DASSA: Parallel DAS Data Storage and Analysis for Subsurface Event Detection

Bin Dong*, Verónica Rodríguez Tribaldos*, Xin Xing†, Suren Byna*, Jonathan Ajo-Franklin*‡, Kesheng Wu*,
*Lawrence Berkeley National Laboratory, Berkeley, CA, USA. Email: {dbin, VRodriguezTribaldos, sbyna, kwu}@lbl.gov
†Georgia Institute of Technology, Atlanta, GA, USA. Email: xxing33@gatech.edu
‡Rice University, Houston, TX, USA. Email: ja62@rice.edu

***Abstract*—Recently developed distributed acoustic sensing (DAS) technologies convert fiber-optic cables into large arrays of subsurface sensors, enabling a variety of applications including earthquake detection and environmental characterization. However, DAS systems produce voluminous datasets sampled at high spatial-temporal resolution and consequently, discovering useful geophysical knowledge within these large-scale data becomes a nearly impossible task for geophysicists. It is appealing to use supercomputers for DAS data analysis, as modern supercomputers are capable of performing over a hundred quadrillion FLOPS operations and have access to exabytes of storage space. Unfortunately, the majority of geophysical data processing libraries are not geared towards these supercomputer environments. This paper introduces a parallel DAS Data Storage and Analysis (DASSA) framework to enable easy-to-use and parallel DAS data analysis on modern supercomputers. DASSA uses a hybrid (i.e., MPI and OpenMP) data analysis execution engine that supports a user-defined function (UDF) interface for various operations and automatically parallelizes them for supercomputer execution. DASSA also provides novel data storage and access strategies, such as communication-avoiding parallel I/O, to reduce the cost of retrieving large DAS data for analysis. Compared with existing data analysis pipelines used by the geophysical community, DASSA is 16× faster and can efficiently scale up to 1456 computing nodes with 11648 CPU cores.**

I. INTRODUCTION

Cutting-edge scientific studies nowadays usually create and analyze a large volume of data [27]. For example, in geophysics, the monitoring of subsurface processes has shifted from direct measurements from drilling, to non-invasive imaging techniques, such as distributed acoustic sensing (DAS) [23], [21]. DAS can record active and ambient vibrations to characterize the subsurface environment with high resolutions for both spatial and temporal scale. These high-resolution recordings are important for a variety of geophysical applications, such as earthquake detection and groundwater monitoring. However, this high-resolution acquisition can easily produce massive amounts of data, usually in the order of terabytes per day [16]. Additionally, the recorded DAS data may contain lots of undesired noise. As a result, extracting useful information for geophysical explorations from large-scale DAS data has become nearly impossible task so far.

High performance computing (HPC) systems, or generally known as supercomputers, are desirable platforms to perform large-scale DAS data analysis because of their thousands of computing nodes and exabytes of storage space [27], [28], [29]. Currently, the normal practice within the DAS community is to use MATLAB or other digital signal processing (DSP) systems for data analysis. Obviously, scaling these systems to petabytes of DAS data analysis still suffers from issues in performance and productivity on HPC systems [25]. In this paper, we explore new methods and algorithms to develop an efficient DAS data analysis system. Specifically, we anticipated three major challenges associated with developing such a new and scalable DAS data processing system:

- The overall DAS data size is extremely large, but scattered among large numbers of small files [16], [32]. These files are usually recorded and stored per a small time granularity, such as a minute. A data processing pipeline with intrinsic parallel is needed to handle such large DAS datasets. There is a constant overhead in accessing a file on a typical disk-based file system. This constant overhead can become a bottleneck for I/O operations when accessing a large number of files. It is important to keep I/O overhead low.
- Different analysis operations are required in different DAS data investigations. For example, one researcher may apply time-domain analysis like local-similarity to the data [18]. But, other researchers may apply frequency domain operations, such as a fast Fourier Transform (FFT) [16]. Even for the same operation, users may apply it to different datasets produced per week, per month or other time interval.
- The new data analysis system should be abstract enough to hide all data-management and parallelization tasks under the hood while delivering optimal performance on HPC systems [15]. Our data analysis system should allow the application scientists to concentrate on their core analysis operations, while hiding the details of a HPC system with its massive number of computing nodes, complex storage layers, and inter-communication networks.

To address these requirements and their associated challenges, we present here the DAS data storage and analysis (DASSA) framework. DASSA provides a scalable and easy-to-use system for geophysicists to perform DAS data analysis on HPC systems. It supports various DAS data analysis operations though a single user-definable interface and more importantly, hides all underlying data management, communication and parallelization tasks for these operations. In summary, the technical contributions of our paper are listed below:

- We introduce the DASSA framework (Section III) for paral-

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00035

lel DAS data analysis. DASSA has a storage engine and an analysis execution engine. The storage engine stores the DAS dataset in a HPC storage system for efficient parallel access during analysis. The analysis engine has a high-level abstraction for users to customize domain-specific analysis operations and also to transparently parallelize these operations on HPC systems.

- We propose a virtually concatenated array (VCA) data model (Section IV) to store DAS data and preserve DAS related metadata for analysis. We develop a search tool, named `das_search`, for users to find and merge small DAS files together as contiguous input of the analysis code. VCA can store the merged file without duplicating original data but with a small amount of metadata.
- We develop a hybrid ArrayUDF execution engine (Section V) to execute DAS data analysis operations on multicore computing nodes efficiently. The hybrid execution engine allows multiple threaded analysis code to share data without duplication for multiple cores on a single computing node. It also reduces the number of I/O calls to avoid high IOPS[1] pressure on storage devices.
- We propose a communication-avoiding parallel I/O algorithm (Section IV-B) to reduce the cost of accessing DAS data. This communication-avoiding parallel method reduces the number of expensive communications for accessing the VCA data built on thousands of small DAS files.
- We apply all of these capabilities to two state-of-the-art geophysical data analysis problems (Section V-C), which were simply impossible to address before due to challenges posed by the large volume of DAS data and its complexities.

Our evaluations (Section VI) manifest that DASSA is at most 16X faster than a Matlab-based system, which is currently used by a DAS team to perform data analysis on a single node and it is popular in geophysical community. By contrast, DASSA shows its scalability up to 1456 computing nodes (i.e., 11648 CPU cores) with high parallel efficiency.

