ArrayUDF: User-Defined Scientific Data Analysis on Arrays

Bin Dong[†], Kesheng Wu[†], Surendra Byna[†], Jialin Liu[†], Weijie Zhao[‡], Florin Rusu[†‡]

[†]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720
[‡]University of California, Merced, 5200 Lake Rd, Merced, CA 95343
{DBin,KWu,SByna,Jalnliu}@lbl.gov,{wzhao23,frusu}@ucmerced.edu

ABSTRACT

User-Defined Functions (UDF) allow application programmers to specify analysis operations on data, while leaving the data management tasks to the system. This general approach enables numerous custom analysis functions and is at the heart of the modern Big Data systems. Even though the UDF mechanism can theoretically support arbitrary operations, a wide variety of common operations – such as computing the moving average of a time series, the vorticity of a fluid flow, etc., – are hard to express and slow to execute. Since these operations are traditionally performed on multi-dimensional arrays, we propose to extend the expressiveness of structural locality for supporting UDF operations on arrays. We further propose an *in situ* UDF mechanism, called ArrayUDF, to implement the structural locality. ArrayUDF allows users to define computations on adjacent array cells without the use of join operations and executes the UDF directly on arrays stored in data files without requiring to load their content into a data management system. Additionally, we present a thorough theoretical analysis of the data access cost to exploit the structural locality, which enables ArrayUDF to automatically select the best array partitioning strategy for a given UDF operation. In a series of performance evaluations on large scientific datasets, we have observed that – using the generic UDF interface – ArrayUDF consistently outperforms Spark, SciDB, and RasDaMan.

KEYWORDS

ArrayUDF; User-Defined Data Analysis; Array Structural Locality; SciDB; MapReduce; Spark

1 INTRODUCTION

As technology advancements in large-scale scientific experiments, observations, and simulations are generating unprecedented amounts of data, scientists are in need of novel techniques to process the data. Scientific datasets typically contain multi-dimensional arrays and are stored as files in shared disk-based storage systems [36]. Analysis of these large datasets has to be conducted directly on the raw data files, in an *in situ* manner [1], because loading the data into database systems that typically assume shared-nothing architecture is a very expensive and cumbersome operation. Furthermore, analysis operations on datasets are different from one run to another. Therefore, it is necessary to allow application programmers to customize their analysis operations through the User-Defined Function (UDF) mechanism. To provide such an extensible analysis capability, we propose a novel UDF abstraction for multi-dimensional arrays and present an *in situ* system that executes UDF operations over large datasets dramatically faster than the state-of-the-art data management systems.

UDFs have been explored extensively in data management literature [4, 33] and are widely implemented in database servers [13, 30, 33, 44], recent parallel data processing systems [15, 18, 26, 28], as well as specialized scientific data processing systems [5, 6, 9, 17]. The assumption behind UDFs is that most data analysis operations and their input data have a relatively simple relationship. Typically, the relationship is that a single operation is applied to each element of a dataset, e.g., a tuple of a relational table or a cell of an array. In these cases, a user only needs to specify the operation on a single data element, while the underlying system automatically executes the operation over all the elements. To the user, the operation on a single data element is the application logic, while the task to manage the dataset is a support function. However, this support function is often much more complex, especially for large datasets that tend to be processed in parallel. The UDF mechanism allows users to concentrate on the application logic and leave the data management task to the system, which significantly improves productivity of the users.

A critical limitation of most existing UDF implementations is that they typically allow users to define an operation only on a single element. However, most real-world data analysis tasks, such as computing the moving average of a time series or the vorticity of a flow field [14] (see detailed examples in Section §2), require the values of not just a single element, but also many of its neighbors. This dependency on adjacent elements is referred as *structural locality* in the literature [27]. To alleviate this limitation and to support a flexible UDF mechanism, some DBMS systems allow UDFs on an entire data table [30, 42, 44], while MapReduce systems [18] allow users to define an operation on a set of related elements in the *reduce* stage. However, this flexibility comes at the expense of tedious aggregation that is required to build the input of UDF properly. Moreover, since a data element is required by multiple neighbors in real applications, the data management system often replicates each element multiple times during execution, which

[1]The ability to process data directly in their native file formats has been referred to as "*in situ* processing" in data management literature [3, 8].

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '17, June 26-30, 2017, Washington, DC, USA
© 2017 ACM. 978-1-4503-4699-3/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3078597.3078599

degrades the performance significantly. Furthermore, the reduction operations are performed uniformly on all neighboring elements involved, while real applications typically need to perform different operations on different neighboring elements. In short, there are many real-world operations that are not well-supported by the existing UDF mechanisms, i.e., these operations are hard to express and slow to execute.

In this paper, we present the design and implementation of ArrayUDF, a novel *in situ* UDF abstraction optimized for multi-dimensional arrays. ArrayUDF supports generalized *structural locality* by allowing users to define operations not only on the local value of an array cell, but also on the values of its neighbors. Meanwhile, using ArrayUDF to define an operation for a given cell, users can define a specific operation for each neighbor of this cell in the UDF. ArrayUDF automatically identifies the optimal array chunking strategy that guarantees the efficient execution of each operation. In summary, the contributions of this paper include:

- Introduction of ArrayUDF, the first UDF mechanism for multi-dimensional arrays with a generalized structural locality support. By providing a novel operator to express the relative position of a neighbor, ArrayUDF allows users to define complex analysis operations directly on arrays. Compared with the computing model of MapReduce, ArrayUDF uses a single step to complete the task for both Map and Reduce operations.
- Implementation of ArrayUDF and its processing system using the SDS framework [19], a database-like system for high-performance computing (HPC). ArrayUDF executes in an *in situ* manner [3, 8] for efficiency. ArrayUDF works directly on raw scientific file formats, e.g., HDF5 [39], where the files are stored in parallel file systems.
- Algorithms to dynamically identify the optimal chunking strategy and build a "ghost zone" for a given UDF based on the data access cost. Compared with the chunking strategies for shared-nothing array databases, these unique features of ArrayUDF enable it to work efficiently on dynamically-scheduled resources in an HPC environment as well as on the large scientific datasets stored on shared-storage systems as files.
- An analytical performance model for providing theoretical support to justify the chunking strategies of ArrayUDF and also for tuning ArrayUDF to different array organizations on disk.
- Evaluation of ArrayUDF using both synthetic and real scientific datasets on a Cray XC30 supercomputer and also a commodity Linux server. We have compared ArrayUDF with SciDB [9] and RasDaMan [6] – specialized systems for multi-dimensional array processing – and Spark, the state-of-the-art MapReduce system that supports generic UDFs. Our evaluations show that ArrayUDF is considerably faster than existing alternatives. For instance, using the generic UDF interface, ArrayUDF is up to 2070 times faster than Spark to complete a real-world data analysis.

In Section §2, we present several motivating examples from real scientific applications. In Section §3, we introduce ArrayUDF and its design and implementation. We present performance evaluation of ArrayUDF in Section §4. In Section §5, we discuss related research efforts. We conclude the paper in Section §6.

2 BACKGROUND AND MOTIVATION

Here we introduce several real-word applications that motivate this research. Our work can be the building blocks for advanced data mining algorithms [32] and systems, e.g., TensorFlow[2].

2.1 Motivating examples

Example 1: Moving average based smoothing for time series. A variety of applications produce 1D time series data. Examples include collection of temperature periodically on a flux tower in climate observations and daily stock prices in finance industry. These time series datasets usually contain two parts: a meaningful pattern (e.g, seasonal trend) and a superimposed noise with limited scientific meaning. Moving average based smoothing is widely used to extract the meaningful patterns. Specifically, at a time t, moving average based smoothing has to determine the average of observed values that are close to this particular time, as shown in the following equation:

$$V'_t = \frac{w_{t-k}V_{t-k}+\ldots+w_{t-1}V_{t-1}+w_tV_t+\ldots+w_{t+m}V_{t+m}}{k+m+1}, \quad (1)$$

where V_t is the observed value, w_t the weight and V'_t the smoothed value. k and m are the steps before and after t.

Example 2: Vorticity computation. S3D is a high-fidelity direct numerical simulation (DNS) designed to capture key turbulence-chemistry interactions in a combustion engine [14]. A key variable related to the turbulent motion is vorticity. It defines the local spinning motion around a given location. To simplify the description, we give the z component of the vorticity at a point (i, j):

$$\zeta_{i,j} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \approx \frac{u_{i,j+1}-u_{i,j-1}}{2\triangle x} + \frac{v_{i+1,j}-v_{i-1,j}}{2\triangle y}, \quad (2)$$

where u and v are the flow velocity (i.e., 2D arrays) on the x and y axes, respectively, and $\triangle x$ and $\triangle y$ are the constant differences. Note four neighbors per cell are required in this computation.

