SLOPE: Structural Locality-Aware Programming Model for Composing Array Data Analysis

Bin Dong(✉), Kesheng Wu, Suren Byna, and Houjun Tang

Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA
{dbin,kwu,sbyna,htang4}@lbl.gov

Abstract. MapReduce brought on the Big Data revolution. However, its impact on scientific data analyses has been limited because of fundamental limitations in its data and programming models. Scientific data is typically stored as multidimensional arrays, while MapReduce is based on key-value (KV) pairs. Applying MapReduce to analyze array-based scientific data requires a conversion of arrays to KV pairs. This conversion incurs a large storage overhead and loses structural information embedded in the array. For example, analysis operations, such as convolution, are defined on the neighbors of an array element. Accessing these neighbors is straightforward using array indexes, but requires complex and expensive operations like self-join in the KV data model. In this work, we introduce a novel 'structural locality'-aware programming model (SLOPE) to compose data analysis directly on multidimensional arrays. We also develop a parallel execution engine for SLOPE to transparently partition the data, to cache intermediate results, to support in-place modification, and to recover from failures. Our evaluations with real applications show that SLOPE is over ninety thousand times faster than Apache Spark and is 38% faster than TensorFlow.

Keywords: Multidimensional array · Programming model · Structural locality · Composable data analysis · User-defined function · ArrayUDF · Apache Spark · TensorFlow · MapReduce · Array cache

1 Introduction

The MapReduce (MR) programming model [11] transformed the way of developing data analysis algorithms and led to advanced data analysis systems, such as Spark [42]. These systems enable users to compose complex data analysis algorithms without implementing the details of parallel execution, data management, error recovery, among other challenging tasks. Unfortunately, the MR programming model could not be effectively used for scientific data analysis due to its lack of support for multidimensional array data model and their limited support for accessing neighbors in arrays [13,24]. As arrays are prevalent in scientific data, such as 2D sky survey images [4], we propose a programming model that

M. Weiland et al. (Eds.): ISC High Performance 2019, LNCS 11501, pp. 61–80, 2019.
https://doi.org/10.1007/978-3-030-20656-7_4

operates directly on arrays and an execution engine to run composite analysis on massively parallel computing systems efficiently.

A programming model for data analysis conceptually includes an abstract data type and a set of generic operators. The abstract data type defines input and output data structures for generic operators. This abstract data model for the MR programming model is the key-value (KV) pairs and the corresponding generic operators are *Map* and *Reduce*. The *Map* and *Reduce* execute user customized procedures on a list of KV instances. To apply MapReduce to a multidimensional array, array elements need to be converted into KV pairs. Common analysis operations, such as convolution [7,24], access a group of neighboring array elements to compute a new value. In array representation, these neighbors are near each other – we refer to this feature as structural locality [13,24]. This structural locality is lost when the values are represented as KV pairs. Although there are many ways to improve MapReduce systems [7,42], these intrinsic limitations still exist within its programming model.

Array databases (e.g., SciDB [33] and RasDaMan [2]) use multidimensional array as the native model. Their built-in operators, such as *window*, can capture structural locality, but they only allow performing uniform aggregation on all window cells, i.e., running non-discriminative operations (e.g., *sum*) on all window cells. Moreover, the definition of a *window* as *(starting indices, ending indices)* limits its shape to be rectangular bounding boxes. However, data analysis tasks on scientific data, such as computing gradients or slope [13,22], need different operations on distinct window cells, which together may form non-rectangular shapes. Blanas et al. [3] also show that loading scientific array data from their native file formats (e.g., HDF5 [38]) into array databases is a time-consuming and error-prone process.

Recent efforts, such as ArrayUDF [13], support user customized data analysis on array file formats and represent structural locality for the aforementioned data analysis operations. However, ArrayUDF lacks a formally defined programming model, which is essential for composing analysis tasks as discussed previously. ArrayUDF only allows operation on a single array with only one attribute, and its input and output arrays must have the same dimensions and the data have to be stored on disk. The lack of caching intermediate data in memory limits its performance with significant I/O overhead. TensorFlow [1] uses tensor abstraction to represent an array in machine learning applications. However, it only provides customized operations (e.g., *conv2d*) for specific tasks. Tuning existing operations and adding new ones still need non-trivial amount of code development for data management, parallelization and other tasks.

To address the gaps in composing various data analysis operations for multidimensional arrays, we propose a new "structural locality"-aware programming model (SLOPE), describe the design and implementation for its parallel execution engine, and present a thorough evaluation comparing with state-of-the-art Big Data systems. SLOPE offers a formally defined abstraction for users to customize various data analyses directly on multidimensional arrays. Each array, as an input of SLOPE, can have multiple attributes. The output array may have different dimensions from the input array. SLOPE has a distributed DRAM-based cache layer to stage the intermediate array of the analysis pipeline. In summary, contributions of this effort include:

- A formally defined structural locality-aware programming model (Sect. 3) and its parallel execution engine (Sect. 4.1). SLOPE has a novel new data structure – namely *Stencil* – and a generic operator (*SLApply*) to capture and express structural locality existing in multidimensional array data access and analysis. SLOPE allows customizable analysis operations directly on multidimensional array stored in native scientific data formats. A vectorization interface of SLOPE allows it to compose data analysis involving multiple attributes of one or more arrays.
- A distributed DRAM-based cache layer (Sect. 4.1) to store intermediate output arrays. SLOPE supports asynchronous checkpoint method for cached array. SLOPE also has a asynchronous halo exchange algorithm that synchronizes array cached in distributed DRAMs.
- Describe optimization features of SLOPE, such as mirror values and in-place data modification, to improve performance and semantics of data analysis on multidimensional array (Sect. 4.1).

We demonstrate the performance of SLOPE by using it to compose real scientific data analysis tasks on a supercomputer with $2,388$ Intel Xeon processors. These data analysis tasks come from applications: CAM5 [28,40], VPIC [8,22], and BISICLES [10,44]. We also compare SLOPE with state-of-the-art data analysis systems, such as Apache Spark [42] and TensorFlow [1]. SLOPE achieves up to ≈90,000× performance speedup over Apache Spark and ≈38% speedup over TensorFlow. We show that SLOPE can scale linearly as the data size and the number of CPU cores increase, achieving 512 GB/sec (≈128 GFLOPs) processing bandwidth using 16K CPU cores.