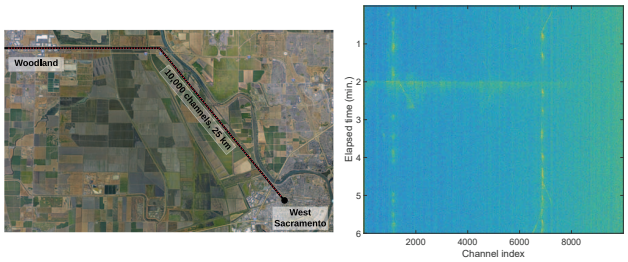
II. Preliminaries

A. Distributed acoustic sensing (DAS)

Distributed acoustic sensing (DAS) [2] is an emerging and rapidly developing technology that is used to record distributed measurements of strain or strain-rate along commercial fiber-optic cables in the subsurface. DAS has now been applied successfully in many geophysical applications including earthquake detection [19], seismic imaging [16], and permafrost condition monitoring [2]. Compared with traditional sensing methods, DAS enables efficient investigation of the subsurface at high spatiotemporal resolutions. This continuous and high-resolution monitoring, in turn, produces large amounts of recorded data. These large-volume DAS datasets have become an obstacle for geophysicists to find useful information for geophysical research, as explained below.

DAS system contains large number of channels (i.e., sensors). These channels can generate enormous amounts of data,

[1]IOPS: Input/output operations per second



(a) (b)

Fig. 1: The DAS system used in this study that runs on an unused fiber-optic cable from West Sacramento to Woodland, CA. (a) The dotted line shows the cable going through highways, bridges, and other diverse environments. (b) A 6-minute DAS data, illustrated as a 2D array indexed by channel and time, which contains lots of noise and some signals from moving cars and a M4.4 earthquake.

even for a short time period on a short fiber. The DAS dataset [2] we work with comes from a 25 kilometer (km) fiber-optic cable running between the cities of West Sacramento and Woodland (California), as illustrated by Figure 1a. It has 11,648 channels along the cable and the sampling rate of each channel is 500 Hz, giving around 1 terabyte per day. These data are stored in 1440 files per day and each of them contains a 1-minute recording. As shown in the figure, the fiber-optic cable used traverses diverse noise environments, leading to different and complicated signals recorded along the sensing array. An overall illustration of the signals recorded by this DAS array is shown in Figure 1b, which contains a 6-minute record around the time when a magnitude M4.4 earthquake occurred in Berkeley, California. Thanks to the high density of channels provided by DAS, seismic waves can be easily identified travelling across the array.

B. ArrayUDF

Based on the classical idea of structural locality [20], we developed ArrayUDF [15], a parallel framework for large scale scientific data analysis. For specific types of calculations, ArrayUDF can be a thousand time faster than Spark [34], SciDB [6], and other modern data processing frameworks. ArrayUDF has the following abstraction:

$$B = Apply(A, f),$$

where A is an input multidimensional array and B is an output multidimensional array. $Apply$ is a function provided by ArrayUDF to run the user-defined function (UDF) f from A to B. Function f can be customized for different operations by different users. The $Apply$ function is executed on each MPI process [24] on multiple computing nodes with the corresponding data partition processed automatically. Meanwhile, ArrayUDF can also build a ghost zone for each data block to avoid communication during the execution.

To support a structural locality-aware operation on the array, ArrayUDF introduces a data structure (or abstract) called

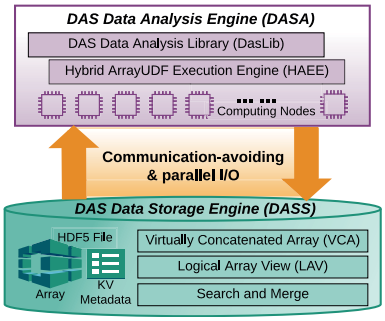


Fig. 2: Overview of DASSA framework

`Stencil`. The `Stencil` data structure enables users to express the user-defined function (f) on a logical array cell and its neighborhood. An example of using `Stencil` to express a three point moving average on a 1D time series data is:

```
f(Stencil S1){
  Stencil S2;
  S2 = (S1(-1) + S1(0) + S1(1))/3;
  return S2;
}
```

where $S1$ is the input Stencil and $S2$ is the output Stencil. The $S(i)$ refers to the value at offset i from current cell. Combining it with *Apply* above, users can run f on the whole time series data in parallel. One challenge of applying ArrayUDF to analyze DAS data is that ArrayUDF takes a single array as input but DAS data are stored in many small files. Also, ArrayUDF only supports MPI-based parallelization which limits the massive data sharing across MPI processes on the same computing nodes. As a result, DAS data may also need to be duplicated across these MPI processes. We address these technical challenges in this paper.

III. Overview of DASSA Framework

In this section, we introduce the DASSA framework with the goal of providing an easy-to-use and scalable system for DAS data storage and analysis. We present a high-level overview of DASSA in Fig. 2. The major components of DASSA are a DAS data storage engine (DASS) and a DAS data analysis engine (DASA). DASS provides essential functions to search, merge (VCA) and subset (LAV) DAS data for analysis. DASS also provides a communication-avoiding parallel I/O method to reduce the cost of accessing DAS data stored in small files. DASA contains a DAS data analysis library (DasLib) for popular DAS data processing operations. DASA also provides a hybrid ArrayUDF execution engine (HAEE) to run all these operations in parallel. Details of these sub-components are discussed in the following sections.

IV. DASS: DAS Data Storage Engine

DAS data storage engine (DASS) supports efficient and flexible DAS data storage and access operations. Based on the Hierarchical Data Format version 5 (HDF5) file format [30], DASS provides an array-data model for storing DAS data and a key-value model for storing metadata. More details on HDF5 can be found in [30]. Our work focuses on how to define new data structures and how to develop new algorithms for accessing DAS data stored in a large number of HDF5 files. Given that DAS data is often scattered over a number of small files, DASS provides functions to search and concatenate the data with different criteria. DASS also contains a novel communication-avoiding method to reduce the I/O cost when accessing concatenated DAS data.

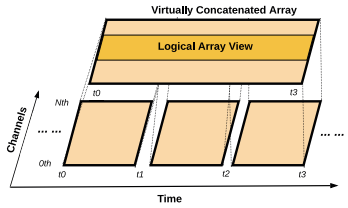


Fig. 3: Example of using VCA and LAV to merge and subset DAS data built from three small files.

DASS Array Data Model. DASS uses the multidimensional array from the HDF5 file format as the basic data model to store the DAS data. Most DAS data can be represented by a 2D array: $[C, T] \mapsto A$, where C is the channel set, T is the set of time samples, and A is the amplitude. In certain cases, a multidimensional array is needed to store intermediate data during analysis. For example, during the stacking operation of the DAS data analysis pipeline [16], a 3D data array with a striping size as the third dimension may be produced.

