MsPASS: A Data Management and Processing Framework for Seismology

Yinzhi Wang*[1], Gary L. Pavlis[2], Weiming Yang[1], and Jinxin Ma[1]

Abstract

This article introduces a new framework for seismic data processing and management we call the Massive Parallel Analysis System for Seismologists (MsPASS). The framework was designed to enable new scientific frontiers in seismology by providing a means to more effectively utilize massively parallel computers to handle the increasingly large data volume available today. MsPASS leverages several existing technologies: (1) scalable parallel processing frameworks, (2) NoSQL database management system, and (3) containers. The system leans heavily on the widely used ObsPy toolkit. It automates many database operations and provides a mechanism to automatically save the processing history for reproducibility. The synthesis of these components can provide flexibility to adapt to a wide range of data processing workflows. We demonstrate the system with a basic data processing workflow applied to USArray data. Through extensive documentation and examples, we aim to make this system a sustainable, open-source framework for the community.

Cite this article as Wang, Y., G. L. Pavlis, W. Yang, and J. Ma (2021). MsPASS: A Data Management and Processing Framework for Seismology, *Seismol. Res. Lett.* **93**, 426–434, doi: 10.1785/0220210182.

Introduction

In the twentieth century, two branches of seismology were major forces in advancing information technology (Bates *et al.*, 1982). First, the seismic reflection method remains a major consumer of high-performance computing (HPC) cycles. As a result, there exist multiple, extremely advanced data handling systems for seismic reflection data. The second major force that drove our field until the 1990s was the underground nuclear testing problem that began with the Vela program in the 1960s (Bates *et al.*, 1982) until the main technical problems were largely solved by the mid-1990s. Much of the data processing infrastructure of earthquake seismology is a legacy of the nuclear monitoring program. Two notable examples are the Seismic Analysis Code (SAC, Goldstein *et al.*, 2003) and Antelope (a descendent of Datascope developed during the Incorporated Research Institutions for Seismology [IRIS] Joint Seismic Program). We are all aware of the incredible advances in information technology since the 1990s, but we assert seismology's exploitation of these advances has been irregular. Reflection seismic processing systems remain state of the art but also are rigid, being highly optimized for handling seismic reflection data. The societal importance of earthquake monitoring has also yielded multiple solutions for real-time earthquake monitoring and processing. Finally, thanks to the community resources of IRIS, the data archive problem in our field is largely solved, and any seismologist can quickly acquire far more data than they can handle. The system we introduce in this article was developed to fill a hole we assert exists in the current data handling infrastructure: a flexible but state-of-the-art system for handling research data that may not always match the assumed model for systems that handle seismic reflection data or bulletin preparation.

Our design was impacted by the recent emergence of applications of big data, cloud computing, and machine learning to earthquake seismology. Addair *et al.* (2014) first demonstrated the applicability of the Hadoop software framework to large-scale seismic data processing. Others have also adopted the MapReduce parallel programming model and applied it successfully to different domains of seismology (e.g., Mohammadzaheri *et al.*, 2013; Dodge and Walter, 2015; Chen *et al.*, 2016; Magana-Zook *et al.*, 2016; Junek *et al.*, 2017; Choubik *et al.*, 2020; Clements and Denolle, 2020). These previous articles demonstrate that the MapReduce concept, upon which our system is founded, is an efficient programming model for seismic data processing. The reason is that the parallelism is simple being naturally defined by atomic units equal to one or a group of seismic signals. A recent study by MacCarthy *et al.* (2020) shows that the

1. Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas, U.S.A., https://orcid.org/0000-0001-8505-0223 (YW); 2. Department of Earth & Atmospheric Sciences, Indiana University, Bloomington, Indiana, U.S.A.