Example 3: Peak detection. Mass-spectrometry imaging (MSI) is an essential technology required by biological sciences to understand metabolism [34]. In MSI, the mass-to-charge ratio (m/z) is a variable of interest which is usually a 3D array for a single object (e.g., brain tissue sample). A key data analysis task in MSI is to find peaks of m/z via calculating the gradient of each point, typically through the Laplacian operator [31]. The Laplacian for a given point (i, j, k) is defined as:

$$\begin{aligned} g_{i,j,k} = 6 \times v_{i,j,k} - (&v_{i,j,k+1} + v_{i,j,k-1} + v_{i,j+1,k} \\ &+ v_{i,j-1,k} + v_{i+1,j,k} + v_{i-1,j,k}), \end{aligned} \quad (3)$$

where v is the m/z value and g denotes the gradient value. Note that seven values of v are needed for each value of g.

Example 4: Trilinear interpolation. Plasma physics simulations, such as VPIC, are used to study magnetic reconnection [11]. During a simulation, magnetic field values are computed at mesh points. However, data analysis requires to find the magnetic field at the location of each particle. Generally, the magnetic field at the location of a particle is interpolated from the nearest *eight* mesh points in a 3D mesh. For a particle at (x, y, z), its magnetic value $v_{x,y,z}$ can be computed with the trilinear interpolation equation:

$$\begin{aligned} v_{x,y,z} = v_{i,j,k}N_0 + v_{i,j,k+1}N_1 + v_{i,j+1,k}N_2 + v_{i,j+1,k+1}N_3 + \\ v_{i+1,j,k}N_4 + v_{i+1,j,k+1}N_5 + v_{i+1,j+1,k}N_6 + v_{i+1,j+1,k+1}N_7 \end{aligned} \quad (4)$$

where N_i ($i \in \{0, \ldots, 7\}$) are the distances to each corner. Thus, we can see that the trilinear interpolation for a single particle requires access to eight adjacent magnetic values.

Key observations from the examples above: .

- The computation follows a stencil [7, 29] pattern. In general, a new array B is computed from an existing array A, where the value of B at location (i, j, k) is determined not only by $A(i, j, k)$, but also by its neighbors. This is called *structural locality* in a previous work [27].
- The neighbors of $A(i, j, k)$ do not go through the same operation. Previous work – such as MapReduce [18] and GLADE [15] – generally assume that a uniform reduction or aggregation operation is sufficient.
- There is a variety of analysis operations for different applications. The best option for implementing all of them is to follow the UDF approach to support the common data management operations, while allowing users to define custom operations on data.

2.2 Research challenges

SciDB, RasDaMan, and AML [27] implement the "window" operator, which supports some form of structural locality. However, the operations on the values within a window are generally a reduction operation, while the above examples show a variety of operations. Furthermore, the "window" typically has to be a rectangular subdomain, e.g., a 2×2 square, while the above examples contain more complex definitions of neighborhood. Additional flexibility is needed in specifying both the operations and the neighborhood. While UDFs provide a general framework to express custom analysis operations, they have to address the following challenges in order to be applicable to structural operations defined over arrays:

- How to define neighborhood cells involved in a UDF operation? These definitions have to be compact, easy to construct, and – at the same time – efficient to evaluate.
- How to develop UDFs that support *in situ* array data analysis? As most scientific datasets are stored in file formats such as HDF5 and netCDF [36], a new UDF mechanism has to work directly on these files without loading the data into a separate DBMS.
- How to effectively partition the data and computation on parallel computing systems? As each array cell is needed at different computation steps, the data partition may require overlapping. Maintaining load-balance and reducing overlap is critical to achieve good overall performance.

3 ArrayUDF APPROACH

To address the identified challenges, we propose a UDF system named ArrayUDF. With ArrayUDF, we extend the expressiveness of structural locality to allow users to easily define operations on adjacent cells of an array and to perform data management tasks efficiently for supporting these user-defined operations. ArrayUDF is also capable of identifying the minimum portion of an array accessed on each process (e.g., CPU core) and operate on that portion of the array data stored in files, without loading the entire array into the system. This optimization is possible because the array syntax used by ArrayUDF to describe the operations provide a clear mechanism to identify the relevant cells and the optimal data partition can be determined analytically.

ArrayUDF computational model. For comparison, we first introduce the computational models of relational databases and MapReduce systems. We use f to denote a user-defined function.

For relational tables T and T', the generic UDF model is:

$$t' \leftarrow f(t) \tag{5}$$

where f is applied to each tuple $t \in T$ and t' represents the tuple in the output table T'. There is a one-to-one mapping between input and output tuples. Aggregate UDFs allow for more input tuples to determine an output tuple. However, it is done by grouping on the values of some attributes, i.e., the SQL GROUP-BY operator.

In the MapReduce paradigm, the input of a UDF is a $(key, value)$ pair, or (k, v) for short. MapReduce has two components – *Map* and *Reduce* – which can be formally expressed as:

$$\begin{aligned} Map &: \Big\{(k_i, v_i) \mid i \in [1, m]\Big\} \leftarrow f_1\Big((k, v)\Big) \\ Reduce &: \Big\{(k_i', v_i') \mid i \in [1, p]\Big\} \leftarrow f_2\Big(\Big\{(k', v_i) \mid i \in [1, n]\Big\}\Big) \end{aligned} \tag{6}$$

where $m, n, p \in \mathbb{N}$, $\mathbb{N}$ is the natural number set, f_1 is an enhanced UDF that implements a one-to-many mapping from the input pair to an intermediate set of key-value pairs and f_2 is a SQL GROUP-BY AGGREGATE identical to the relational aggregate UDF. Key k' is the grouping parameter in f_2. Value v' is generated by f_2 through uniformly applying a single operator – such as *SUM* – to each v_i.

The computational model of the ArrayUDF is defined on two d-dimensional arrays, A and A' ($d \in \mathbb{N}$). The cell c' at coordinate $(i_1, i_2, \ldots, i_d)$ in A' is computed by a stencil S of the cell c at the same coordinate in A. Theoretically, the stencil S is a set of array cells which have structural locality. Specifically, for the cell c at coordinate $(i_1, i_2, \ldots, i_d)$, $S = \{c_{i_1+\delta_1, i_2+\delta_2, \ldots, i_d+\delta_d} | \forall j \in [1, d], \delta_j \in [L_j, R_j]\}$, where $L_j \in [-i_j, 0]$, $R_j \in [0, N_j - i_j]$, and the N_j is the size of the jth dimension. Obviously, the $\delta_1, \delta_2, \ldots, \delta_d$ are relative distances from the cell c. For simplicity, each cell in S is expressed as $s_{\delta_1, \delta_2, \ldots, \delta_d}$, which is $c_{i_1+\delta_1, i_2+\delta_2, \ldots, i_d+\delta_d}$. With these notations, the formal computational model of ArrayUDF is:

$$c'_{i_1, \ldots, i_d} \leftarrow f\left(\left\{s_{\delta_1, \ldots, \delta_d} \mid \forall j \in [1, d], \delta_j \in [L_j, R_j]\right\}\right). \tag{7}$$

Distinct from the UDF models of relational databases and of MapReduce, in the model of ArrayUDF, the function f has a stencil S as input. A stencil S allows ArrayUDF to express any neighborhood shape implicitly via relative distance. This is also different from AML [27], where a shape parameter is required to express neighbor cells. Meanwhile, users can specify different operators on different cells of S within the UDF. This distinguishes ArrayUDF from most existing aggregate UDFs [15], where a single aggregate operator is applied onto all values. In the relational model, this functionality requires a chain of self-joins having cardinality equal to the number of cells in the stencil. In MapReduce, the self-joins are substituted by the one-to-many replications in the *Map* stage. In general, when the size of S is equal to one, ArrayUDF is identical to the relational and *Map* UDFs. Otherwise, ArrayUDF is similar to relational aggregates and the *Reduce* function. In summary, ArrayUDF eliminates the shape operators for computing a stencil (S) and allows a more concise definition of UDFs for arrays.

System overview. Towards implementing the computational model of ArrayUDF, we introduce its key software components:

- **Array** is a data structure that encapsulates the multi-dimensional array stored in files. This is the primary object a user interacts with. **Array** implements the function *Apply* to execute the UDF defined by the user. On a parallel computing system, each process creates its own instance of the **Array** object with the same arguments, and invokes the same UDF with the function *Apply*. Moreover, **Array** has functions to partition the multi-dimensional array automatically, to build the necessary overlapping regions (known as ghost zones), and to divide the computation among the processes. This partitioning method is guided by a theoretical analysis to be discussed in § 3.3. More details on **Array** are reported in Section § 3.2.
- The data structure **Stencil** represents an array cell and its neighborhood cells in relative coordinates. As defined earlier, **Stencil** is a relative coordinate-based notation that allows users to describe the operations to be performed on each neighbor separately. Previously used notations for "shape" and "window" demand all neighbors to be described in a collective form, which limits aggregation operations used in scientific data analysis. In contrast, our relative coordinate-based notation is more flexible. Users define their UDFs with these **Stencil**s. In C++ syntax, this relative coordinate is expressed using the parenthesis operator of the **Stencil** object. More details about **Stencil** are in § 3.1.