Based on the above data model, DASS provides two advanced array extensions called the Virtually Concatenated Array (VCA) and the Logical Array View (LAV), as shown in Fig. 3. VCA merges DAS data files sampled at contiguous times as a single input for data analysis operations. LAV allows users to select a subset of a larger array as the input of DASA. Details of the two extensions are discussed below:

- **Virtually Concatenated Array (VCA).** Most of DAS data analysis operations are performed on data collected at a contiguous period, such as a few hours, days, or months. Usuasly, one DAS dataset only contains data of 1 minute. Therefore, it is important for DASS to efficiently merge different DAS datasets together to create the input for analysis operations. To support such functionality, DASS provides a search function (see Section IV-A) to find target small files for merging. After that, DASSA allows users to create either a real concatenated array (RCA) or a virtually concatenated array (VCA) from small files. As indicated by our evaluation results and discussions in Section IV-A, there are trade-offs between RCA and VCA. Overall, in comparison with RCA, VCA does not duplicate data during construction but only records metadata (e.g., dataset names). More importantly, VCA needs less construction time than RCA because it only accesses metadata. The challenge of using VCA is that analysis operations may have high I/O overhead to access an VCA. Because a large request targeted on a VCA may be broken into numerous small requests for individual files contained within VCA. To address this issue, DASS also provides a novel communication-avoiding method (presented in Section IV-B) to reduce the cost of accessing VCA.

- **Logical Array View (LAV).** LAV provides a logical subset view of an array, which is similar to the hyperslab of HDF5 file. As shown by the example in Fig. 3, LAV can help users to run the analysis on a subset of interested channels.

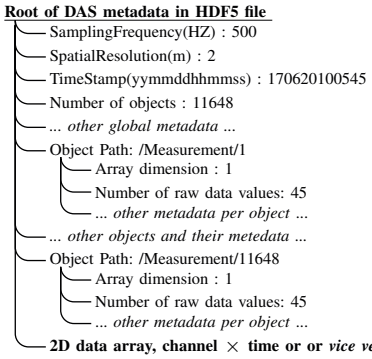


Fig. 4: The hierarchical metadata (w/ data) structure created by DASSA in HDF5 file to store DAS data acquired per minute. Each item is a key-value pair with (`key : value`) format.

DASS Metadata Model. The metadata for DAS data includes significant information about how/where/when the data are recorded. To preserve these metadata for analysis, DASS uses a key-value (KV) pair structure to store these metadata. As shown by Figure 4, the metadata contain two levels of KV list. The first level of KV-list contains the global metadata that applies to the whole dataset, such as the number of objects (i.e, channels). For each channel, its metadata is stored as the second level of KV-list. These metadata provides useful information for users to query data for different purposes. One common operation is to filer the data within a time interval, as discussed in the following subsection.

A. DAS File Search and Merge

As mentioned in previous sections, DAS data are scattered across many files. Users can have different operations to run on different parts of the data. One of the more common data operation is to search and merge files within a time interval when events of interest (e.g., traffics or earthquakes) occur. Based on metadata stored, DASS provides two types of search function as the command line tool `das_search`. The first one is the *time-stamp based range query*. It specifies a start time (`-s`) and the number of samples (`-c`) after the start time. This type of search is simple enough to capture most of search requirements. The second one is a *regular expression based query interface (`-e`)*. This option is for advanced users to define an arbitrary search criterion via `regex` pattern. Below are examples demonstrating how to use `das_search` to find three files after the time-stamp `170728224510`:

```
Type 1: das_search -s 170728224510 -c 2
Type 2: das_search -e 170728224[567]10
```

Once these targeted files are found by DASS, users can also use DASS to create a real concatenated array (RCA) or a virtually concatenated array (VCA). Table I presents the high-level comparison of RCA and VCA. Details are discussed in following paragraphs:

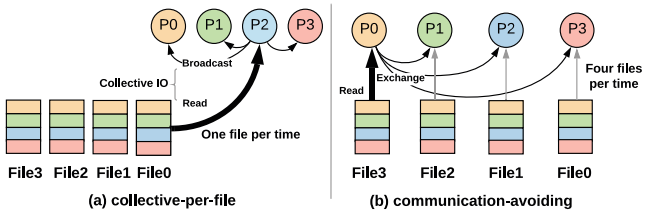


Fig. 5: A comparison of two I/O methods to read small DAS files. This example uses four processes (P0 $\sim$ P3) and four small files (File0 $\sim$ File3). Each process reads a subset of all four files, marked with different colors. For example, P0 reads all yellow subsets (i.e., the 1st subset) of four files. a) "collective-per-file" method: all processes read a file at a time with collective-I/O per file. b) "communication-avoiding" method: each process reads a file and then, all processes have a all-to-all data exchange to obtain their own data portion. We omit arrows denoting communication from P1, P2 and P3 for simplification.

- RCA concatenates all necessary DAS files as a single HDF5 file. RCA may reduce the burden of managing small files, but it lacks flexibility to allow a file for different analysis operations without duplication. It also may double the storage space requirement during the construction. Also, concatenating the DAS files into a single files tends to be tedious process because it accesses the whole data. But, reading a single large file in parallel has been supported by prior optimizations in the community [13].
- VCA creates a logical file which only contains the metadata (e.g., name) of all files to merge. During the read operation, these metadata are analyzed to find the exact location of the original file. VCA has small overhead in construction since it only performs small metadata operation. Users can also merge the same file into different VCAs without extra copies. However, the issue with VCA is that it may introduce significant I/O cost for later analysis operations because the data of VCA are actually scattered among small files and accessing the data in parallel may needs lots of small I/O and communications. To mitigate this issue, we introduce a communication-avoiding method to reduce the I/O cost of accessing VCA, as presented in the following subsection.

TABLE I: Comparison between RCA and VCA

	Construction		Duplication across groups	Parallel I/O
	Extra Space	Overhead		
RCA	100%	High	Exist	Yes
VCA	0%	Low	No	NO

B. Communication-avoiding Parallel I/O

As discussed in the previous section, VCA is a flexible approach for merging lots of small DAS files for analysis without requiring substantial extra storage space and high computational overhead during the VCA construction. However, the access performance of VCA in parallel is low because it may break large I/O into small operations on individual files. As shown in our experimental study in subsections VI-B and VI-C, the time for I/O operations could be a dominant

factor in the overall performance of the DAS analysis pipeline. Based on the typical I/O access patterns of DAS data analysis operations [18], [2], [16], we proposed a communication-avoiding I/O method to address this issue.