2 Preliminaries

2.1 Multidimensional Array

A d-dimensional array has dimensions $D = D_1, D_2, \ldots, D_d$ and a set of m attributes $A = A_1, A_2, \ldots, A_m$ [5]. Each dimension is a continuous range of integer values in $[0, N-1]$. Each indices $[i_1, i_2, \ldots, i_d]$ defines an array cell. All cells have the same data type for an attribute. An array can be expressed as a function mapping defined over dimensions and taking value attributes tuples: $Array : [D_0, D_1, \ldots, D_{d-1}] \mapsto < A_0, A_1, \ldots, A_{m-1} >$. In array, cells are stored in well-defined layouts, such as row-major and column-major orders. The row-major order is popular in scientific data formats, such as HDF5 [38] and NetCDF [21], and its last dimension is stored contiguously in storage devices. Offsets from the beginning to a cell at $(i_1, i_2, \ldots, i_d)$ is given by $\sum_{k=0}^{d-1} \prod_{l=k+1}^{d-1} D_l i_k$.

2.2 User-Defined Function and Programming Model

User-defined function (UDF) is a classic mechanism in database systems [26] to extend their query language. With the prevalence of MapReduce (MR), UDF evolves into a stand-alone *programming model* [11,32]. The MR has an abstract KV data model and two generic operators: *Map* and *Reduce*. *Map* and *Reduce* accept UDF defined on KV data model from users for different purposes. MR has achieved a lot of successes since its emergence. MR, however, has several intrinsic issues in supporting array data analysis:

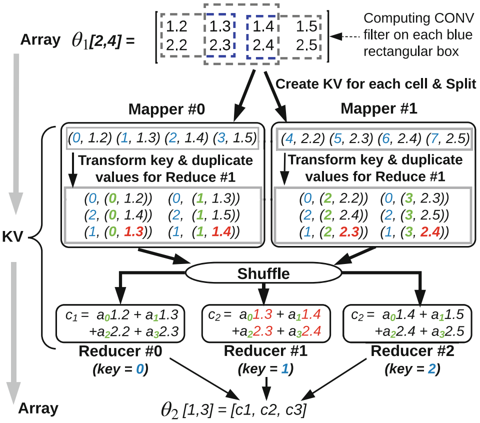

Fig. 1. An example of using MapReduce to compose convolution computing (*CONV*) from array θ_1 to θ_2. θ_1 is first linearized to a 1D KV list. *Mappers* then unify the key of KV pairs belonging to the same CONV filters (i.e., blue dashed rectangular boxes). Based on the index of a cell in original rectangular box, a secondary key (light green) is added to find corresponding filter weight in later *Reducers*. Also, KV pairs for *Reducer #1* (denoted as red) are added via duplicating existing values. After shuffle, *Reducers* perform a weighted summary to compute *CONV* ($\{a_0, \ldots, a_3\}$ are weight). Obviously, MapReduce is an inefficient programming model for array because of the conversion between array and KV and the duplication of values for *Reducers*. We propose a new programming model that enable users to directly customize and execute operations from θ_1 to θ_2 without data model conversion and with negligible duplications. (Color figure online)

- **KV data model makes MapReduce perform poorly on multidimensional arrays.** Multidimensional arrays have to be linearized into KV list before using them as the input of MapReduce. First, this linearization may need to explicitly handle array index. Second, the linearization breaks the array structural locality, which exists in many data analysis operations such as convolution [19] and gradient computing [13]. In Fig. 1, we give an example of using MapReduce to compute convolution on a 2D array and discuss its problems. The discrete convolution [6] is expressed as $*$ operator: $(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m)$, where f, g are two complex-valued functions on the set Z of integers. For a single convolution on array, it needs a few neighborhood cells. Linearization breaks the structural locality by scattering these neighborhood cells in different Mappers. The penalty is the extra cost to rebuild this structural locality though *Shuffle* and *Reduce*. Although using linearization methods (e.g., SFC [35]) may preserve structural locality, developing these methods and converting operations originally defined on multidimensional array to ones on KV are still cumbersome tasks for users.

- **A set of KV pairs processed by a *Reducer* is independent from the set of KV pairs processed by another *Reducer*.** Assuming $S1$ and $S2$ are two sets of KV pairs belonging to two *Reducer*s, it is common to have intersection $S1 \cap S2$ in use cases, such as convolution [19]. Because MapReduce schedules and executes these two *Reducer*s separately, duplicating KV pairs in $S1 \cap S2$ for two *Reducer*s are required to make parallel processing on *S1* and *S2* work properly. As shown in Fig. 1, the input KV pairs for Reducers #1 are added by MapReduce through duplication. These duplicated KV pairs consume memory for storing and network bandwidth for shuffling and therefore degrade overall data analysis performance.

3 SLOPE Programming Model

As discussed above, multidimensional arrays and KV pairs are different data models. Using the MapReduce to handle multidimensional arrays requires converting them into a KV list. Therefore, it is critical to explore a programming model with multidimensional arrays as the first-class citizens. Toward that goal, we focus on addressing the following question: what are the abstract data types and generic operators for a comprehensive programming model supporting data analytics on multidimensional arrays? We answer these questions with the proposal of SLOPE that contains a new abstract data type, called *Stencil* and a generic operator *SLApply*. Details are reported in following subsections.

3.1 Abstract Data Type—*Stencil*

An abstract data type represents both the input and output data structures of generic data analytic operators in a programming model. In other words, the abstract data type describes the data structure for the smallest subsets of array data in composite data analysis. In this work, we present a new abstract data type, called *Stencil*, which is inspired by the stencil-based numerical computing [14]. Being different from these existing work, we focuses on generalizing these ideas for modern data analysis (e.g., convolution neural network). Conceptually, a *Stencil* can represent a geometric neighborhood of an array, which further contains a *center* cell and a few neighborhood cells at different *relative offsets* from the center. We use the ***absolute index*** (c) of an array cell to refer the *center* cell and use the ***relative offsets*** $(\vec{o})$ to represent the neighborhood cells, which can be expressed as $(c, \vec{o})$. For example, in a 2D array, a geometric neighborhood expression using *Stencil* can be $\Big((1,1), \big((0,0), (0,-1), (0,1), (-1,0), (1,0)\big)\Big)$, where $c = (1,1)$ is the absolute index of the center cell and $\vec{o} = \Big(\mathit{(0,0),\ (0,-1),\ (0,1),\ (-1,0),\ (1,0)}\Big)$ contains relative offsets for the center cell itself, the cell on the left, the cell on the right, the cell on the top, and the cell below. This geometric neighborhood is visually shown in Fig. 2(b).