ArrayUDF is currently implemented as part of the Scientific Data Services (SDS) framework [19], which provides the basic I/O drivers for reading and writing the data from parallel file systems. We implement **Array** and **Stencil** as C++ classes. We show an example of using ArrayUDF in Fig. 1, where "MyAvg" is a UDF to compute the average value using four adjacent cells. "MyAvg" is executed by calling the *Apply* function of an **Array** instance within the main function. The template feature is used to support different data types, e.g., float and double.

```
//A UDF on a stencil containing four neighborhood cells
float MyAvg(Stencil<float> &s){
 return (s(1,0)+s(-1,0)+s(0,-1)+s(0,1))/4;
}
int main(){
 vector<int> cs(2)={2,2};//Chunk size
 vector<int> gs(2)={1,1};//Ghost zone size
 Array<float> A("d2d.h5",cs,gs);
 Array<float> B("d2davg.h5");
 //Run UDF code using Apply function
 //Store the result in B
 A->Apply(MyAvg, B);
  ...  //Other operations on B or A
}
```

Figure 1: An example using ArrayUDF to compute user-defined average, i.e., "MyAvg", on a 2D array A, stored in a HDF5 file, named "d2d.h5". The results are stored in array B, which has the same dimensions as that of A and is stored in file "d2davg.h5". Given different UDF operators, B can have different dimensions from A. The user-specified chunk size (cs) and ghost zone size (os) are used to support parallel processing. Both cs and os are optional and can be determined by ArrayUDF automatically.

3.1 Stencil design considerations

The **Stencil** data structure represents an array cell and its neighbors needed for a single invocation of the UDF. **Stencil** plays a role similar to a tuple in a database or a key-value pair concept, working as the input to the UDF. It is more flexible than existing concepts such as "window" and "shape" in two ways. A **Stencil** can be used to define more complex neighborhoods than "window" and "shape" and it allows the user to specify a different operation on each of the neighboring cells. This flexibility allows a much wider variety of analysis operations to be defined.

Since modern CPUs can carry out many arithmetic operations quickly, we anticipate that the complexity in arithmetic operations is less dominant in the overall performance of a UDF than the cost of data accesses. As with "window" and "shape," we expect operations defined on our relatively compact stencils to access a small number of neighbors and, therefore, can be carried out efficiently, while operations involving a large number of neighbors, no matter in the form of a "window", a "shape", or a "stencil", require more data accesses and take a longer time to complete. In short, we expect the flexibility to address individual neighboring cells in a UDF not to impose a significant cost on its own.

To understand the relative coordinates used in the definition of a **Stencil**, it is useful to visualize the coordinates of a multi-dimensional array to be a set of mesh points in space and each array to be an attribute of a mesh point. For example, one set of attributes might be their location in space, (e.g., x, y, and z dimensions), and another set might be temperature, pressure, and concentrations of some chemical species. The analysis examples given in earlier sections follow this basic schema and a typical analysis function computes $B(i, j, \ldots)$ from $A(i, j, \ldots)$ and its neighbors. Furthermore, the neighbors are of a fixed combination of offsets around $A(i, j, \ldots)$. In applied math, the pattern formed by these offsets (positions relative to a center $(i, j, \ldots)$) is known as a "stencil". ArrayUDF uses a syntax implemented as the parenthesis operator of C++ to allow each cell in the stencil to be explicitly named. For example, in a 2D case, $S(0, 0)$ refers to the "center" of the stencil $A(i, j)$, and $S(1, 0)$ refers to the neighbor $A(i + 1, j)$.

The use of relative coordinates in the UDF allows the users to define the operations without mentioning the coordinates $(i, j, \ldots)$. When the function is actually evaluated, we need to convert these relative coordinates back to the absolute coordinates in order to access the specific cells. In this context, the use of the relative coordinates also allows each execution thread to work on its own portion of the global mesh as illustrated in § 3.2 and Fig. 6.

3.2 Array design considerations

The **Array** class is a high-level abstraction and representation for multi-dimensional arrays that contains a group of functions for run-time tasks. Users employ **Array** to represent an array stored either in memory as std::array or on disk as an HDF5 dataset. For this work, the key function of **Array** is *Apply*, which executes a UDF. Behind this function, we are able to implement a number of data management techniques for parallelizing the execution of the UDF automatically through partitioning the global mesh into suitable sub-domains and overlapping the sub-domains through ghost zones.

3.2.1 Parallel processing with dynamic chunking. A typical approach for parallel processing of a large problem is to divide the problem into smaller *partitions* or *chunks*. Following this general practice of data management systems, we also divide the evaluation of the UDF by partitioning data. We observe that an invocation of UDF requires access to values near $(i, j, ...)$. Therefore, it is essential for us to keep the neighbors close to each other as much as possible. This basic requirement is similar to many parallel computing applications [25], which allows us to borrow a number of techniques for designing an efficient data partitioning algorithm. Our overall approach can be viewed as partitioning (chunking) the mesh defined by the multi-dimensional coordinates into chunks, and then map each chunk to a parallel processing element (PE). We assume that each chunk is small enough to fit into the available memory on a PE. A processing element is responsible for producing the output array belonging to the chunk. To complete this task, it not only needs to access the corresponding chunk of the input data array, but also some extra portion of the array, which we call *ghost zone* (as illustrated in Fig. 2). Since ArrayUDF is designed to process data directly on raw data files, there is no data pre-processing step (e.g., loading data into the system) for it to figure out an efficient chunking strategy in advance. We need to perform all the chunking related decisions (e.g., chunk size) dynamically. Moreover, for a given chunk, it is necessary to consider its logical view (i.e., shape) and its physical view (i.e., data layout on storage) as they affect the performance of reading the data.

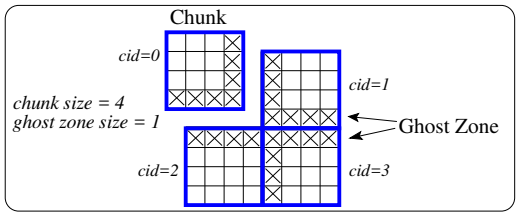


Figure 2: Example of chunks and ghost zones in a 2D array.

Chunk management overview. In order to maintain load balance among processing elements, ArrayUDF attempts to keep the size and the shape of chunks similar. An example of chunking in ArrayUDF is shown in Fig. 2, where four chunks are created. Each chunk has a dynamically assigned ID (i.e., *cid* in the example). The ID is calculated using the row-major ordering from the coordinates of chunks, which allows ArrayUDF to identify a chunk quickly.

FUNCTION ***ArrayUDFChunk***$(M, P, (N_0, \ldots, N_d))$
M: the available memory size per process;
P: the total number of processes;
$(N_0, \ldots, N_d)$: the size of an array for d dimension
0. S = **min** $(M, (\prod_{i=1}^{d} N_i)/P)$
1. **if** general chunking **then**
2. Find an integer w that minimizes $\mid S - w^d \mid$
3. $c_1 = w, c_2 = w, \ldots, c_d = w$
4. **if** layout-aware chunking **then**
5. **for** $i \in (d, d-1, \ldots, 2)$ **do**
6. $c_i = S\%N_i; S = S/N_i$
7. $c_1 = S$
8. return $(c_1, c_2, \ldots, c_d)$

Figure 3: The method for selecting chunking parameters.

Optimal chunking strategy selection. A key challenge to support dynamic chunking is to decide the optimal chunking parameters, including chunk size, chunk shape, and chunk layout. The chunk size is the number of cells within a chunk, the shape is its logical view, and the layout is its physical data layout on disk. To address this challenge, ArrayUDF provides two chunking strategies: **a general chunking** and **a layout-aware chunking**, as shown in the pseudocode in Fig. 3. We describe the theoretical reasoning for these strategies in § 3.3. In the following, we describe the high-level idea and concrete applications.

- To find **an optimal and general chunking**, we assume that each array cell is accessed separately. Such an assumption guarantees that the chunk layout has no impact on the performance of accessing a chunk. In other words, we consider it as an average case for different chunk layouts. Users can choose this chunking strategy when the layout of the array on disk is unknown. As shown in line 0 in Fig. 3, the chunk size in this strategy is set to be as large as possible to fit in the memory of each processing element to reduce the startup overhead of I/O as well as to assign at least one chunk to each process to maximize parallelism. In terms of the chunk shape, ArrayUDF chooses a square shape (lines 2 and 3 in Fig. 3) to minimize the number of ghost cells for each chunk. A formal analysis of this strategy is given in § 3.3 and is illustrated in Fig. 4, where the square chunk reads 12 ghost cells but the non-square one reads 20 ghost cells.
- To identify **an optimal and layout-aware chunking**, we take the row-major layout as an example because it is popular in most array data formats. A storage system typically organizes the elements of a multi-dimensional array in a linear order based on their coordinates. With the row-major layouts, the 1^{st} dimension is the slowest varying dimension and the last dimension is the fastest varying dimension. In this case, ArrayUDF chooses the chunk whose layout on disk is as contiguous as possible by maximizing the fast-varying (or higher) dimensions for a chunk. In other words, ArrayUDF chooses the chunk size based on the linearized organization of an array in row-major order (lines 5 to 7 in Fig. 3). We also consider the memory limit and the parallelism (line 0 in Fig. 3). The impact of reading extra ghost cells is not considered here because the cells from the chunk and the ghost zones often form a single or few contiguous reads as illustrated in Fig. 5. More detailed analysis is given in § 3.3.