The most typical DAS data analysis operation is to apply time series analysis methods to a single channel or a set of neighboring channels. The I/O access pattern is to first read the data of a (or a few) channel(s) from all small files and then concatenate them together. Considering a parallel running analysis code with p processes and n files and each process access equal part $(1/n)$ of each file, there are at most $O(p \times n)$ I/O requests issued by the analysis code to get the data. Dealing with large number of I/O requests is challenging for the underlying storage system [14], [12]. Another factor may degrade the overall performance is the possible communication that implicitly or explicitly happens during I/O operations. For example, collective I/O is a normal optimization for HPC storage system. However, its "merge-read-broadcast" pattern may introduce lots of inter-process communication, which we will explain with details in the following paragraph. To address these issues, there are two I/O design considerations: 1) all processes read (i.e., share) each file one by one, called "collective-per-file" method [2], 2) each process first reads a file independently and then exchanges data after the read, called "communication-avoiding" method. Figure 5 illustrates these two methods with their discussions in following paragraphs.

To use the "collective-per-file" method, all processes share each small file and read them one by one, as shown by Figure 5 (a). When all processes share the same file, it is normal to apply the collective I/O [33]. Using collective I/O can significantly reduce the cost of reading data because it merges small reads on all processes into large ones, and therefore reduces disk seeks. In this case, there are at least $O(n)$ I/O requests. However, the most popular implementation of collective I/O follows the "merge-read-broadcast" pattern. For p processes, their read requests are merged by certain number (say $k, k < p$) of processes at first. Then, these k processes issue I/O requests in a larger and contiguous manner. Once these k processes finish the read, they broadcast the results back to all p processes. Assuming that the collective I/O can always merge small reads into large ones, the performance bottleneck may become the broadcast operation when p is large. Since this method reads a file a time, each file needs at least one broadcast. For n small files, it needs $O(n)$ broadcasts. When the number of DAS files is large, this I/O strategy can significantly increase the data access time, too. As we discussed in the background section, long-term DAS deployments with continuous recording tend to create infinitely many files for analysis operations to read.

The second I/O design method we proposed to access a VCA file is the "communication-avoiding" method (Figure 5 (b)). This method allows each process to read the entirety of a single file and then it exchanges the data with other processes. In this way, the read on each process tends to be fast because it reads the whole data with a single I/O call, giving at least $O(n)$ I/O requests (same as the "collective-per-file" method). Also, the read on a single process tends to be contiguous. When all processes read the data at the same time, the I/O is paralleled for high throughput. For the data exchange stage, all processes can participate at the same time. It allow lots of current transfers among node pairs. Based on well-proven theory and practice about the idea of communication-avoiding during linear algebra [10], these concurrent data transfers can reduce the overall time for data exchange. When we have p processes and n files, the number of communications in the key step is $O(n/p)$. Comparing it with the "collective one-by-one" method described above, with $O(n)$ broadcasts, our method can avoid lots of expensive communications and therefore has better performance in accessing the VCA file built from lots of small files.

[2]The "collective-per-file" method is different from the collective buffering or two-phase I/O of MPI-IO. The former works on multiple files and the later on a single file.

V. DASA: DAS DATA ANALYSIS ENGINE

DAS data analysis engine (DASA) is the second major component of the framework to enable various DAS data analysis operations. DASA has two subcomponents: a DAS data analysis library (DasLib) and a hybrid ArrayUDF execution engine (HAEE). DasLib provides several popular and sequential DAS data analysis operations. To enable processing of large-scale DAS data with hundreds or even thousands of CPU cores on HPC systems, DASA provides HAEE to automatically and transparently run DasLib in parallel without the burden on geophysicists to parallelize the analysis code.

A. DasLib: DAS Data Analysis Library

MATLAB and Python have existing libraries [5] for seismic signal processing. Because of the scalability issues of their native run-time environment, these libraries, however, are not well-fitted for DAS data analysis on HPC systems. New data analysis systems such as ArrayUDF have the capability to deal with terabytes of scientific data efficiently on HPC systems. However, ArrayUDF only provides a run-time system for analysis codes that are provided by users as user-defined functions. Although it is possible for ArrayUDF (in C++) to call MATLAB/Python code, this method has a significant overhead caused by communication, data transfer, and conversion across different systems/languages. Hence, we developed DasLib to enable HPC-friendly data analysis systems like ArrayUDF to perform large-scale DAS data analysis.

DasLib contains sequential codes, and these codes are thread-safe. DasLib can be parallelized by HAEE presented in the following subsection to run on HPC systems efficiently. Table II summarizes the most popular operations within DasLib functions for data analysis. We classify these functions into two categories: time-domain operations and frequency-domain operations. The time-domain library provides functions to be applied on the raw DAS data. A common operation is, for example, local similarity, which was originally used for processing large arrays of conventional spatially-discrete electronic sensors[18]. The frequency domain operations convert

DAS data into the frequency domain. Most commonly used functions include a fast Fourier transform (FFT) function and bandpass filter functions. Unless otherwise noted, the name and semantics of these functions follow the style of the signal processing toolbox in MATLAB [3].

TABLE II: List of sample functions from DasLib.

Functions	Semantic
Das_abscorr(c_1, c_2)	absolute correlation of c_1 and $c2$ defined as $\lvert\cos(\theta(c_1, c_2))\rvert$
Y = Das_detrend(X)	removes the best straight-line fit of x
(c_1, c_2) = Das_butter(n, f_c)	create Butterworth filter coefficients c_1 and c_2 with the cutoff frequency f_c
Y = Das_filtfilt(c_1, c_2, X)	apply c_1 and c_2 to X
Y = Das_resample(X, 1, R)	samples the X with new rate R
Y = Das_interp1(X_0, Y_0, X)	linearly interpolates f that satisfies $f(X_0) = Y_0$ to obtain the values Y at X
Y = Das_fft(X)	perform FFT on X
Y = Das_ifft(X)	perform inverse FFT on X

B. HAEE: Hybrid ArrayUDF Execution Engine

The existing ArrayUDF only supports MPI (i.e., process) based parallelization for data analysis operations. This parallelization works for most operations (like local-similarity) on DAS data. However, it may not be efficient for other DAS data analysis operations such as FFT based cross-correlation calculations. There are two major issues with using MPI based ArrayUDF here: 1) Using a purely MPI based parallelization method may cause memory footprint pressure. Being different from the local-similarity operation on a set of neighboring channels, the FFT based cross-correlation operation needs to compare a master channel with all other channels. Hence, by using MPI based parallelization, the master channel needs to be duplicated for each process. For a computing node with k CPU cores, the master channel needs to be replicated k times in this approach. 2) The number of I/O requests might be too large for the storage system to handle during purely MPI based parallelization. Each MPI process on each CPU core triggers their own I/O requests. Most of storage devices are bound by input/output operations per second (IOPS). Having large number of I/O requests can lead to long waiting queues and high contention frequency on storage disks [14], [12].