In the following description, we use the symbol S to refer an instance of *Stencil* and use $S_{o_1,o_2,\ldots,o_d}$ to refer a single cell at the relative offset $[o_1, o_2, \ldots, o_d]$. In the above example, $S_{0,0}$ is the center cell and $S_{0,-1}$ is the cell at the left. The value of the cell in a *Stencil* is represented by dot "." operator on the symbol S. For example, during an array with multiple attributes A_j ($j \in [1, m]$), the value of the attribute A_j is represented as $S_{o_1,o_2,\ldots,o_d}.A_j$. The value of all attributes is represented as a vector $\overrightarrow{S_{o_1,o_2,\ldots,o_d}.A}$ or $\overrightarrow{S.A}$. In the array with a single attribute, we use $S_{o_1,o_2,\ldots,o_d}$ to represent its value by omitting the dot operator and attribute name. We conclude the properties of *Stencil* as follows:

- **A *Stencil* abstraction provides a new way to logically partition a large array into subsets.** Compared with chunking in array databases [31], *Stencil* splits array into even smaller subsets for processing. *Stencil* obeys the structure locality of array data access by allowing relative offset from a center cell, which is important in many data analytic operations [13].
- **A *Stencil* has flexible size**. By taking a *2 by 2* array as an example, $S_1 = \Big(\mathit{(0,0),}\ \big(\mathit{(0,0)}\big)\Big)$ contains the first cell. $S_2 = \Big(\mathit{(0,0),}\ \big(\mathit{(0,0),\ (0,1),\ (1,0),\ (1,1)}\big)\Big)$ represents the whole array. Such flexibility enables users to group any number of desired cells for analysis.
- **Cells within a *Stencil* can form any geometric shape.** One can specify any cell as the center of a *Stencil* and add any other cell into the *Stencil*. So, the cells within a *Stencil* can form any shape. A few examples of typical *Stencil*s are visually presented in Fig. 2, which will be further discussed in

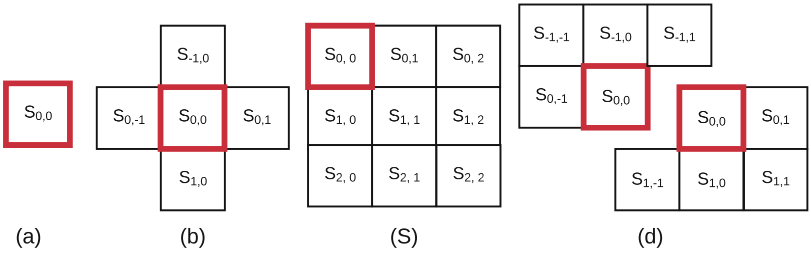

Fig. 2. Example of geometric neighborhoods expressed by *Stencil* on a 2D array: (a) a single cell neighborhood; (b) a neighborhood containing a center cell and four surrounding cells; (c) a neighborhood contains a center cell and its neighbors at right-bottom; (d) a forward-scan neighborhood (at top) and a backward-scan neighborhood (at bottom). These examples are presented from the point-view of users. The center of each is ignored and will be automatically handled by the execution engine in following Sect. 4. Usages of these examples in real applications are presented in Sect. 3.3.

the following Sect. 3.3. By comparison, *window* in SciDB is defined as *(start indices, end indices)* and it can only represent rectangular shape. Compared with *sliding* idea of the Spark [42] on a 1D KV list and the *window* idea of SQL database systems [26,41] on relational tables, our *Stencil* can express more diverse geometric shapes on top of a multidimensional array.

3.2 SLOPE Programming Model

The *Stencil* abstract data type provides a flexible way to represent a small set of array cells with structural locality. We use it as the input and the output data type of a generic operation in our programming model. Hence, we refer our programming model as Structural Locality-aware Programming Model (SLOPE). Inspired by the *Apply* primitive in R [15] and in several other languages, our SLOPE has a generic operator, named Structural Locality Apply, or *SLApply* for short. Hence, the SLOPE model can be represented as the following two-element tuple: (*Stencil*, *SLApply*), where *Stencil* is the generic data type and *SLApply* is the generic operator. Given two arrays θ_1 and θ_2, the generic operator *SLApply* is expressed as,

$$SLApply : \overrightarrow{S} \quad \overset{f}{\mapsto} \quad \overrightarrow{S'}, \quad S \subset \theta_1, S' \subset \theta_2, \tag{1}$$

Semantically, *SLApply* maps the *Stencil* instance $\overrightarrow{S}$ to the result *Stencil* instance $\overrightarrow{S'}$. The $\overrightarrow{S}$ and $\overrightarrow{S'}$ represents geometric neighborhoods from input array θ_1 and output array θ_2, respectively. The UDF f defines the desired operation for the map. Within function f, users can use the *Stencil* abstraction and its member cells to describe any desired operation. The → (hat) symbol on *Stencil* means both input and output can be a vector *Stencil* across all attributes of input and output arrays. In following parts, we uses $\theta_2 = SLApply(\theta_1,\, f)$ to represent the execution of function f from θ_1 to θ_2. Users also control the return value of function f which is used to initialize output *Stencil*.

In SLOPE implementation, the *SLApply* internally executes the function f on all instances of *Stencil* abstraction. There are two ways for *SLApply* to create *Stencil* instances and execute function f:

- By default, *SLApply* creates *Stencil* instance for each array cell in θ_1. Then, it executes f on each *Stencil* instance. This default method works for most data analysis tasks (such as *filter*) which run an operation on the whole array. The value of output *Stencil* is the return value of function *f*. The output *Stencil*s inherit the coordinate of the center cell of the input *Stencil*.
- Additionally, *SLApply* can create *Stencil* instances for certain cells and execute f only on them. For example, users can specify a skipping parameter on θ_1 to avoid creating *Stencil* instances, i.e., calling function f for certain cells. One example operation is convolution computing which may run filter on every two cells on each dimension.

In most cases, output $\boldsymbol{S'}$ only has a single cell, but it can also include a set of neighborhood cells. By default, the semantic of *SLApply* only allows "read only" on original array θ_1. But SLOPE provide users an option namely *"In-place Modifications"* (in Sect. 4.5) to change the semantic to update θ_1 as the propagation of f onto the whole array.

```
//User customized functions
Gradient(S):
    return $4S_{0,0} - (S_{-1,0} + S_{0,-1} + S_{0,1} + S_{1,0})$

$\theta_2 =$ SLApply($\theta_1$, Gradient)

Interpolation:(S)
    return BilinearInter($S_0.X$, $S_0.Y$, $\theta_2$)

SLApply($\theta_3$, Interpolation)
```