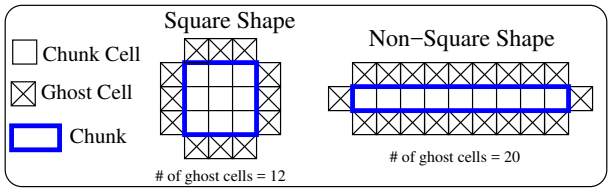


Figure 4: An example showing the number of ghost cells for different chunk shapes.

3.2.2 Dynamic ghost zone building using a trial run. In most use cases, the programmer who develops the UDF can determine the thickness of the ghost zone and therefore can provide the information to the ARRAY object to help with dynamic data chunking. However, it is possible that the size of ghost zones are unknown

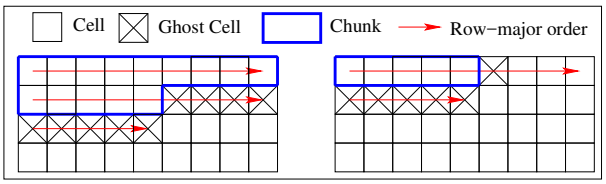


Figure 5: An example showing the layout-aware chunking for a 4×9 array. On the left, a chunk occupies more than one row. On the right, a chunk occupies a partial row.

a priori, for example when the UDF source code is not accessible to the user. In such cases, ArrayUDF provides a mechanism based on a "trial run" to determine the ghost zone size. During the trial run, ArrayUDF gathers all the relative coordinates used in the UDF. We assume the relative coordinates do not depend on the values of the input array, and, therefore, only perform the trial run on a small number of cells. At this time, we only perform this trial run on the first data point on each PE. After the trial run, ArrayUDF chooses the maximum absolute value among the gathered relative coordinates for a dimension as the number of ghost cells of this dimension. Therefore, different dimensions can have a different number of ghost cells. For a certain dimension, the current ArrayUDF implementation requires that both directions have the same number of ghost cells. This reduces the number of input parameters needed by ArrayUDF, but it also causes reading unnecessary ghost cells. However, as observed in most example applications, they require the same number of cells in both directions of a given dimension. We can easily extend ArrayUDF to set the sizes of a ghost zone for different directions of a dimension. Sometimes, it is impossible to determine the ghost zone size using the trial run. In this situation, ArrayUDF uses the default ghost zone size. Moreover, users can specify a default ghost zone size when initializing **Array** object, as shown in Fig. 1.

*3.2.3 Putting it all together: the **Apply** algorithm.* The **Apply** function of **Array** is the entry point to execute a UDF and it also contains a skeleton of our runtime system. The pseudocode of **Apply** is outlined in Fig. 6. Overall, the pseudocode shows that the UDF P_{udf} processes array A and stores the results in array B. As shown in Fig. 1, users can provide their own chunk size (cs) and ghost zone size (gs), both of which are therefore considered as the input. To support parallel processing, each process (PE) initializes an **Array** instance by itself and follows the same algorithm without any communication. During the initialization, each process obtains the total number of processes P and the rank R of itself among all processes using certain library calls such as MPI. In the following, we describe how **Apply** works on a single process.

From line 0 to line 1, the **Apply** algorithm determines the chunk size and ghost zone size, if a user does not provide them. The methods for determining these two parameters are discussed in § 3.2.1 and § 3.2.2. Then, the algorithm starts to read a chunk into memory for processing (lines 2 and 3). The ID of the first chunk to be read by a process is equal to the rank of the process, which permits each process to have different chunks to process concurrently. Reading the chunk is performed by the *LoadNextChunk* function, which first converts the chunk id to the coordinates of the top-left corner and bottom-right corner of a chunk. Then, it extends the coordinates of the chunk to include the ghost zone. Finally, the *LoadNextChunk* function reads the chunk into memory if the coordinates are valid.

Once the **Apply** algorithm receives the chunk data from *LoadNextChunk*, it starts to apply the UDF (i.e., P_{udf}) on each cell of the chunk (lines 4 – 6). Specifically, a new instance of **Stencil** is initialized. Then, P_{udf} is called with the new **Stencil** instance. The returned result of the UDF is stored in B. After a PE processes a chunk, it reads the next chunk to process. The ID of the next chunk is the current id plus the total number of processes (line 7). The data stored in B can be flushed to persistent storage, depending on the available memory space.

3.3 Analytical model for ArrayUDF

In this section, we present an analytical model to characterize the performance of ArrayUDF, with the primary goal of deriving a chunking strategy. We first build a generic model without considering array layout and then adapt the model for specific array layouts. The notations used in the model are shown in Table 1.

Table 1: Notations used in the chunking strategy model

c_i	Chunk size in the i^{th} dimension, $i \in (1, \ldots, d)$
c	Number of cells in a chunk, $c \equiv \prod_{i=1}^{d} c_i$
C	Number of chunks $C \equiv \prod_{i=1}^{d} \lceil N_i / c_i \rceil$
δ	Size of a ghost zone
e	Number of elements in a chunk plus its ghost zones
G	Number of elements in all ghost zones
d	Rank of an array (number of dimensions)
M	Memory size of a single process
N_i	Size of the i^{th} dimension, $i \in (1, \ldots, d)$
N	Number of array elements, $N \equiv \prod_{i=1}^{d} N_i$
P	Number of parallel processes

Execution of a computation function typically contain three overheads: computation time, communication time, and I/O time. In the following analysis, we will ignore the computation and communication costs. By design, each process in ArrayUDF can evaluate the UDF in parallel without inter-process communication. We will, therefore, assume that there is no communication overhead. The computation time for each element of the output array can be reasonably represented as a constant, independent of the logical array layout and the physical chunk layout; therefore, as long as there are enough array cells to be divided evenly among processes, the computation cost can be divided evenly and will not depend on how we partition the chunks. Similarly, selection of the layout of the chunks does not affect the computation time. To further simplify the analysis, we also assume the ghost zones are of the same width in each direction and every dimension.

Overall, ArrayUDF divides the evaluation into chunks whose shape and size is determined analytically. On each process, the evaluation proceeds independently. From Fig. 6, we see that the key I/O cost is to read a chunk of the input array and the ghost zones. Let c and e denote the number of elements in a chunk and the extended region including the chunk and its surrounding ghost zones, as shown in Table 1. Given the size of a chunk ($c_1 \times c_2 \times \ldots \times c_d$), the total volume of data in the chunks is $c = \prod_{i=1}^{d} c_i$. To evaluate the size of e, we need to know how many ghost zones are present. For simplicity, if a part of the ghost zone is available, we will count it as

FUNCTION ***A→Apply***(P_{udf}, B)

P_{udf} : pointer to user-defined function. A : an array to Apply P_{udf}. B : result array.

$A{\rightarrow}cs$: chunk size. $A{\rightarrow}gs$: ghost zone size. $A{\rightarrow}P$: the total number of processes. $A{\rightarrow}R$: the rank of current process.

0. **if**($A{\rightarrow}cs$ == NULL) **then** Determine the chunk size $A{\rightarrow}cs$, as discussed in § 3.2.1 //No user-defined chunk size
1. **if**($A{\rightarrow}gs$ == NULL) **then** Issue a trial-run to get $A{\rightarrow}gs$, as discussed in § 3.2.2 //No user-defined ghost zone size
2. cid = $A{\rightarrow}R$; // the ID for the first chunk is equal to the rank of current process
3. **while** (($cbuf$=***LoadNextChunk***(cid, $A{\rightarrow}gs$, $A{\rightarrow}cs$)) != NULL) **do**
4. **for each cell** c **within the chunk** $cbuf$ **do**
5. $cell_{udf}$ = **Stencil**(c, $A{\rightarrow}gs$, $A{\rightarrow}cs$, $cbuf$) //Initialize a **Stencil** to represent the real cell c
6. $B = P_{udf}\ (cell_{udf})$ //Run the UDF function on the **Stencil**
7. $cid = cid + A{\rightarrow}P$ //Next chunk in round-robin manner
8. flush B to disk if necessary

FUNCTION ***LoadNextChunk***(cid, $A{\rightarrow}gs$, $A{\rightarrow}cs$)