To address these issues, our work extends ArrayUDF to support the hybrid OpenMP and MPI model, i.e., hybrid thread and process model. Our goal is to have a single or fewer MPI process on a single computing node. Within the single node, multiple OpenMP threads are started to run the analysis code. Hence, all the CPU cores share the same data, e.g., the master channel. Meanwhile, each computing node only issues one I/O request for all its CPU cores, and therefore reduces the number of I/O requests. The core algorithm that enables the hybrid ArrayUDF is the multithreaded Apply (`ApplyMT`) function, shown in Algorithm 1. The hybrid ArrayUDF uses the same functions as the original ArrayUDF for I/O operations and in-memory data management [15]. `ApplyMT` accepts a linearized sub-block read by a single MPI process and a user-defined function. Then, the multithreaded Apply starts multiple threads to process the same data at the same time. To avoid conflict in accessing the output vector, `ApplyMT` creates a separate output vector per thread. Once the process has finished the entire dataset, these separate output vectors are merged in parallel too.

[3]https://www.mathworks.com/products/signal.html

Algorithm 1 Multithreaded Apply function in Hybrid ArrayUDF Execution Engine (Based on OpenMP)

Note: B is a 1D linearized sub-block ($m \times n$) read by a single MPI process (usually per computing node) from a 2D DAS data. f_p is the user-defined function. R is the result vector. The t is the number of threads. The h ($h = 0, \ldots, t-1$) is the index of thread.

```
function ApplyMT(B, f_p, R)
    #pragma omp parallel                          ▷ Start the parallel block
    {
        Vector<T> R_p                             ▷ Result vector per thread
        Vector<int> p(t)                          ▷ Prefix for t threads
        #pragma omp for schedule(static)          ▷ Split work
        for i = 0 : m × n − 1 do                  ▷ among threads
            s = CreateStencil(B[i])               ▷ Create Stencil on B[i]
            r = f_p(s)                            ▷ Run UDF f_p on s
            R_p.append(r)                         ▷ Append local result
        end for
        p[h]= R_p.size()                          ▷ Get local size for the hth thread
        #pragma omp barrier
        #pragma omp single                        ▷ Calculate displacement
        for i = 1 : t + 1 do
            prefix[i] += prefix[i - 1];
        end for
        R[p[h-1]: p[h]] = R_p                     ▷ Insert local to R
    }
end function
```

C. Case Study: HAEE in real DAS data analysis

Most DAS data analysis algoirthms are presently implemented in MATLAB or Python (e.g. ObsPy [5]). In this subsection, we demonstrate how to use HAEE to express two typical algorithms with our DASSA framework. Note that HAEE only requires users to specify a sequential analysis code as the user-defined function (UDF) and to provide some basic parallelization parameters (e.g., the number of MPI processes and OpenMP threads). Note that we focus on UDFs for two recent geophysical studies with high impact.

- **Earthquake detection via local similarity.** The local similarity method is a time-domain data analysis algorithm recently developed to detect earthquakes in array seismic datasets [18]. The user-defined function (UDF) namely `LocalSimi` for the local similarity calculation is presented in Algorithm 2. `LocalSimi` first extracts two time series segments (referred to as windows) at two neighboring channels via the `Stencil` abstraction of ArrayUDF. The `Das_abscorr` function from DasLib is then used to calculate the absolute correlation between the two windows. `ApplyMT` in HAEE then applies `LocalSimi` over all the window pairs in the DAS data in parallel.
- **Traffic-noise interferometry** uses a series of ambient-noise data analysis operations to generate empirical Green's functions used for imaging the shallow subsurface [2], [16]. The processing approach involves a long pipeline to convert

the raw DAS data (large in size) into shear-wave velocity profiles. Here we only build the most expensive collection of processes, referred to as ambient-noise interferometry. Ambient-noise interferometry converts raw data into a noise-correlation in the frequency domain, after applying a series of pre-processing and filtering operations. Algorithm 3 presents the user-defined function for the ambient-noise interferometry on a single channel. `ApplyMT` in HAEE then applies this UDF over all channels in parallel.

Algorithm 2 Local similarity calculation within HAEE as the user-defined function on DAS data.

Note: S is the Stencil abstraction representing an abstract cell and its neighborhood. Each window (e.g., W, W_1, and W_2) has width $(2M+1)$. Two neighboring channels have offsets $+K$ and $-K$ relative to the central channel, respectively. $(2L+1)$ windows are sampled on each neighboring channel (i.e., W_1 and W_2).

function LOCALSIMI(S)
  $W = S(-M:M, 0)$ ▷ Extract current window via S
  $C_{+K} = C_{-K} = 0$ ▷ initialization
  for $l = -L : L$ **do**
    $W_1 = S((l-M):(l+M), +K)$
    $W_2 = S((l-M):(l+M), -K)$
    $C_{+K} = \max\{C_{+K}, \text{Das\_abscorr}(W, W_1)\}$
    $C_{-K} = \max\{C_{-K}, \text{Das\_abscorr}(W, W_2)\}$
  end for
  return $\frac{1}{2}(C_{+K} + C_{-K})$ ▷ Local similarity.
end function

Algorithm 3 Traffic-noise interferometry within HAEE as the user-defined function on DAS data.

Note: S is the Stencil abstraction representing an abstract cell and its neighborhood. Each channel has a time series of length W. M_{fft} denotes the FFT transformed master channel for each process.

function TRAFFICNOISEUDF(S)
  $W_{n,0} = S(0:(W-1), 0)$ ▷ Time series per channel
  $W_{n,1} = \text{Das\_detrend}(W_{n,0})$
  $W_{n,2} = \text{Das\_filtfilt}(Das\_(n, fc), W_{n,1})$
  $W_{n,3} = \text{Das\_resample}(W_{n,2})$
  $W_{\text{fft}} = \text{Das\_fft}(W_{n,3})$
  return $\text{Das\_abscorr}(W_{\text{fft}}, M_{\text{fft}})$
end function

VI. EXPERIMENTAL RESULTS

We evaluate DASSA experimentally on the `Cori` supercomputer located in NERSC[4]. `Cori` is a Cray XC40 system with 2880 "Haswell" computing nodes. Each node has 32 CPU cores. Our tests use a DAS dataset recorded for two days by the experiment presented in Section II. The size of the whole DAS dataset is 1.9TB and it consists of 2880 files. Each file is a 2D array (11648 channels $\times$ 30000 time samples). We evaluated DASSA with different numbers of files and different data sizes to understand better how DASSA works with searching and scaling. These files are stored in a Lustre file system. Details of our tests are presented while we report the results. Unless specifically mentioned, we used Algorithm 3 as the workload driver of experiments.