Fig. 3. Gradient computing and interpolation with SLOPE on a 2D field data θ_1 and on a 1D particle data θ_3. The θ_3 has two attributes X and Y. It first invokes *SLApply* to execute *Gradient* on θ_1. Then, the *Interpolation* is applied onto θ_3, which uses θ_2 from first *SLApply* too. For simplicity, we use *BilinearInter* (omitted) to denote the interpolation formula. In parallel execution, θ_2 can be cached in memory and broadcast to all processes for performance.

```
//Three user customized functions
CONV(S):
        return $[S_{0,0}, S_{0,1}, \dots, S_{2,2}] \begin{bmatrix} a_{0,0} & \dots & a_{0,k} \\ \vdots & \vdots & \vdots \\ a_{8,0} & \dots & a_{8,k} \end{bmatrix}$
POOL(S):
        return $max([S_{0,0,0}, S_{0,0,1}, ...S_{0,0,2}])$
ReLU(S):
        return $max(0, S_{0,0,0})$

//Execution by chaining three SLApplys on array $\theta_1$
SLApply(SLApply(SLApply($\theta_1$,CONV),ReLU), POOL)
```

Fig. 4. Expression of CNN on a 2D array θ_1 with SLOPE. The convolution has k kernels whose parameters are $a_{ij}, i \in [0, 8]$ and $j \in [0, k]$. The output of each *CONV* is a vector, turning θ_1 into a 3D array. *CONV* is applied onto array θ_1 at first. Then, *ReLU* and *POOL* are applied onto the result of *CONV*. Users can define and add more layers to have a deep architecture.

3.3 Example Data Analysis Using SLOPE

SLOPE can express various single data analysis operations, such as *filter, moving average*, etc. Here we use SLOPE to compose a few advanced data analytic algorithms with multiple steps:

- **Gradient and interpolation computations**. Magnetic reconnection is an important phenomenon in plasma particle acceleration research [22]. Data analysis pipeline used to study the reconnection phenomenon involves computing gradients on the magnetic field mesh data and then interpolate the gradient value for all particles scattered in space. An example of using SLOPE to express this data analysis pipeline on a 2D filed data is shown in Fig. 3. The gradient procedure uses the *Stencil* shown in Fig. 2(b). The interpolation uses the *Stencil* from Fig. 2(a).
- **Convolutional neural network (CNN)**. CNN [19] is a deep feed-forward artificial neural network, typically used in recognizing images. A CNN has been applied to identify extreme weather events [27]. CNN usually includes three main layers: convolution, ReLU, and pooling. Figure 4 shows expressions of these three layers with SLOPE. The *Stencil* for convolution and pooling is shown in Fig. 2(c). The ReLU uses a single cell *Stencil*, as shown in Fig. 2(a).
- **Connected Component Labeling (CCL)**. CCL [34] is one fundamental algorithm in pattern recognition. CCL assigns each connected component member with a unique label. The standard CCL repeatedly applies an 8-way connection detection for each point to replace its label with the minimum label of its neighbors. CCL converges until no such replacement happens. To accelerate the converge process, advanced CCL algorithms break the connection detection into forward and backward passes, as shown in Fig. 5. The *Stencil* in Fig. 2(d) are used during forward and backward passes in advanced CCL.

4 Parallel Execution Engine

This section introduces a parallel execution engine for SLOPE. We especially focus on following problems: (1) how to enable the user-defined operation defined on *Stencil* to run parallel. (2) how to cache intermediate data of analytic pipeline in DRAM to avoid expensive I/O operations.

4.1 Overview of Parallel Execution Engine

SLOPE execution engine follows single program and multiple data (SPMD) pattern [16,29,30], where multiple SLOPE processes are launched with the same analysis program and each process handles different data. Within a process, an *SLApply* executes an user-defined operation. An example of SLOPE on three nodes processing a 2D array in parallel is shown in Fig. 6. The input array is partitioned by SLOPE into chunks. SLOPE augments each chunk with halo layers

```
//Two user customized functions
$ForwardScan(S)$:
    return $min(S_{0,0}, S_{-1,-1}, S_{-1,0}, S_{-1,1}, S_{0,-1})$
$BackwardScan(S)$:
    return $min(S_{0,0}, S_{0,1}, S_{1,0}, S_{1,1}, S_{1,-1})$

//Repeatedly executing two $SLApply$s on array $\theta_1$
do
    $\theta_1 = SLApply\Big(SLApply(\theta_1,\ ForwardScan),\ BackwardScan\Big)$
while (NOT converged)
```

Fig. 5. CCL algorithm on a 2D array θ_1 expressed with SLOPE. It includes a backward scan and a forward scan. The converged status is reached when no new label is found for all cells. For simplification, we omit the functions to detect converged status and to switch execution direction.

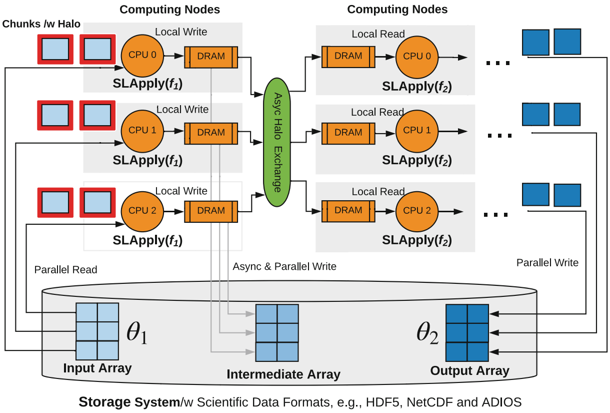

Fig. 6. Example of SLOPE execution for two user customized analysis functions (f_1 and f_2) on a 2D array θ_1. The θ_1 is split into 6 (3×2) chunks and read into 3 computing nodes for parallel processing. The intermediate array from f_1 is cached locally in memory for f_2 to read. Before actual read, a halo exchange algorithm is executed to augment cached chunks with halo layer. The intermediate array can be written onto disk for fault tolerance. The final output array is θ_2. Both θ_1 and θ_2 are stored as scientific data formats, e.g., HDF5, in storage systems.

(or called ghost zones) to avoid possible accessing cells from different nodes during execution. The output array of *SLApply* can be cached in memory. Caching the output from the previous *SLApply* allows the following *SLApply* to quickly access input data. SLOPE execution engine uses a halo exchange algorithm to synchronize each cached chunk for cache array. While storing the intermediate array in memory is prone to software and hardware failures, SLOPE supports asynchronously checkpointing intermediate arrays to persistent storage system.

4.2 Data Partitioning and Halo Layer

An array is split into chunks for parallel processing in SLOPE. The chunk in SLOPE is defined as *(starting indices, ending indices)*, where *indices* is a vector. SLOPE augments each chunk with a halo layer. The halo layer eliminates the need to retrieve cells from other chunks. Users can specify the size of the halo layer as the maximum offset used by *Stencil* in the user-defined function f. Note that the halo layer in SLOPE introduces a small amount of duplicated cells at boundary. However, these duplications has a negligible performance impact since halo layer is usually smaller than a chunk. In contrast, MapReduce (as shown in Fig. 1) duplicates KV instances for each operation across the whole chunk, which significantly increases data size to be processed.