0. Obtain the top-left corner coordinate c_{tl} and the below-right corner coordinate c_{br} from cid //See section § 3.2.1
1. $c_{tl} = c_{tl} - A{\rightarrow}gs$; $c_{br} = c_{br} + A{\rightarrow}gs$ //expand chunk with ghost cells
2. **if** checking array boundary fails **then**
3. return NULL
4. Read all cells from c_{tl} to c_{br} within A into buffer $cbuf$
5. **read** $cbuf$

Figure 6: *Apply* algorithm in ArrayUDF. We use "→" symbol to denote the components of A. For example, $A \rightarrow$ *Apply* is the *Apply* method of A and $A{\rightarrow}cs$ is the chunk size metadata of A.

present. As shown in Figure 4, it is possible for a chunk to have two ghost zones along each dimension i. However, those ghost zones might not be present if the expected ghost zones are outside of the extent of the array[2], as shown in Fig. 2. For dimension i with chunk size c_i, the N_i points are divided into $\lceil N_i/c_i \rceil$ chunks, which creates need for $2(\lceil N_i/c_i \rceil - 1)$ ghost zones along this dimension. Given the surface area perpendicular to this dimension to be $\prod_{j\neq i} N_j$, the total volume of ghost zones along dimension i is $2\delta(\lceil N_i/c_i \rceil - 1) \prod_{j\neq i} N_j$, and the total volume of data in ghost zones over all dimensions is:

$$G = 2\delta \sum_{i=1}^{d} \left((\lceil N_i/c_i \rceil - 1) \prod_{j\neq i} N_j \right) \tag{8}$$

The average number of elements for processing a chunk is:

$$e = c + G/C \tag{9}$$

3.3.1 Generic performance model. In this performance model, we assume the layout of the array dimensions is unknown or the time to read an arbitrarily shaped subarray is strictly a linear function of the number of elements in the subarray. Thus, the time to read a chunk plus its surrounding ghost zones is given by:

$$t_{io} = \alpha_0 e + \beta_0 \tag{10}$$

For C chunks and P processes, each process has at most $\lceil C/P \rceil$ chunks and the maximum read time is:

$$T_{io}^{ge} = (\alpha_0 e + \beta_0) \lceil C/P \rceil \tag{11}$$

The selection of the optimal chunk size can be formulated as an optimization problem:

$$\min_{c_1, c_2, \ldots, c_d} T_{io}^{ge}(c_1, c_2, \ldots, c_d) \quad \text{s.t.}$$
$$(1)\ c_i \le N_i, 1 \le i \le d; \quad (2)\ \prod_{i=1}^{d} c_i \le M; \quad (3)\ C \ge P \tag{12}$$

Constraint (2) guarantees that each chunk fits within the available memory while constraint (3) enforces that each process has at least one chunk to work on. Given a large array – the case we are interested in – constraint (3) is easily satisfied and constraint (2) can be turned into:

$$\prod_{i=1}^{d} c_i = M \tag{13}$$

To generate an analytical solution for the above optimization problem, we further assume the ceiling operator can be removed from all the above expressions, which leads to:

$$\begin{aligned} T_{io}^{ge} &\sim \left(\alpha_0 \left[\prod_{i=1}^{d} c_i + \frac{2\delta \sum_{i=1}^{d} \left((N_i/c_i - 1) \prod_{j\neq i} N_j \right)}{\prod_{i=1}^{d} N_i/c_i} \right] + \beta_0 \right) \frac{\prod_{i=1}^{d} N_i/c_i}{P} \\ &= \frac{\alpha_0 N}{P} + \frac{2\alpha_0 \delta N}{P} \sum_{i=1}^{d} \left(\frac{1}{c_i} - \frac{1}{N_i} \right) + \frac{\beta_0 N}{PM} \end{aligned} \tag{14}$$

In the above expression for T_{io}^{ge}, the only term that is affected by the choices of c_i is the expression $\sum 1/c_i$ (multiplied by a positive constant). Given the constraint in Eq. 13, the minimal value of T_{io}^{ge} is obtained with $c_1 = c_2 = \ldots = \sqrt[d]{M}$, which minimizes the total number of ghost cells G. In short, when there are no preferred dimensions, the chunks should be as large as they can fit in the available memory and have each side of the same size.

3.3.2 Layout-aware performance model. When the layout of an array in a file is known, typically there are preferred dimensions for partitioning the array into chunks. The key reason for this is that the sequential read operations are significantly more efficient than random reads. In many cases, a multi-dimensional array is organized in row-major ordering. Hence, there are significant advantages to partition along the slowest varying dimension:

- Under the row-major ordering, when partitions are based on the slowest varying dimension, each partition can be read with a single sequential scan operation.
- When a dimension is fully contained in a chunk ($c_i = N_i$), there is no ghost zone for the dimension, which reduces the number of read operations needed for ghost cells.

[2]It is possible to have a periodic boundary condition, which has a different requirement on ghost zones.

- When the partition is only along the slowest varying dimension, the ghost cells follow the chunk in the file, which allows the ghost cells to be read with the chunk in a single sequential scan.

Therefore, partitioning along the first dimension (i.e., the slowest varying dimension) is highly desirable as long as the array can be evenly divided onto the P processes. In this case, Eq. 8 turns into:

$$G = 2\delta(\lceil N_1/c_1 \rceil - 1)\prod_{j=2}^{d} N_j \sim 2\delta N(1/c_1 - 1/N_1)$$

This leads to the following expression for the read time:

$$T_{io}^1 \sim \frac{\alpha_1 N}{P} + \frac{2\alpha_1 \delta N}{P}\left(\frac{1}{c_1} - \frac{1}{N_1}\right) + \frac{\beta_1 N}{PM} \tag{15}$$

As c_1 approaches 1, the value of G approaches $2N$, which means each process reads more ghost cells than in the generic case considered above. However, since the read operations in this case are large sequential scans, the value of α_1 is considerably smaller than α_0 in the generic case. Thus, dividing chunks along the slowest varying dimension still reduces the overall execution time.

In practice, we find that dividing the array according to the linearization order gives the same advantage as sequential scans, while maintaining better load balance among the processes. This chunking approach may produce ghost zones of irregular shape as illustrated in Fig. 5, however, the ghost zones can still be read with the array elements in the chunk in a single sequential scan operation. Lines 5 and 6 in Fig. 3 implement this approach.

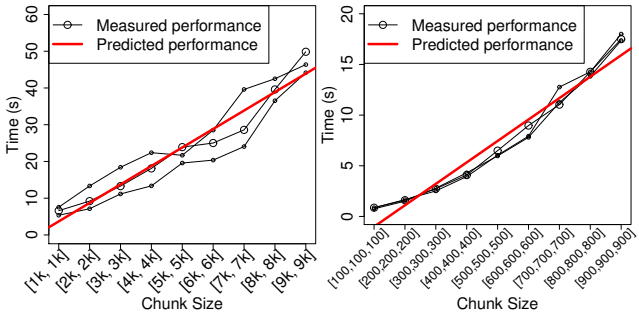
4 PERFORMANCE EVALUATION

We have evaluated ArrayUDF extensively to demonstrate its effectiveness. We explored the design considerations of ArrayUDF and the assumptions used by its performance model with synthetic datasets. We have also performed several tests to compare ArrayUDF with RasDaMan [6], SciDB [9], and EXTASCID [16, 17], where we used the latest versions of these systems available online. Finally, we compared ArrayUDF with Spark – the system with the state-of-the-art generic UDFs – using four real-world scientific datasets and analysis operations.

Experimental setup. We ran our tests on Edison, a Cray XC30 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Edison is equipped with 5576 computing nodes. Each computing node has two 12-core 2.4 GHz Intel "Ivy Bridge" processors and 64 GB DDR3 memory. All tested datasets are stored as HDF5 files in a Lustre parallel file system. The ArrayUDF implementation is compiled with the Intel C/C++ Compiler version 16.0. The Spark installation used as the main comparison is state of the art on HPC [12]. Since Spark does not have native support for HDF5 files, we have use H5Spark [26] to read HDF5 data directly into its RDDs, and therefore reduce the potential impact of the file system and have a fair comparison between ArrayUDF and Spark on the same storage model.