[4]https://www.nersc.gov/users/computational-systems/cori/

A. Search and Merge

We first evaluate the search and merge function from DASSA with a single CPU core, i.e., single process. We use the 2880 files and select parts of these files to create either a real concatenated array (RCA) or a virtually concatenated array (VCA). The time taken by the search and merge function is reported in Figure 6. Both RCA and VCA contain the metadata and they use the same `das_search` tool to search on metadata. Hence, they have the same performance in finding specific files among 2880 files. The search time is at most 0.002 seconds. After searching, creating VCA took at most 0.01 seconds. In contrast, creating the RCA can consume up to 9978 seconds for 2880 files in our tests. On average, creating VCA is 70,000X faster than creating RCA. Searching data and creating VCA is fast because it only works on metadata. Creating RCA is slow because it needs to read the entire data and also write the data into a large array. Note that our DASSA supports both RCA and VCA but it is desirable to use VCA as the way to merge the searched files.

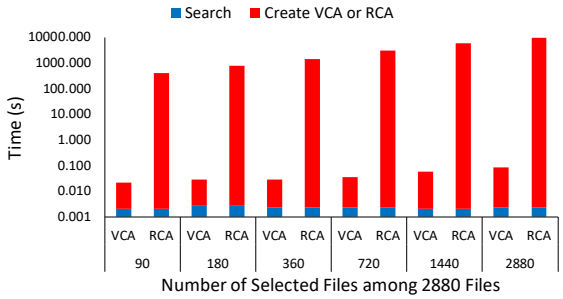


Fig. 6: Experimental results of searching and creating a RCA or a VCA with DASSA.

B. Read Data

As demonstrated in previous tests, it is efficient for users to use a VCA to merge DAS files for data analysis. However, accessing a VCA may have large overhead as it needs to access each file individually. We proposed a "communication-avoiding" method to reduce the I/O cost of accessing the VCA. Here, we compare the performance of "communication-avoiding" with the "collective-per-file" method. As a reference, we also include a test to access the RCA. The RCA creates a really merged file for both smallest size and largest size. This test uses 90 MPI processes to evenly partition and access the data. The timing results of these processes are reported in Figure 7. As expected, "communication-avoiding" is much faster (on average 37X) than "collective-per-file" in accessing data from lots of files behind the VCA. As its name states, the "communication-avoiding" approach can avoid lots of communications during access to the VCA, whereas the "collective-per-file" method needs a broadcast per file. The "collective-per-file" method is even more time-consuming than the RCA. But, our "communication-avoiding" is faster than the RCA. Hence, our proposed "communication-avoiding" method enables an efficient way of accessing data in the VCA.

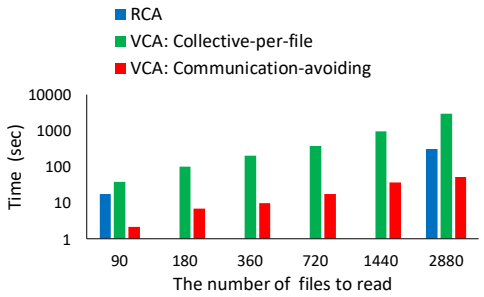


Fig. 7: Experimental results of reading DAS data from a VCA using "collective-per-file" and "communication-avoiding" methods. For reference, we also evaluate the time of accessing RCA (i.e., creating a really merged HDF5 file) on both the smallest case and the largest case. As shown in previous test, creating RCA is time consuming for all cases.

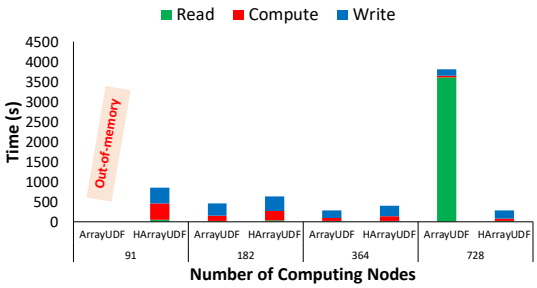


Fig. 8: Experimental results of evaluating MPI ArrayUDF and Hybrid ArrayUDF (marked as HArrayUDF) with different number of CPU cores.

C. HAEE: Hybrid ArrayUDF Execution Engine

This section reports results of comparing the hybrid ArrayUDF execution engine (HAEE) with the original ArrayUDF. Our tests fix the input data size (1.9TB) and evaluate total number of computing nodes from 91 to 728. Taking the case using 91 computing nodes as example, the original ArrayUDF runs 16 MPI processes per node. Our HAEE starts 1 MPI process per node and uses 16 threads per process. Figure 8 reports our experimental results. As we discussed before, DAS data analysis may duplicate data (i.e. the master channel) during cross-correlation analysis. Using the original ArrayUDF, we need to duplicate the master channel 16 times per node. This duplication makes the original ArrayUDF approach run out of memory in the case using 91 computing nodes. Our HAEE, however, can finish the analysis without any memory issues. As the scale increases, the original ArrayUDF shows certain performance benefits because of the coordination overhead of multiple threads in HAEE, as illustrated in Algorithm 1 . Nonetheless, when 728 computing nodes are reached, the I/O overhead (especially for the read) increases significantly. The main reason for this is that all 11648 processes on 728 computing nodes used by ArrayUDF issue I/O calls at the same time, and these I/O calls introduce access contention on the storage devices, the network, and the network interface [12]. In contrast, our HAEE issues 16X less I/O calls. Reducing the number of I/O calls at large scale is important to keep high performance on HPC systems [14]. HAEE and original ArrayUDF have the same performance in writing because they write the output as a single and big array. In summary, our hybrid ArrayUDF execution engine can perform DAS data analysis in both large-scale and small-scale tests cases where ArrayUDF fails or has worse performance.