4.3 Data and Computing Scheduling

The scheduling method in SLOPE assigns chunks among processes with the structural locality at chunk level to reduce the ghost zone exchanges when the array is needed by following *SLApply*s. By default, array chunks are linearized by the row-major order and the linearized index is used as the 'ID' of the chunk (denoted as id_{chunk}). Given p processes, the chunk id_{chunk} is assigned to the process at rank $\lfloor id_{chunk}/\lfloor \frac{id_{chunk}}{p} \rfloor \rfloor$. SLOPE also allows users to choose the assignment in the reverse direction, i.e., from the end to the beginning along linearized order. Within a single chunk, cells are also scheduled by their row-major order.

4.4 Output Array Dimension

Mostly, the output array from a *SLApply* has the same number of dimensions and size as the input array. However, the output array may have different number of dimensions and size. SLOPE detects these attributes for output array based on information extracted from input array and the user customized function on it. For example, the convolution operation (in Fig. 4) converts a 4×4 2D data to an $4 \times 4 \times 8$ 3D array. In the *CONV* function, the return value for a single array cell is a vector with 8 cells. *SLApply* detects the size of this return vector and uses it as the size of the third dimension in output array.

4.5 Advanced Features

In SLOPE, we also provide various advanced and optional features that may apply only to selected data analytic tasks.

Intermediate Array Cache. To support efficient data movement between multiple *SLApply* operations, we support caching [43] intermediate data on distributed memory instead of storing on file systems. The in-memory cache layer has a metadata table containing array dimensions, cell type, and other array related attributes [36]. This metadata table is created when the intermediate array is produced through the output array dimension reasoning method discussed before. Based on the SPMD pattern of SLOPE, each SLApply can have its own copy of metadata table to avoid communication when it writes or reads data. Intermediate arrays are stored as chunks and each is produced by its corresponding input chunk on the same process. Each chunk does not contain the halo layer when it is produced. SLOPE builds halo layers for the following *SLApply* when needed using an **asynchronous halo exchange** method. SLOPE uses classic halo exchange algorithm [18] but we improve it with MPI asynchronous primitives *MPI_Isend* and *MPI_Ireceive* to improve its performance.

Asynchronous Check Point Method. Caching intermediate arrays in memory is prone to data loss because of potential hardware or software faults [20]. SLOPE allows to asynchronously checkpoint intermediate arrays [37]. SLOPE provides interface for users to control the checkpoint frequency, which can be calculated via: $\frac{V}{T \times E}$, where V is the data size, T the bandwidth of storage system, and E the time of executing a *SLApply*.

Mirror Value. Invoking a user customized function incurs overhead. In reality, there are some operations that have defined mapping from the input value to the output values, e.g., values greater than or equal to zero in $ReLU$ used by CNN. The SLOPE introduces a mirror value feature to allow *SLApply* to skip invoking customized function for these points.

In-Place Modification. During real-world data analysis, computing on a cell may need the result from previous cells. For example, in CCL, the label of the previous point can be used to find the label for the current one. To optimize for these operators, SLOPE allows users to replace input of *SLApply* with its output. By contrast, Spark does not allow analysis operation to modify the original data, which may create lots of RDDs in long pipeline.

Multiple Arrays. SLOPE allows users to compose data analysis on multiple arrays. For example, the user-defined Interpolation function in Fig. 3 describes the operations on array θ_3 and on array θ_2. This operation is similar to the map-side join in MapReduce.

4.6 Implementation of SLOPE

Based on ArrayUDF [13], we provide an implementation[1] of SLOPE in plan C++ with its template feature to support different data types. Specially, it provides two C++ classes, e.g., *Stencil* and *Array*, which are included in header file "*ArrayUDF.h*". The *Stencil* class implements abstract type *Stencil* in Sect. 3.1. The *Array* contains all functions related to execution engine in Sect. 4. The *Array* has a method namely *Apply* (i.e., *SLApply*) to run UDF on its data. The UDF is standard C++ function and is passed to *SLApply* as function pointer. An example for using SLOPE to implement CCL code (with a single step and 8-way check) is presented in Fig. 7. User can use standard C++ compiler to compile it. MPI is required to run it in parallel for both intra-node and inter-node.

[1] https://bitbucket.org/arrayudf/.

```
#include <ArrayUDF.h>
//Define function to find minimum label for a Cellhood
float  ccl_check(Stencil<float> c){
          return Min(c(-1,-1), c(-1, 0), c(-1, 1), c(0,-1),\
                     c( 0 ,0), c( 0, 1), c( 1,-1), c( 0, 1), c(1, 1));
}
//Initialize chunk_size & ghost_size for parallel processing
//Initialize Array instance A with a 2D array "data" from
//      a HDF5 file "file.h5"
//Apply ccl_check on array A, output is ignored
void main(){
     vector<int>   chunk_size{10,10}, ghost_size{1,1};
     Array<float>  A("file.h5", "/data",chunk_size,ghost_size);
     A->Apply(ccl_check);
}
```

Fig. 7. SLOPE C++ example code for CCL (with a single step and 8-way check).

5 Evaluation

We demonstrate the effectiveness of SLOPE on the Cori supercomputer at The National Energy Research Scientific Computing Center, or NERSC[2] with over 2,400 computing nodes. We compare SLOPE with following systems:

- **Apache Spark** [42] represents state-of-the-art MapReduce. We use new technologies including H5Spark [23] and file pooling [9] to realize a fair comparison between Spark and SLOPE on supercomputer. We also have a test in non-supercomputer environment to compare Spark and SLOPE.
- **ArrayUDF** [13] provides a native user-defined function on array. However, it lacks a clearly defined programming model and supports for multiple attributes, multiple arrays and in memory cache. We use a few customized operations to compare ArrayUDF to match its capability.
- **TensorFlow** [1] includes hand-tuned and public available procedures for data analytics. The goal to use TensorFlow in tests is to show how fast/slow the analytic tasks that are expressed through Spark and SLOPE.
- **C++ Imp**, developed by authors, represents the way users implement data analytic function without programming model. C++ Imp only implements the data analytic logics (e.g., convolution) with hand-tuned codes (e.g., directly calculating convolution on array in memory). The C++ Imp has the same data management (e.g., cache) as we have done in SLOPE.

