSTM in particle tracing tasks to the best of our knowledge. Our target problem is different from those classical applications. One significant difference is that the data dependencies estimation is just a sub-task of particle tracing. The prediction accuracy of the model cannot reflect the overall performance of our parallel framework directly. Therefore, we need to prepare our data as the model inputs carefully, and utilize the model outputs wisely, since they are also tightly bound to the final performance. Besides, the diversity of training samples in our application is limited due to spatiotemporal coherence, which is less common in classical application. The experiment results indicate that more sophisticated techniques should be employed to squeeze more information to overcome overfitting.

As a new effort to deploy deep learning in flow visualization, although we have explored the influence of several hyper-parameters, our exploration is still very limited in considering the full space.

We have found the relations between parameters and the prediction accuracy are complicated and behave different from those classical applications. Right now, it is necessary to generate variations of parameters as many as possible and then to choose better options. Besides, the neural network architecture is fixed in our work for now, after considering the training time and required memory. We also tried utilizing other types of layers, like batch normalization layers or drop-out layers, into the network, but did not obtain significant improvements. In the future, we can either systematically investigate architecture variations or adopt more from the state-of-the-art techniques [7, 9, 28]. Nonetheless, the capability and potential of deep learning models has been successfully demonstrated in access pattern estimation and particle tracing tasks.

7 Conclusion and Future Work

In this work, we present an LSTM-based model to estimate access patterns in particle tracing tasks, which is the first work of employing deep learning techniques to tackle flow visualization challenges. A parallel particle tracing framework with prefetching function driven by our model is further developed, where CPUs and GPUs collaborate for the tasks. The effectiveness and potential of our deep learning model has been demonstrated by various flow visualization applications.

In the future, we would like to explore more possibilities of our method in two directions. The hyper-parameters of the model architecture, including the configurations in data mapping and training, are worth a thorough design study. At the same time, we could try more possibilities based on the learned patterns, such as file reorganization, data partitioning, etc., to improve particle tracing tasks and even general flow visualization.

Acknowledgments

This work is supported by NSFC No. 61672055, the National Program on Key Basic Research Project (973 Program) No. 2015CB352503, and the National Key Research and Development Program of China (2016QY02D0304). This work is also supported by PKU-Qihoo Joint Data Visual Analytics Research Center.

References

[1] Y. Bengio, P. Y. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994.

[2] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy. Parallel stream surface computation for large data sets. In *Proc. of IEEE Symposium on Large Data Analysis and Visualization*, pp. 39–47, 2012.

[3] D. Camp, H. Krishnan, D. Pugmire, C. Garth, I. Johnson, E. W. Bethel, K. I. Joy, and H. Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 1–8, 2013.

[4] C.-M. Chen, B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Flow-guided file layout for out-of-core pathline computation. In *Proc. of IEEE Symposium on Large Data Analysis and Visualization*, pp. 109–112, 2012.

[5] C.-M. Chen, L. Xu, T.-Y. Lee, and H.-W. Shen. A flow-guided file layout for out-of-core streamline computation. In *Proc. of IEEE Pacific Visualization Symposium*, pp. 145–152, 2012.

[6] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *Proc. of IEEE Pacific Visualization Symposium*, pp. 87–94, 2008.

[7] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.

[8] M. Edmunds, R. S. Laramee, G. Chen, N. Max, E. Zhang, and C. Ware. Surface-based flow visualization. *Computers & Graphics*, 36(8):974–990, 2012.

[9] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *IJCNN (3)*, pp. 189–194, 2000.

[10] F. A. Gers, J. Schmidhuber, and F. A. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[11] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[12] H. Guo, F. Hong, Q. Shu, J. Zhang, J. Huang, and X. Yuan. Scalable Lagrangian-based attribute space projection for multivariate unsteady flow data. In *Proc. of IEEE Pacific Visualization Symposium*, pp. 33–40, 2014.

[13] H. Guo, X. Yuan, J. Huang, and X. Zhu. Coupled ensemble flow line advection and analysis. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2733–2742, 2013.

[14] H. Guo, J. Zhang, R. Liu, L. Liu, X. Yuan, J. Huang, X. Meng, and J. Pan. Advection-based sparse data management for visualizing unsteady flow. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2555–2564, 2014.

[15] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D: Nonlinear Phenomena*, 149(4):248–277, 2001.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[18] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. B. Ross. Toward a general I/O layer for parallel-visualization applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[19] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2011.

[20] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[21] K. Lu, H.-W. Shen, and T. Peterka. Scalable computation of stream surfaces on large scale vector fields. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1008–1019, 2014.

[22] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed parallel particle advection using work requesting. In *Proc. of IEEE Symposium on Large Data Analysis and Visualization*, pp. 1–6, 2013.

[23] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel particle advection and FTLE computation for time-varying flow fields. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 61:1–61:11, 2012.

[24] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-balanced parallel streamline generation on large scale vector fields. *IEEE Trans. Vis. Comput. Graph.*, 17(12):1785–1794, 2011.

[25] M. Otto, T. Germer, and H. Theisel. Uncertain topology of 3D vector fields. In *Proc. of IEEE Pacific Visualization Symposium*, pp. 67–74, 2011.

[26] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2009.

[27] L. Shang, Z. Lu, and H. Li. Neural responding machine for short-text conversation. In *Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics ACL*, pp. 1577–1586, 2015.

[28] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proc. of Annual Conference on Neural Information Processing Systems (NIPS)*, pp. 3104–3112, 2014.

[29] J. Zhang, H. Guo, and X. Yuan. Efficient unsteady flow visualization with high-order access dependencies. In *Proc. of IEEE Pacific Visualization Symposium*, pp. 80–87, 2016.

[30] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio, and A. C. Courville. Towards end-to-end speech recognition with deep convolutional neural networks. In *Proc. of Annual Conference of the International Speech Communication Association*, pp. 410–414, 2016.