D. Case Studies: scientific applications

Section V-C presents two case studies of applying DASSA to real DAS data analysis. This section reports the evaluation of the these data analysis with DASSA and MATLAB. The main reason behind choosing MATLAB for this comparison is that this is the platform used by the geophysicists to develop the DAS data analysis pipeline explored here. Because of limitations related to MATLAB license availability in our supercomputing center, we could only compare DASSA with MATLAB running on a single node. To fit the memory size of a single node, we use a single 1-minute file (≈700MB) as the input for our tests. We use the multi-thread feature (12 CPU cores) of both MATLAB and DASSA. Figure 9 reports the results of these tests. This comparison clearly shows that MATLAB is at most 16X slower than DASSA. Note that this test is only performed on a single node with a single file. As a result, both MATLAB and DASSA have similar performance in reading and writing. The Matlab codes used by geophysicists rely on its multi-thread feature to utilize multicores. It is difficult for the whole Matlab code pipeline to be parallelized. But, the DASSA actually parallelizes the entire code pipeline. Thus, the DASSA is much faster than Matlab in computing. On multiple nodes, MATLAB tends to slow down even more [25]. DASSA, on the contrary, has multiple novel methods to enable it to work well on hundreds of computing nodes. Results of a scalability tests of DASSA on more computing nodes are reported in the following subsection. Results of the application of DASSA for the detection of seismic signals using local similarity is shown in Figure 10. This analysis reveals clear signals caused by a distant earthquake and a series of vehicles as recorded in the 6-minute DAS record shown in Figure 1b.

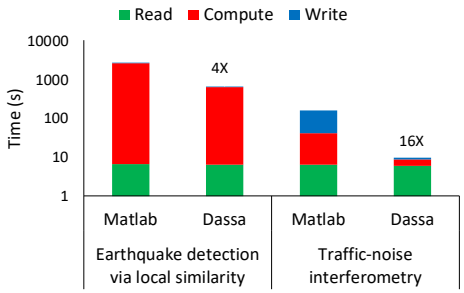


Fig. 9: Experimental results of comparing the same real DAS data analysis pipeline developed with DASSA and MATLAB.

E. Scaling Evaluations of DASSA

We test DASSA in both strong scaling and weak scaling settings. For the strong scaling case, we fixed the data size to 1.9TB. For the weak scaling case, we fixed the data size per

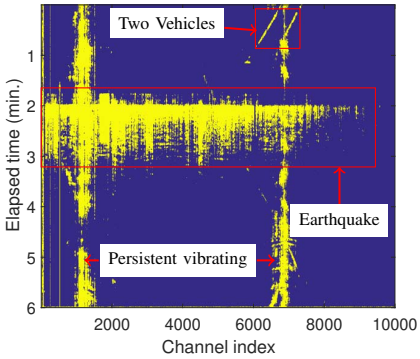


Fig. 10: Demonstration of results produced by DASSA to detect varying events with local similarity in Algorithm 2. It is possible to distinguish two moving vehicles and a M4.4 earthquake that occurred in Berkeley.

CPU core to 171MB. We increase the number of computing nodes from 91 to 1456, and start 8 threads per node, i.e., using 8 CPU cores per node. Results are shown in Figure 11. For display purposes, we normalize the test results to be shown in terms of parallel efficiency. The parallel efficiency for strong scaling is defined as $t_1/(N*t_N)*100\%$, 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 $t_1/t_N * 100\%$. Clearly, DASSA has perfect (≈100%) parallel efficiency in terms of computing time. However, the parallel efficiency for I/O operations show a downward trend. The main reason for this decay in efficiency is that increasing the number of computing nodes triggers more I/O requests, and these requests have more chances to have contention at Lustre file system and the network [14], [12]. The Cori supercomputer used in the evaluation has a fixed number of disk-based storage targets in its Lustre file system. Hence, as the number of nodes continues to increase, the parallel efficiency trends for I/O tends to decrease much further. The Burst Buffer-based storage system [12] has high IOPS than disk system. Hence, using the Burst Buffer addresses the down trend of the parallel efficiency for I/O. Based on current results, the case with 364 computing nodes gives the best efficiency.

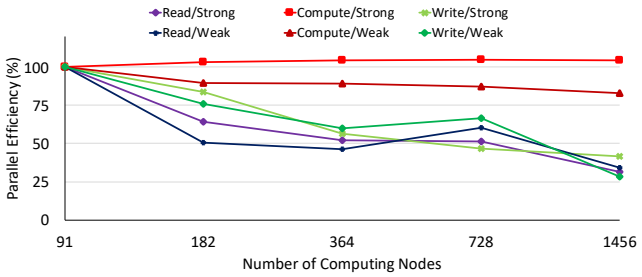


Fig. 11: Scaling experimental results of DASSA.

VII. Related Work

Traditional DAS data analysis are mostly performed using Digital signal processing (DSP) methods and hand-developed code in MATLAB [16], [18]. Since most of these pioneer works were dealing with small sets of data for conceptual evaluation, MATLAB with DSP libraries provided enough capability. Similar Python packages like ObsPy [5] also exist. Recently, there are the development of more advanced processing approaches, such as machine learning based DAS data analysis procedures. For example, work [7] compares classic and image based classification method for DAS events detection. Other studies use Generative Adversarial Network (GAN) to produce a large-scale dataset of tagged events for training ML models [26]. Most of these works use tools such as Theano [3] or Lasagne [11] originally designed for a single node. As stated in the background section, DAS may produce large amounts of data that are almost impossible to analyze using single node approaches. In these cases, the use of HPC systems for such large DAS data analysis is appealing, due to its capability in high floating-point operations per second (FLOPS). Inspired by the idea, we develop the new DASSA framework for DAS data analysis on HPC systems.

More generic large scale data analysis systems are already available. However, most of them lack efficient support for DAS data storage and analysis. For example, Relational DBMS [31], [22] and array DBMS [4], [20], [6] provide `tuple` and `array` based data model and abstract data analysis operations, such as `select` and `join`. Obviously, the `tuple` is not appropriate for DAS data, since it consists of a multidimensional array. Although the SciDB's array model would be able to handle the DAS data structure, it lacks the flexibility and efficient support needed for DAS data analysis operations, such as local similarity [18]. The local similarity shows high structural locality in semantic [15], [20], [32].

MapReduce[9] and its improved generation Spark [34] provide a simple `key-value (KV)` data model and two generic and user-definable operations, `map` and `reduce`. SciHadoop [8] adopts MapReduce to process data in array. However, because the `KV` data model is totally different to the array data model, MapReduce-based systems have high overhead when dealing with scientific data such as the DAS data in array structure. New systems, such as Cassandra [17] and Prometheus [1], have adjusted MapReduce-type system for time series data processing. Their built-in operators and user-defined function mechanism, however, lack the support for operators with structural locality, such as the DAS local similarity calculation shown in Algorithm 2.