4.1 Synthetic data and Poisson equation solver

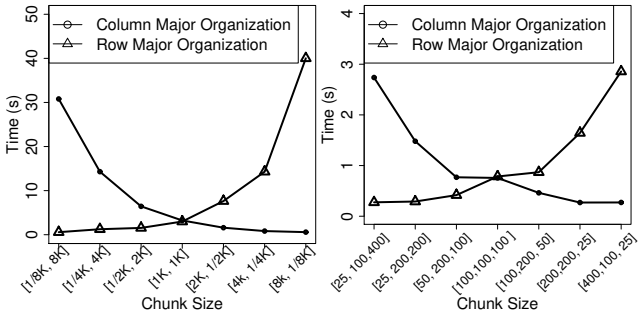
To explore the impact of chunking parameters, to verify the analytical model, and to compare with the performance of SciDB, RasDaMan, EXTACID, and Spark, we have used synthetic two datasets. We have created these datasets that contain two multi-dimensional arrays, $S1$ and $S2$, containing floating point data. $S1$ is a 2D array with ranges (100000, 100000) for (x, y) dimensions, giving 38 GB in file size. $S2$ is a 3D array with ranges (10000, 1000, 1000) for x, y, and z, respectively, resulting in 38 GB in file size. We have used the Poisson equation solver, which is widely used in financial mathematics, Riemannian geometry, and, thus, topology. Using the stencil operator of ArrayUDF, we can express the 2D Poisson equation solver as $4S(0,0) - S(-1,0) - S(0,1) - S(1,0) - S(-1,0)$. Similarly, the 3D Poisson equation solver can be expressed as $6S(0,0,0) - S(-1,0,0) - S(0,1,0) - S(1,0,0) - S(-1,0,0) - S(0,0,-1) - S(0,0,1)$. For the tests describing SciDB in this subsection, we used at most 11 compute nodes. These 11-node tests were dedicated to match the SciDB installation at NERSC supercomputing center, where 10 nodes are used for the data instances of SciDB and 1 node for the metadata instance of SciDB.



(a) S1 (b) S2

Figure 7: Linear relationship between the chunk size and the cost of reading a chunk.

Linear relationship between the chunk size and the time of reading it. In our performance model, we assume that the cost of reading a chunk is proportional to its size. To evaluate this assumption, we ran tests on $S1$ and $S2$ multiple times for a certain configuration and then to build a theoretical model. We show the I/O cost with different chunk sizes for $S1$ and $S2$ in Fig. 7. It is obvious that as the chunk size increases, the time to read the chunk increases linearly as well. The residual standard deviations of this fitting are 3.53 and 1.30 for $S1$ and $S2$, respectively. Thus, we can conclude that the linear relationship exists between the size of a chunk and its read time.

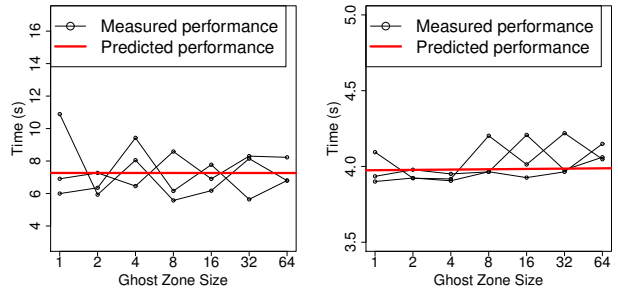


(a) S1 (b) S2

Figure 8: Cost of reading ghost cells for different chunk shapes and for different layouts.

Impact of chunk shape on the cost of reading ghost cells. Using the general performance model (i.e., Eq. 12), we predict that

the chunk shape has a significant impact on data read performance. Specifically, for a fixed size chunk, the square shape guarantees that the number of ghost cells is minimum and, therefore, it has minimum I/O cost. Since we develop the general performance model without relying on any specific data organization, it can characterize the average performance of different organizations. To justify this result, we consider two data organizations, including row-major and column-major, in this test. We compare the performance of reading a chunk with the same shape from these two organizations. As the HDF5 format uses row-major organization to store data, we use the transpose-based data reorganization service of SDS framework [19, 38] to turn the row-major organization into the column-major one. We report the results in Fig. 8. As the chunk shape changes from the left (row-major) to the right (column-major), the time for reading the chunk from column-major organization decreases but the time for row-major organization increases. The square shaped chunk in the middle has the smallest overhead when we consider both organizations together. In other worlds, without considering the organization, the square-shaped chunk has minimum overhead. Taking only the row-major data organization as an example, we can observe that the squared chunk is not always the optimal shape, although it needs minimum amount of ghost cells. Actually, at this time, the chunk layout on disk is the dominant factor for the I/O performance. Our analysis in Section 3.3.2 takes this into account. In summary, these test results confirm our model formed based on a theoretical analysis.



(a) S1 (b) S2

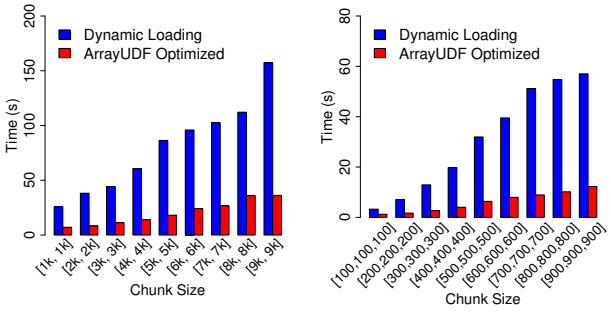
Figure 9: Performance of reading ghost zone in row-major layout.

Overhead of reading ghost cells with layout-aware chunking. In the layout-aware performance model, we assume that the overhead of reading ghost zone can be ignored. To justify this assumption, we have designed a test to measure the time for reading ghost zone sizes, from 1 to 64. In Fig. 9, we show the performance of reading ghost cells from $S1$ and $S2$ datasets. From the figure, we conclude that by increasing the ghost zone from 1 to 64, the time spent to read a single chunk remains flat. Linear regression of multiple measured times show a flat line to represent constant time. With the layout-aware chunking, the ghost cells tend to be contiguously organized with other cells on disk, resulting in the same read performance. Therefore, reading a small amount of ghost zone cells has negligible impact on the performance of reading a chunk. We conclude that during the layout-aware chunking, the assumption we used in the model is reasonable.

Overhead of a "trial run" to detect the size of the ghost zone. ArrayUDF uses a "trial-run" approach to decide the size of the ghost zone for a given UDF. In this test, we have measured the overhead of a "trial run" with respective to different numbers of array cells used by UDF. These numbers range from 4 to 256. When the number of the tested cells is larger than the number of cells required by the Poisson equation solver, we append random array cells at the end. In Table 2, we show the overhead of a "trial run". Overall, we observe that the overhead of trial run is less that *one* microsecond. Compared with the other components of ArrayUDF presented above, this overhead is negligible.

Table 2: Overhead of the trial run (microsecond).

	The number of cells used by UDF						
Data sets	4	8	16	32	64	128	256
S1	0.37	0.38	0.46	0.48	0.54	0.59	0.80
S2	0.48	0.52	0.65	0.75	0.79	0.84	1.04

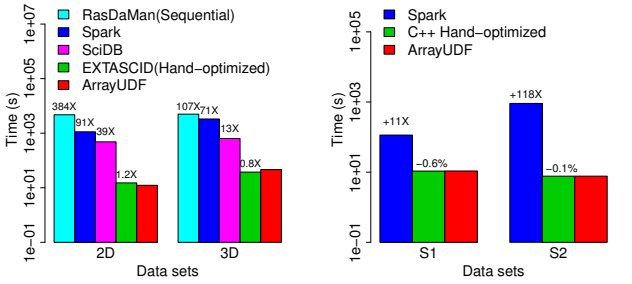


(a) S1 (b) S2

Figure 10: Comparing two methods to handle ghost zone.

Comparing different methods to handle ghost zone. Array data management systems, such as ArrayStore [37], used a dynamic loading method to access ghost cells. In this method, when a chunk needs to access ghost cells from its neighborhood chunk, this method reads the neighborhood chunk from disk into memory. Based on the fact that a UDF has a well-determined access pattern for ghost cells, ArrayUDF optimizes the accessing of ghost cells via statically extending each chunk by a ghost zone when the chunk is first read from disk. As a result, separate disk accesses are not needed for ghost cells in ArrayUDF. We use the low-overhead "trial run" approach to obtain accurate size estimates for a ghost zone. In this test, we compare these two methods by implementing the dynamic chunk loading in ArrayUDF. As shown in Fig. 10, in reading ghost zones from $S1$ and $S2$ datasets, the optimized method in ArrayUDF is on average *four* times faster than the dynamic chunk loading method. Thus, ArrayUDF has an efficient way in handling access to ghost zones.

Comparing ArrayUDF with SciDB, RasDaMan, EXTASCID, and Spark in executing the standard "window" operator. As we discussed in the previous sections, the Poisson equation solver cannot be expressed directly using the "window" operators of SciDB and RasDaMan. To compare the performance and versatility of ArrayUDF, we compare the performance of the "window" operator these systems provide, where the operation is to compute the average for a 2×2 window on 2D data and a $2\times2\times2$ window on 3D data. We also include EXTASCID and Spark for a complete comparison. For fairness of the evaluation platform for all these systems, we have installed them on a single Linux desktop. The desktop has two



(a) Time for evaluating "window" operators.

(b) Time for solving the Poisson equation solver.

Figure 11: Comparison of ArrayUDF with the state-of-the-art data processing systems.

CPU cores (Intel i7-5557U CPU with 3.10 GHz), two threads per core, and a local disk (Seagate ST1000LM014-1EJ1) with the EXT4 file system. We show these test results in Fig. 11a. The original datasets *S*1 and *S*2 are too large for the processing on a single node. Thus, we used smaller size datasets for tests. The tested 2D dataset has an array with a size of (10000, 30000) and the 3D dataset has a size of (1000, 1000, 400). The chunk size for all the systems is set to be (5000, 15000) for 2D and (1000, 1000, 100) for 3D which give 4 chunks per array—equivalent the number of threads in the system.