5.1 Evaluation Using Synthetic Data Analysis

Our evaluation in this section uses a two-layer convolutional neural network (CNN) on a 2D synthetic float typed array with the dimension of *64,000 by 64,000*. Specifically, we focus on a single forward pass computing of the CNN. Unless otherwise noted, all experiments use 256 CPU cores and *4096 by 4096* chunk size. The size of halo layer is set to 1. The two-layer CNN has a *convolution (CONV)* layer and a *ReLU* layer. *CONV* is configured with a *8 2 by 2* kernels. *ReLU* is: $f(x) = max(0, x)$, where x is the input to a neuron.

[2] https://www.nersc.gov/.

Comparing SLOPE with ArrayUDF. We use customized workloads from CNN described above to match the capability of ArrayUDF. Specifically, we use a single convolution kernel to keep both input and output arrays have the same size, which is supported by ArrayUDF. As discussed before, SLOPE allows input and output arrays to have different sizes. We find out that cache feature of SLOPE enable it to be at most 10× faster in accessing data. In following parts, we will focus on comparing SLOPE with Spark and hand-tuned code.

Comparing SLOPE, Spark, TensorFlow, and C++ Imp. This test uses a 2 layered CNN workload. TensorFlow uses *tf.nn.conv2d* and *tf.nn.relu* for *CONV* and *ReLU*, respectively. Spark uses a chain of *Map* and *Reduce* to express *CONV* and *ReLU* (as shown in Fig. 1). The expression of *CONV* and *ReLU* in SLOPE is presented in Fig. 4. To isolate impact of supercomputer software, such as MPI and Lustre, we have two separate tests on non-supercomputer environment and supercomputer environment, as discussed below:

- **Non-supercomputer environment.** This test uses a single Linux box. For Spark, in order to clearly measure performance for each step, we test it without its Lazy Evaluation (LE) optimization [42]. We report test results in Fig. 8. Spark without LE spends 11% of time to read data and 38% of time to shuffle data. As explained in Fig. 1, the read function accesses data from the disk and converts the array into KV pairs. The shuffle function aggregates the inputs for *Reduce*. Both read and shuffle are expensive. Spark's LE optimization improves its performance by 3X. Based on this observation, we test Spark only with the LE optimization in following sections. The "C++ Imp" has the best performance. TensorFlow is ≈15% slower than SLOPE. TensorFlow may implement *CONV* as expensive matrix multiplication [1]. Overall, SLOPE has comparable performance as hand-tuned code and is much faster than Spark.
- **Supercomputer environment.** On a parallel environment with Lustre and MPI, we scaled above tests from 1 CPU core to 16 CPU cores. Note that, to be fair, we only consider the computation time of these systems because we believe TensorFlow and Spark are not originally designed and optimized for Lustre. For Spark, we force the read operation to complete before *Map* and *Reduce* starts. The results are presented in Fig. 9. The trend in these results are consistent with the one from non-supercomputer environment. On average, SLOPE is 13× faster than Spark. SLOPE maintains structural locality in the assignment of array cells and partitions, eliminating the need for communication. Spark has to shuffle data between *Map* and *Reduce*. SLOPE is 38% faster than TensorFlow. As explained in previous paragraph, TensorFlow may convert *CONV* into large and expensive matrix multiplications.

Scalability Tests of SLOPE. We evaluated both weak scaling and strong scaling of SLOPE using the number of CPU cores from 256 to 16K. In weak scaling tests, the data size for each process is fixed at 64 MB, totaling 1 TB at 16K CPU cores. In strong scaling tests, the file size is fixed at 1 TB. We use the parallel efficiency to measure the result. The parallel efficiency for strong scaling is defined as

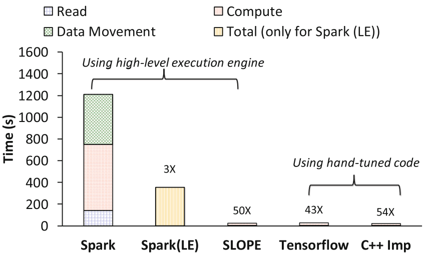

Fig. 8. Comparing SLOPE with Spark and hand-tuned codes to perform *CONV* for a 2D array on a Linux server. The Spark was tested with and without Lazy Evaluation (LE).

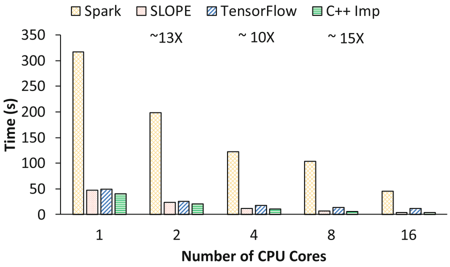

Fig. 9. The computing time (only) for executing CNN with SLOPE, Spark and hand-tuned codes (i.e., TensorFlow and C++ Imp) on a supercomputer.

$t_1/(N*t_N)*100\%$, where t_1 is the time to finish a work unit with 1 process and t_N is the time to finish the same work unit with N process. For weak scaling, its parallel efficiency is $t_1/t_N*100\%$. Results are presented in Fig. 10. Both cases show that SLOPE has high parallel efficiency. Actually, processing a 1 TB data takes around 2 s, giving around 512 GB/sec throughput at 16K CPU cores. The primary reason SLOPE can achieve such scalability is that it can maintain structural locality on processing multiple dimensional array and all computing can happen on a local node without communication across nodes.

5.2 Evaluation for SLOPE and Spark Using Real Applications

In this section, we compare SLOPE with Spark in executing data analysis operations for three applications: CAM5 [28,40], VPIC [8,22], and BISICLES [10,44]. All datasets are stored in the HDF5 files. Spark uses H5Spark [23] to read HDF5 data into RDDs. Since H5Spark does not support writing RDDs back to HDF5, we compare read performance and execution performance.

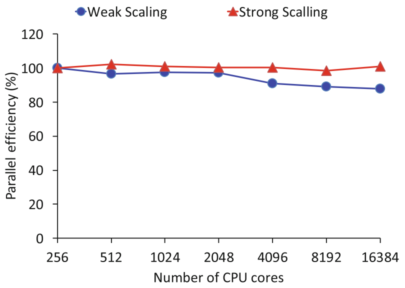

Fig. 10. Strong and weak scaling test results for SLOPE with upto 16K CPU cores.

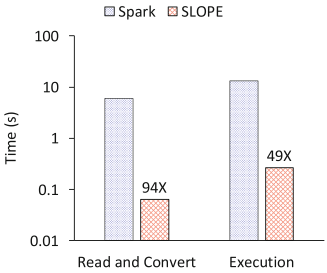

Fig. 11. Test results of executing CAM5's CNN with SLOPE.