ArrayUDF [15] addresses the issues inherited within MapReduce by supporting native array data model. It also provides user-definable and generic operators to support data analysis operations with structural locality. The problem with the existing ArrayUDF is that it only supports MPI model for parallel data analysis. In DAS data analysis, we found that this method may create lots of duplicated data across MPI ranks. Hence, we extended ArrayUDF to support a hybrid (MPI and

OpenMP) execution model for efficient execution. HDF5 [30] virtual dataset is similar to our Virtually Concatenated Array (VCA). The HDF5 virtual dataset, however, only supports sequential data access. Our VCA has the new communication-avoiding method to work efficiently in parallel on multiple computing nodes. The MPI collective I/O [33] shares certain similarities with our method, but it only works on a single and binary file. Our communication-avoiding method works on Virtually Concatenated Array (VCA) with array structure.

VIII. Conclusions

We study the methods and algorithms to enable efficient and productive DAS data analysis on HPC systems. The proposed DASSA framework provides efficient storage for large-scale DAS data. It also allows user-definable data analysis operations and runs these defined operations transparently on HPC in parallel. We proposed a communication-avoiding I/O access method to reduce the cost of accessing DAS data, and a hybrid execution model for user-defined operations with MPI and OpenMP. Experimental results demonstrate that DASSA can operate at most 16X faster than the existing DAS data analysis pipeline developed in platforms such as MATLAB. We show that the scalability of DASSA can reach up to 1456 computing nodes. Future work on DASSA includes an API in Python or even in MATLAB to enable interactive DAS data analysis. We also intend to study how to apply the DASSA in other applications, such as plasma simulation, which may store the data of each simulated domain as an individual file and lots of domains may be grouped as the input of analysis operations [12]. Moreover, how to automatically select system settings, such as the number of nodes, to run the analysis code is another topic we will explore in future.

Acknowledgment

This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) and Office of Energy Efficiency and Renewable Energy (EERE), Office of Technology Development, Geothermal Technologies Office, under contract number DE-AC02-05CH11231 with LBNL. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility.

References

[1] Prometheus. https://prometheus.io/. Accessed: 2019-12-22.

[2] J. Ajo-Franklin, S. Dou, T. Daley, B. Freifeld, M. Robertson, C. Ulrich, T. Wood, I. Eckblaw, N. Lindsey, E. Martin, et al. Time-lapse surface wave monitoring of permafrost thaw using distributed acoustic sensing and a permanent automated seismic source. In *SEG Technical Program Expanded Abstracts 2017*. Society of Exploration Geophysicists, 2017.

[3] R. Al-Rfou, G. Alain, et al. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, 2016.

[4] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. *SIGMOD Rec.*, 27(2):575–577, 1998.

[5] M. Beyreuther, R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann. ObsPy: A Python Toolbox for Seismology. *Seismological Research Letters*, 81(3):530–533, 05 2010.

[6] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.

[7] M. Bublin. Machine learning for distributed acoustic sensors, classic versus image and deep neural networks approach. *CoRR*, 2019.

[8] J. B. Buck, N. Watkins, and et al. SciHadoop: Array-based Query Processing in Hadoop. In *Supercomputing Conference (SC)*, 2011.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[10] J. Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 585–585, May 2013.

[11] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, et al. Lasagne: First release., Aug. 2015.

[12] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen. Data elevator: Low-contention data movement in hierarchical storage system. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161, Dec 2016.

[13] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *J. Parallel Distrib. Comput.*, 72(10):1254–1268, Oct. 2012.

[14] B. Dong, X. Li, L. Xiao, and L. Ruan. Towards minimizing disk i/o contention: A partitioned file assignment approach. *Future Generation Comp. Syst.*, 37:178–190, 2014.

[15] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. Arrayudf: User-defined scientific data analysis on arrays. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 53–64. ACM, 2017.

[16] S. Dou, N. Lindsey, A. M. Wagner, T. M. Daley, B. Freifeld, M. Robertson, J. Peterson, C. Ulrich, E. R. Martin, and J. B. Ajo-Franklin. Distributed acoustic sensing for seismic monitoring of the near surface: A traffic-noise interferometry case study. *Scientific reports*, 7(1), 2017.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[18] Z. Li, Z. Peng, D. Hollis, L. Zhu, and J. McClellan. High-resolution seismic event detection using local similarity for large-n arrays. *Scientific reports*, 8(1):1646, 2018.

[19] N. J. Lindsey, E. R. Martin, D. S. Dreger, B. Freifeld, S. Cole, S. R. James, B. L. Biondi, and J. B. Ajo-Franklin. Fiber-optic network observations of earthquake wavefields. *Geophysical Research Letters*, 44(23):11–792, 2017.

[20] A. P. Marathe and K. Salem. A Language for Manipulating Arrays. In *VLDB*, 1997.

[21] A. Mateeva, J. Lopez, H. Potters, J. Mestayer, B. Cox, D. Kiyashchenko, P. Wills, S. Grandi, K. Hornman, B. Kuvshinov, et al. Distributed acoustic sensing for reservoir monitoring with vertical seismic profiling. *Geophysical Prospecting*, 62(4):679–692, 2014.

[22] B. Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[23] T. Parker, S. Shatalin, and M. Farhadiroushan. Distributed acoustic sensing–a new tool for seismic applications. *First Break*, 32(2), 2014.

[24] R. Rabenseifner et al. Hybrid mpi and openmp parallel programming. In B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[25] V. Sachdeva. p2matlab: Productive parallel matlab for the exascale. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2109–2112, May 2011.

[26] L. Shiloh, A. Eyal, and R. Giryes. Deep learning approach for processing fiber-optic das seismic data. In *26th International Conference on Optical Fiber Sensors*, page ThE22. Optical Society of America, 2018.

[27] A. Shoshani and D. Rotem. *Scientific data management: challenges, technology, and deployment*. Chapman and Hall/CRC, 2009.

[28] H. Tang, S. Byna, et al. Usage pattern-driven dynamic data layout reorganization. In *CCGrid*, pages 356–365, May 2016.

[29] H. Tang, X. Zou, et al. Improving read performance with online access pattern analysis and prefetching. In *Euro-Par*, 2014.

[30] The HDF Group. HDF5 User Guide, 2010.

[31] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[32] X. Xing, B. Dong, J. Ajo-Franklin, and K. Wu. Automated parallel data processing engine with application to large-scale feature extraction. In *MLHPC 2018*, pages 37–46, Nov 2018.

[33] H. Yu, R. K. Sahoo, et al. High performance file I/O for the bluegene/l supercomputer. In *HPCA-12*, pages 187 – 196, February 2006.

[34] M. Zaharia, M. Chowdhury, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.