RasDaMan has the highest execution time because the version that is publicly available only supports "inter-query parallelization" and, thus, it can only use one core to perform the calculation. Spark can use its Map and Reduce interface to implement the window-based average, however, it needs to duplicate the data for different windows. As a result, it has the second highest execution time. In terms of SciDB, it handles each window independently and inefficiently [23], thus SciDB is also much slower than ArrayUDF. We also scale the SciDB tests to 11 nodes on the Edison system for *S*1 and *S*2 datasets. In these tests, we observe similar results. EXTASCID provides a robust array data storage model, but it has no portable window-like operator yet. For comparison, we have implemented the window function manually as a generalized linear aggregate (GLA) function, which is the abstract interface in EXTASCID. Basically, the window functions are written in C++ and they use the EXTASCID I/O driver to access the arrays. Even with this hand-optimized code, the execution times of EXTASCID are similar to those of ArrayUDF. In other words, the performance of ArrayUDF is very close to the C++ hand-optimized code. Thus, we conclude that ArrayUDF provides more flexibility to define operations, and it is as efficient as highly-optimized code in performing "window" based analysis tasks.

Comparing ArrayUDF with Spark on solving the Poisson equation on datasets *S*1 and *S*2. In this test, we compare the performance of using ArrayUDF and Spark to solve the Poisson equation with 64 computing nodes. By using ArrayUDF, we can directly define and execute the Poisson equation solver on arrays stored in datasets *S*1 and *S*2, as shown at the beginning of this section. To use Spark for expressing these operations, we apply the "`flatmap`" and "`reducebykey`" functions. The general idea is that all the adjacent stencil cells required by a cell are viewed as a group and all cells within the group are aggregated onto a single reducer to perform the computation. Since each cell belongs to multiple groups (of its neighbors), we use "`flatmap`" to transform each cell into multiple (key, value) pairs, where the key is the group ID and the value is the actual data. After "`flatmap`" finishes, we use "`reducebykey`" to consolidate all the cells with the same group ID together. Basically, each reducer is responsible for computing a cell in the result array. One may argue that users can write a specific Map function without a Reduce funtion to solve the Poisson equation. However, as discussed in previous sections, both ArrayUDF and Spark aim at providing a generic UDF mechanism for users to express high-level operations without burdening users to write custom programs for performing different operations.

We compare the performance of ArrayUDF and Spark in executing the Poisson equation solver in Fig. 11b. For datasets *S*1 and *S*2, ArrayUDF is 11 and 118 times faster than Spark, respectively. Since the Poisson equation solver on 3D data needs to access more adjacent cells than on 2D data, ArrayUDF achieves a higher performance improvement for processing *S*2. Basically, the more neighbor cells are required, the more times the entire array is replicated in Spark. As the size of the data to be replicated increases, Spark spends more time on communication and shuffling for "`flatmap`" and "`reducebykey`" functions. Compared with Spark, ArrayUDF allows users to define and execute the Poisson equation solver directly on an array. Thus, there is no data replication during the runtime. Moreover, as ArrayUDF can automatically build the ghost zone using a trial-run, the expensive communication is avoided.

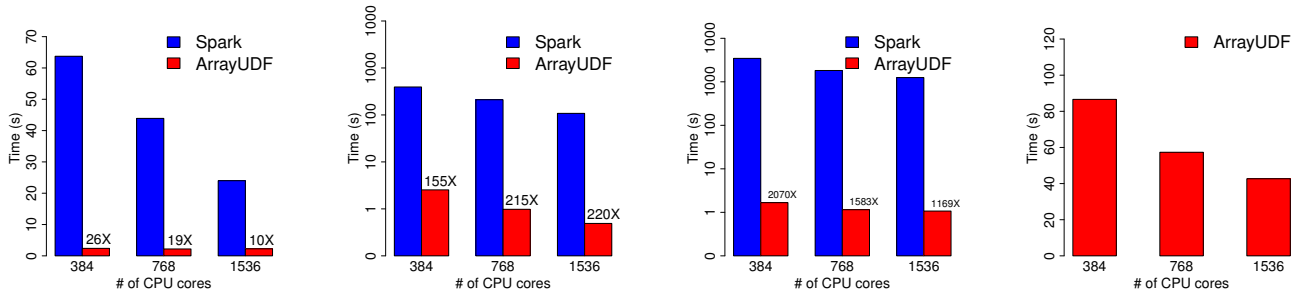
4.2 ArrayUDF for real scientific data analysis

We have evaluated ArrayUDF to perform several real-world analysis tasks on four scientific data sets: S3D, VPIC, MSI, and CoRTAD. We summarize the properties of these datasets and the analysis operations performed in this study in Table 3. A brief background is presented in § 2. To compare performance, we use Spark to implement the same analysis tasks on these datasets. The method to implement these analysis operations is the same as the one we use for the Poisson solver on the *S*1 and *S*2 datasets. We test each task with different numbers of CPU cores, scaling from 384 (i.e., 16 compute nodes) to 1536 (i.e., 64 nodes). As shown in Fig. 12, for S3D, VPIC, and MSI analysis, ArrayUDF outperforms Spark by up to 26×, 220×, and 2070×, respectively. For CoRTAD, Spark crashes due to out-of-memory (OOM) errors. We discuss the performance of each of these analysis tasks in the following.

Table 3: Real-world scientific datasets and operations

dataset	Rank	Size (GB)	Operation
S3D	3D	301	Vorticity computation
MSI	3D	21	Laplacian calculator
VPIC	3D	36	Trilinear interpolation
CoRTAD	3D	225	Simple moving average

S3D. For the S3D dataset, we compute the vorticity, which is defined in Eq. 2. For a single point in S3D, the vorticity computation needs the values of four neighbors in total and two neighbors per direction. In this test, we use the dataset in the x direction. This data has $1100 \times 1080 \times 1408$ dimensions, resulting in a file size of 22 GB. We show the execution time for vorticity computation using ArrayUDF and Spark in Fig. 12a. On average, ArrayUDF is 18 times faster than Spark. We observe similar performance speedup in the y and z dimensions. In summary, we observe that ArrayUDF is more efficient than Spark in computing vorticity over the S3D dataset.



(a) S3D (b) MSI (c) VPIC (d) CoRTAD

Figure 12: Evaluation with analyzing real-world scientific datasets. ArrayUDF is up to 2070X faster than Spark. For the CoRTAD dataset, where we show the performance only for ArrayUDF, Spark crashes due to out-of-memory (OOM) error.

MSI. The MSI data [34] used in our test contains a $123 \times 463 \times 188960$ 3D array which has 21 GB in file size. This array contains images of a potato eye. The operation of interest on this dataset is the Laplacian calculator, as presented in Eq. 3. In our tests, we observe that ArrayUDF is 196 times faster than Spark. Since the Laplacian calculator needs five neighbor cells, but vorticity on S3D only needs *two*, ArrayUDF achieves a higher speedup in this test. As discussed before, the more adjacent cells are needed in UDF, the higher speedup ArrayUDF can achieve.

VPIC. In the space weather simulation using VPIC, the total size for the magnetic field data is 36 GB and the data has x, y and z dimensions [11]. The dimensions of the dataset are $2000 \times 2000 \times 800$. We use both ArrayUDF and Spark to implement the trilinear interpolation shown in Eq. 4. For simplicity, we assume that there is a particle in a cell. But, our ArrayUDF can be extended to support more flexible interpolation. We report the results for the x dimension in Fig.12c—we obtain similar results for dimensions. On average, ArrayUDF is 1607X faster than Spark in this test.

CoRTAD. CoRTAD is a collection of sea surface temperatures (SST) [1]. It contains the weekly temperature for 1617×4320 sites from 1981 to 2010. Thus, it is a 3D array with size $1617 \times 4320 \times 8640$, where the third dimension is the number of weeks (i.e., 8640 weeks). We compute a moving average based smoothing on this data. Basically, for each site, we smoothen its temperature based on the month, i.e., compute the average of four neighbor array cells. In Fig. 12d, we depict only the performance of ArrayUDF. For the tests with the same number of CPU cores, Spark crashes with out-of-memory errors. The reason for the OOM errors is the need for Spark to replicate the entire dataset 4 times in oder to compute the moving average. Moreover, since we also need to use the (key, value) pair structure and the Scala object type to store the data, the total data size increases by more than 8 times, which contributes to the OOM errors. Meanwhile, ArrayUDF successfully completes the moving average computation for this large dataset without any memory footprint pressure.