CAM5's Convolution Encoder. The CAM5 is widely used by the climate community to study the global climate change and prediction [40]. The dataset in test is from a simulation run with 25 km spatial resolution for the years spanning from 1979 to 2005. Each frame of the global atmospheric state is a 768×1152 image. CNN [28] has been used to predict extreme weather events, such as atmospheric rivers, from this data. We have used SLOPE and Spark to compose three key steps of CNN: *CONV*, *ReLU*, and *Pooling*. Following the work [28], we consider the layer close to the Earth's surface and the variables, namely *TMQ*, for atmospheric rivers detection. We use the output data for the year *1979*, which is a 768×1152 2D array. Spark uses two sets of *Map* and *Reduce* to express *CONV* and *Pooling* layers, respectively. The *ReLU* is expressed with a *Map*. The implementation with SLOPE is shown in Fig. 4. Test results are reported in Fig. 11. The read time required by Spark is around 94× slower than that of SLOPE. This performance benefit is because SLOPE directly handles multidimensional array without flattening it into KV pairs. Spark needs to convert the array into KV pairs during read. When comparing the execution time, SLOPE is 49× faster than Spark. Also, as discussed in the previous section, linearization to a 1D KV causes significant communication overhead for *Reduce* to gather inputs. SLOPE reduces communication to synchronize only a small number of halo layers with the preserved structure locality.

VPIC's Gradient and Interpolation Operations. VPIC simulates the magnetic reconnection phenomenon [22]. Magnetic reconnection study involves two key steps: calculating gradient on 3D field mesh and finding gradient field value for each particle via interpolation. These operations involve four arrays, M, X, Y, and Z, where M is a 3D magnetic field mesh data and X, Y and Z contains particle locations. The gradient computing for M uses a Laplace operation, i.e., 3D version of the one in Fig. 3. Using Spark, the gradient on M is expressed with a *Map* and a *Reduce*, where the *Map* duplicates each cell for its neighbors and the *Reduce* operation performs the Laplace calculation. Then, a tri-linear operation is finished with a map-side join, where the gradient value of M is broadcasted to each executor and then a *Map* is used to find the gradient field value of each particle. Implementation with SLOPE is a 3D version of the algorithm in Fig. 3. Since Spark has a limit on the size of broadcast data, we have set the test to use a small 256 MB ($512 \times 256 \times 256$) field data. The particle data has 263 GB with ∼23 billion particles. The tests used *128* CPU cores on 16 nodes. A performance comparison is shown in Fig. 12, where SLOPE performs 106× faster than Spark to execute the analysis. In reading the input data, SLOPE is 45× times faster. Since Spark has to duplicate a lot (∼6×) cells to help *reduce* to calculate the Laplace operator, its performance is poor. In contrast, SLOPE uses logical partitioning without duplications to finish the Laplace operator. Explicitly processing and communicating array index for particle data in Spark consumes CPU time and network bandwidth and therefore degrades its performance.

BISICLES's CCL Algorithm. Connected component labeling (CCL) has been used to detect ice calving events on Antarctica, simulated by BISICLES [44]. We have used both Spark and SLOPE to compose CCL. The data size

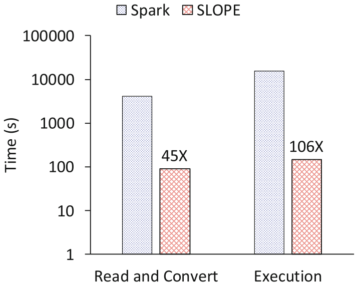

Fig. 12. Test results of executing VPIC's gradient & interpolation analysis with SLOPE.

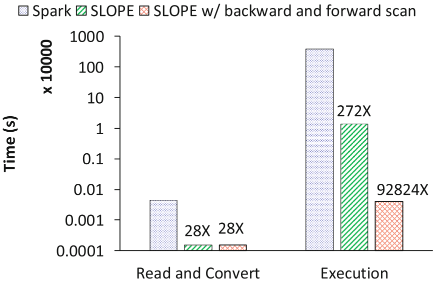

Fig. 13. Test results of executing BISICLES's connected component labeling (CCL) with SLOPE.

is 12288 × 12288 and we test it with 16 CPU cores on 4 nodes. We implemented a standard multi-pass algorithm for CCL with an 8-way connection detection [34] for both Spark and for SLOPE. We report the execution time in Fig. 13. The 8-way connection detection based CCL algorithm needs 10868 iterations to converge and SLOPE is 272× faster than Spark. The main reason for Spark's slower performance is that it needs to duplicate a large number of array cells for neighborhood-based connection detection, while SLOPE needs to duplicate only a small halo layer. SLOPE also can use advanced features, such as in-place data modification and back-forward execution, which can accelerate the convergence of the CLL algorithm [34] (in Fig. 5). The CCL algorithm with these advanced features can not be implemented with Spark as its immutable RDD concept, where in-place data modification is not allowed. By enabling the feature in SLOPE, it converges after 8 steps. Comparing the performance of Spark, SLOPE is 92824× faster.

6 Related Work

Relational [26,41] and array [2,5,24] DBMS provide built-in operators and a UDF mechanism for customization operations. Our SLOPE shares certain similarity with these systems in expressing UDF operation. SLOPE differs from them in its new abstract data type *Stencil* and programming model. The *Stencil* of SLOPE supports a flexible way to logically subset array into any shapes for analytic operations. In DBMS, their *window* operator only subsets array into rectangular boxes and each member in *window* operator is treated equally. In contrast, each member of *Stencil* can be customized with different operations in SLOPE.

MapReduce has a KV data type and two generic operations, *Map* and *Reduce*. Spark [42] introduces a memory cache layer for iterative analysis. SciHadoop [7] provides a scheduling optimization for adopting MapReduce to analyze data in arrays. Our SLOPE has a structural locality-aware programming model on multidimensional array and generalizes both *Map* and *Reduce* into a single operator, *SLApply*, on array. SLOPE has the similar cache mechanism as the one in Spark for KV but the cache in SLOPE works for multidimensional array. The SciHadoop has similar goal as SLOPE. But SciHadoop still uses MapReduce to compose analysis and SLOPE is a new programming model.

Stencil domain-specific languages (DSL) [12,17,25] are mostly developed to solve partial differential equation(PDE) problem. Our SLOPE is a data analysis framework that generalizes MapReduce for multidimensional arrays. SLOPE uses Stencil idea to extend the Map functions (on a single element) to a set of neighborhood array cells towards composing data analysis tasks, e.g., connected component labeling (CCL). Technologically, SLOPE is implemented by plain C++ without compiler extension/new language definition. Stencil DSL may need compiler extension or even new language definition.

SAGA [39] supports aggregator operation on array file formats. R [15] uses *Apply* operators and its parallel version *dmapply* to customize operations over a list or a vector. But, none of these *Apply* operators can express the structure locality on array. ArrayUDF [13] provides a simple API to accept UDF but it can only run a single UDF on a single array stored on disk. TensorFlow [1] provides hand-optimized code for machine learning but has not programming model defined. SLOPE defines programming model and in-memory cache to compose complex analytic operations with multiple stages and multiple arrays.