5 RELATED WORK

UDFs are widely supported by relational database management systems, such as SQL Server [13], IBM DB2 [44], MySQL [42], and PostgreSQL [30]. MonetDB [33] has an extension to support vector-based UDFs that take advantage of the columnar data representation. The UDFs within these systems are based on the relational set semantics and permit users to define operations at tuple or table level—known as User-Defined Aggregates (UDA) [15]. The key difference between ArrayUDF and these types of UDFs is that ArrayUDF is developed for multi-dimensional arrays and allows more general structural locality based operations.

In MapReduce [18], UDFs consist of two steps—Map and Reduce. Map applies the UDF on a single key-value pair and produces one or more key-value pairs as output. Reduce consolidates key-value pairs having the same key and then applies another UDF. Reduce requires expensive data shuffling to repartition data across nodes. Among all extensions to MapReduce [4, 10, 21, 22, 40], Spark provides a much richer set of UDFs for iterative in-memory analysis. Compared to MapReduce and its extensions, ArrayUDF requires a single step to express UDFs on a set of related array cells, thus avoiding the expensive shuffle stage.

Array database systems such as RasDaMan[6], AML[27], SciDB[9], SciQL[24] and EXTASCID[17], have UDF support. Rusu et al. [35] provides a complete survey on this topic. Typically, these array UDFs follow a similar idea to relational database systems, where users define an operation on a single element, i.e., array cell. If the UDF requires multiple adjacent cells, these have to be mapped into tuples and then apply the UDF. ArrayUDF is a novel UDF type for arrays that allows users to define operations directly on adjacent array cells, without any mapping. Moreover, to support efficient data access in a shared-disk system, ArrayUDF performs dynamic chunking and ghost zone building. In contrast, array database management systems have a shared-nothing architecture and rely on data ingestion to handle chunking and ghost zones. A specific type of array UDF is the window-based *Apply* operator [27]. SciDB provides this function via a highly optimized *window* operator. RasDaMan has a similar operator named *condense*. While similar to ArrayUDF, these operators support only fixed-size windows and the operations on a window are limited. ArrayUDF generalizes the window shape and the operations for the cells within a window. *Join* operators are found to be expensive to support these operations because of data replication [20, 43]. SAGA [41] explores aggregate operations on scientific arrays stored in native data formats.

The domain-specific languages (DSL) [7, 29] share similarity with our ArrayUDF in leveraging stencil behaviors to improve performance of array based data analytics. These DSLs are mostly developed as programming language and compiler extensions with the goal of increasing the efficiency of calculation and memory data access. But, in ArrayUDF, we developed a flexible computing model towards large-scale data analytics, i.e., to derive knowledge from the multidimensional arrays in data files directly. The ArrayUDF generalizes the *MapReduce – like* systems to realize a wider range

of operations on arrays, including the stencil operator supported by these DSLs. Beyond that, ArrayUDF has the capability to express other types of operations, such as aggregation, filtering, etc. Moreover, ArrayUDF provides optimizations (e.g., *trial run* based ghost zone determination, disk layout aware/unaware partitions for multidimensional array) for efficiently processing out-of-core data sets, which are too large to be quickly loaded into memory, even with multiple nodes. Our analytical performance model for ArrayUDF can also be used as the foundation to optimize these existing stencil DSL systems in handling large-scale scientific datasets on disk.

6 CONCLUSIONS AND FUTURE WORK

Customized data analysis, especially for data stored as multidimensional arrays in large files, is a common method to extract insights from data and the UDF mechanism is a general strategy to support this analysis. However, the current UDF implementations are not able to effectively express the structural locality present in most array based data analysis operations, such as computing the vorticity for a 3D flow field, the moving average for a time series, etc. These operations have to be expressed as expensive *join* or *reduction* operations. We design ArrayUDF to easily capture the generalized structural locality and implement an *in situ* processing system optimized for multi-dimensional arrays. The generalized structural locality mechanism of ArrayUDF allows users to define a different operation on each neighbor separately and, therefore, express more complex customized data analysis. The *in situ* processing system automatically partitions data stored in raw data files and creates ghost zones for the arrays stored in files without an expensive data ingestion phase. Our automatic data partitioning minimizes the execution time based on a an analytical model of the expected performance. It is also able to take advantage of the layout of arrays in input data files. Our evaluation using a number of different scientific datasets show that ArrayUDF is up to three orders of magnitude faster than Apache Spark. In future, we plan to enhance ArrayUDF with a filter feature to reduce loading of unnecessary array cells in the result arrays. We will also explore the use of ArrayUDF to process stream and real-time data.

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 and by a DOE Career award (program manager Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

REFERENCES

[1] The Coral Reef Temperature Anomaly Database (CoRTAD) Version 4 - Global, 4 km Sea Surface Temperature and Related Thermal Stress Metrics for 1981-10-31 to 2010-12-31 (NODC Accession 0087989), 2012.
[2] M. Abadi, A. Agarwal, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.
[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *SIGMOD '12*, 2012.
[4] L. Antova, A. El-Helw, M. A. Soliman, Z. Gu, M. Petropoulos, and F. Waas. Optimizing Queries over Partitioned Tables in MPP Systems. In *SIGMOD*, 2014.
[5] P. Baumann. Management of Multidimensional Discrete Data. *VLDB J.*, 1994.
[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. *SIGMOD Rec.*, 27(2):575–577, 1998.
[7] M. Bianco and B. Cumming. A generic strategy for multi-stage stencils. In *Euro-Par'14*, pages 584–595, 2014.
[8] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *SIGMOD'2014*.
[9] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.
[10] J. B. Buck, N. Watkins, and et al. SciHadoop: Array-based Query Processing in Hadoop. In *Supercomputing Conference (SC)*, 2011.
[11] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, et al. Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation. In *SC*, 2012.
[12] N. Chaimov, A. Malony, S. Canon, C. Iancu, and et al. Scaling Spark on HPC Systems. In *HPDC 2016*, 2016.
[13] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB '97*, 1997.
[14] J. H. Chen, A. Choudhary, B. de Supinski, and et al. Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.
[15] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.
[16] Y. Cheng and F. Rusu. Astronomical Data Processing in EXTASCID. In *SSDBM 2013*.
[17] Y. Cheng and F. Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *Distributed and Parallel Databases*, 33(3):277–317, 2015.
[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
[19] B. Dong, S. Byna, and K. Wu. SDS: A Framework for Scientific Data Services. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 27–32, New York, NY, USA, 2013. ACM.
[20] B. Dong, S. Byna, and K. Wu. Spatially Clustered Join on Heterogeneous Scientific Data Sets. In *2015 IEEE Big Data*, pages 371–380, Oct 2015.
[21] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proc. VLDB Endow.*, 2(2):1402–1413, Aug. 2009.
[22] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major Technical Advancements in Apache Hive. In *SIGMOD '14*, 2014.
[23] L. Jiang, H. Kawashima, and O. Tatebe. Efficient Window Aggregate Method on Array Database System. *Journal of Information Processing*, 24(6):867–877, 2016.
[24] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In *AD '11*, 2011.
[25] D. E. Keyes, Y. Saad, and D. G. Truhlar, editors. *Doman-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*. SIAM, Philadelphia, PA, 1995.
[26] J. Liu, E. Racah, Q. Koziol, and et al. H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems. In *Cray User Group*, 2016.
[27] A. P. Marathe and K. Salem. A Language for Manipulating Arrays. In *VLDB '97*.
[28] V. Markl. Breaking the Chains: On Declarative Data Analysis and Data Independence in the Big Data Era. *Proc. VLDB Endow.*, 7(13):1730–1733, Aug. 2014.
[29] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *SC '11*, pages 11:1–11:12, New York, NY, USA, 2011. ACM.
[30] B. Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[31] M. A. Onabid. Solving Three-Dimensional (3D) Laplace Equations by Successive Over-Relaxation Method. *AJMCSR*, 5(13), 2012.
[32] Y. Peng et al. The design of the variable sampling interval generalized likelihood ratio chart for monitoring the process mean. *Qual. Reliab. Engng. Int*, 31(2), 2015.
[33] M. Raasveldt. Vectorized UDFs in Column-Stores (Master Thesis), 2015.
[34] O. Rübel, A. Greiner, and et al. OpenMSI: A High-Performance Web-Based Platform for Mass Spectrometry Imaging. *Analytical Chemistry*, 2013.
[35] F. Rusu and Y. Cheng. A Survey on Array Storage, Query Languages, and Systems. *CoRR*, abs/1302.0103, 2013.
[36] A. Shoshani and D. Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2010.
[37] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD'2011*. ACM, 2011.
[38] H. Tang, S. Byna, S. Harenberg, et al. Usage pattern-driven dynamic data layout reorganization. In *CCGrid'2016*, pages 356–365, May 2016.
[39] The HDF Group. HDF5 User Guide, 2010.
[40] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In *CCGrid'2012*, pages 443–450, 2012.
[41] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array Storage As a DB with Support for Structural Aggregations. In *SSDBM '14*, New York, NY, USA, 2014. ACM.
[42] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
[43] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity Join over Array Data. In *SIGMOD 2016*.
[44] P. C. Zikopoulos and R. B. Melnyk. *DB2: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 2001.