7 Conclusions and Future Work

Data analysis operations on multidimensional arrays are typically defined on a set of neighborhood cells in diverse geometric shapes. This structural locality is lost in the programming models, such as MapReduce, making them perform poorly in composing deep data analytic pipeline for multidimensional arrays. In this work, based on multidimensional array data model, we proposed a new structure locality-aware programming model (SLOPE) and its implementation. We also present multiple data analysis examples with SLOPE, such as convolutional neural network, gradient computing, and connected component labeling. In tests with real scientific data analysis, SLOPE is 49× $\sim$ 92824× faster than Spark, which represents state-of-the-art MapReduce programming model. In our future work, we plan to expand SLOPE to compose diverse operations without strong locality of reference.

Acknowledgment. This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231 (program manager Dr. Laura Biven). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

References

1. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI 2016 (2016)
2. Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., Widmann, N.: The multidimensional database system RasDaMan. SIGMOD Rec. **27**(2), 575–577 (1998)
3. Blanas, S., Wu, K., Byna, S., Dong, B., Shoshani, A.: Parallel data analysis directly on scientific file formats. In: SIGMOD 2014 (2014)
4. Bloom, J.S., Richards, J.W., et al.: Automating discovery and classification of transients and variable stars in the synoptic survey era. PASP **124**(921) (2012)
5. Brown, P.G.: Overview of SciDB: large scale array storage, processing and analysis. In: SIGMOD (2010)
6. Brown, P.G.: Convolution is a database problem (2017)
7. Buck, J.B., Watkins, N., et al.: SciHadoop: array-based query processing in Hadoop. In: Supercomputing Conference (SC) (2011)
8. Byna, S., Chou, J., Rübel, O., Prabhat, Karimabadi, H., et al.: Parallel I/O, analysis, and visualization of a trillion particle simulation. In: SC (2012)
9. Chaimov, N., Malony, A., Canon, S., Iancu, C., et al.: Scaling spark on HPC systems. In: HPDC 2016 (2016)
10. Cornford, S.L., et al.: Adaptive mesh, finite volume modeling of marine ice sheets. J. Comput. Phys. (2013)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
12. Denniston, T., Kamil, S., Amarasinghe, S.: Distributed halide. SIGPLAN Not. **51**(8), 5:1–5:12 (2016)
13. Dong, B., Wu, K., Byna, S., Liu, J., Zhao, W., Rusu, F.: ArrayUDF: user-defined scientific data analysis on arrays. In: HPDC (2017)
14. Durlofsky, L.J., Engquist, B., Osher, S.: Triangle based adaptive stencils for the solution of hyperbolic conservation laws. J. Comput. Phys. **98**(1), 64–73 (1992)
15. The R Foundation: The R Project for Statistical Computing. https://www.r-project.org/
16. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996)
17. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: a domain-specific tool for structured grid methods in weather and climate models. In: SC 2015 (2015)
18. Laoide-Kemp, C.: Investigating MPI streams as an alternative to halo exchange. Technical report, The University of Edinburgh (2014)
19. Lecun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
20. Li, D., Vetter, J.S., Yu, W.: Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In: SC (2012)
21. Li, J., Liao, W.-K., Choudhary, A., et al.: Parallel netCDF: a high-performance scientific I/O interface. In: SC 2003, p. 39. ACM, New York (2003)
22. Li, X., Guo, F., Li, H., Birn, J.: The roles of fluid compression and shear in electron energization during magnetic reconnection (2018)
23. Liu, J., Racah, E., Koziol, Q., et al.: H5Spark: bridging the I/O gap between spark and scientific data formats on HPC systems. In: Cray User Group (2016)
24. Marathe, A.P., Salem, K.: A language for manipulating arrays. In: VLDB (1997)

25. Maruyama, N., et al.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: SC 2011 (2011)
26. Momjian, B.: PostgreSQL: Introduction and Concepts. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
27. Racah, E., Beckham, C., Maharaj, T., Kahou, S.E., Prabhat, M., Pal, C.: Extremeweather: a large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events. In: NIPS (2017)
28. Racah, E., et al.: Extremeweather: a large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events. In: NIPS (2017)
29. Shi, R., et al.: Designing efficient small message transfer mechanism for inter-node MPI communication on infiniband GPU clusters. In: HiPC (2014)
30. Shi, R., et al.: HAND: a hybrid approach to accelerate non-contiguous data movement using MPI datatypes on GPU clusters. In: ICPP (2014)
31. Soroush, E., Balazinska, M., Wang, D.: ArrayStore: a storage manager for complex parallel array processing. In: SIGMOD 202011. ACM (2011)
32. Sousa, M., Dillig, I., Vytiniotis, D., Dillig, T., Gkantsidis, C.: Consolidation of queries with user-defined functions. SIGPLAN Not. **49**(6), 554–564 (2014)
33. Stonebraker, M., et al.: Requirements for science data bases and SciDB. CIDR **7**, 173–184 (2009)
34. Suzuki, K., Horiba, I., Sugie, N.: Linear-time connected-component labeling based on sequential local operations. Comput. Vis. Image Underst. **89**(1), 1–23 (2003)
35. Tang, H., Byna, S., et al.: In situ storage layout optimization for AMR spatio-temporal read accesses. In: ICPP (2016)
36. Tang, H., et al.: SoMeta: scalable object-centric metadata management for high performance computing. In: CLUSTER 2017, pp. 359–369. IEEE (2017)
37. Tang, H., et al.: Toward scalable and asynchronous object-centric data management for HPC. In: CCGRID 2018, pp. 113–122. IEEE (2018)
38. The HDF Group. HDF5 User Guide (2010)
39. Wang, Y., Nandi, A., Agrawal, G.: SAGA: array storage as a DB with support for structural aggregations. In: SSDBM 2014. ACM, New York (2014)
40. Wehner, M., Prabhat, et al.: Resolution dependence of future tropical cyclone projections of CAM5.1 in the U.S. CLIVAR hurricane working group idealized configurations. JCLI (2015)
41. Widenius, M., Axmark, D.: MySQL Reference Manual. O'Reilly & Associates Inc., Sebastopol (2002)
42. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI 2012 (2012)
43. Zhang, W., et al.: Exploring memory hierarchy to improve scientific data read performance. In: CLUSTER 2015, pp. 66–69. IEEE (2015)
44. Zou, X., et al.: Parallel in situ detection of connected components in adaptive mesh refinement data. In: CCGrid 2015 (2015)