*Corresponding author: iwang@tacc.utexas.edu

© Seismological Society of America

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

current data-intensive research in seismology can achieve an order of magnitude greater throughput than what the centralized data centers could provide when processing data natively on the cloud. This indicates that an efficient stream processing workflow is only possible with a capable data processing infrastructure as well as a large-scale seismic data sets available on the cloud. IRIS has an ongoing effort to migrate its data archive to the cloud, which may support faster download. The Southern California Earthquake Data Center is already hosting its earthquake catalog and seismic waveform archive on the cloud (Yu *et al.*, 2021). With the growing need of intensive data processing from the emerging field of seismology, such as Distributed Acoustic Sensing data processing and machine learning (e.g., Ajo-Kong *et al.*, 2018; Franklin *et al.*, 2019; Zhu and Beroza, 2019; Mousavi *et al.*, 2020), there will be more and more seismic data archive available on the cloud. The community demand for high-throughput data streaming in the HPC or cloud environments will also grow. The system we introduce in this article can serve as a framework for this evolution in the future.

A foundational assertion of this article is that the data processing infrastructure for research seismology is currently analogous to rusty bridges on U.S. highways. One of us (Pavlis) was involved with IRIS committees that conducted surveys and found SAC was the most common tool used by members of our community. The source code for SAC is FORTRAN, and the package was first written in the late 1970s and early 1980s. It should thus be no surprise that SAC has not yet adopted some of the modern concepts that would make it more efficient at processing large data sets. SAC is an easy target, but similar critiques can be made for most, if not, all general-purpose tools for handling earthquake data. Most of us have become accustomed to developing cumbersome, specialized workflows assembled from various custom software mixed with one or more packages such as SAC.

There have been other attempts to address this problem in recent years with various levels of success (e.g., Morozov and Pavlis, 2011; Eagar and Fouch, 2012; West and Fouch, 2012; MacCarthy and Rowe, 2014; Heimann *et al.*, 2017). Of particular importance to this article is the development of the ObsPy package that is widely accepted by the community as a standardized seismic data processing tool (Beyreuther *et al.*, 2010). ObsPy is important to this article because the system we describe here (Massive Parallel Analysis System for Seismologists [MsPASS]) leans heavily on it.

MsPASS is a framework that was designed to help move the community forward by addressing the following key problems:

1. It is universally acknowledged that single processor computers reached their operational limit in early 2000s (Geer, 2005). All advances in "supercomputers" have utilized some form of parallel processing. A generic solution to parallel processing is thus a primary goal of MsPASS.
2. For a long list of reasons, the volume of digital data available to all seismologists has become overwhelming. Anyone aiming to assemble a large data set to address a particular research question will nearly always run into a data management problem. For this reason, we designed MsPASS with an integrated database system. We were also aware, however, of the serious issues of the complexity of all common relational database systems. Few research groups can afford to hire a database manager to maintain industrial-strength database software. We address this in MsPASS using what we have found to be a more flexible and scalable system for data management during processing called a document database. The specific open-source implementation that we integrated is called MongoDB.
3. As the complexity of institutional information technology infrastructure has grown, all of us have experienced the frustration in issues in dealing with that infrastructure. Examples include security concerns of installing software packages, dealing with conflicting package libraries, and simply installing a package on an HPC system. To address these issues, MsPASS is intended to mainly run in the containers. Containerization dramatically reduces the complexity of installing packages such as the database manager (MongoDB) and parallel schedulers (Spark and Dask) as well as the MsPASS package itself.

This article introduces key concepts of MsPASS for the seismology community. MsPASS is a framework we hope the community will embrace and help expand the number of algorithms available. If it grows as we hope, this article should also provide a citable source for papers utilizing the package.

Implementation

The design of MsPASS is illustrated with a simple workflow of filter and stack in Figure 1. The major components include the core implementation in Python and C++, a data management system, and a parallel scheduler. These components are encapsulated into a container for distribution. We will discuss each of the components in detail in this article.

Command language

MsPASS uses Python as the command language. Important reasons for that choice are:

1. Core packages in our design were Spark, Dask, and MongoDB. All have well-tested Python libraries.
2. The system was designed to extend ObsPy for parallel computing. ObsPy is a Python package.
3. Python is rapidly becoming a core component of scientific computing in many fields. That has two important corollaries: (1) it provides a mechanism for research code extension using

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

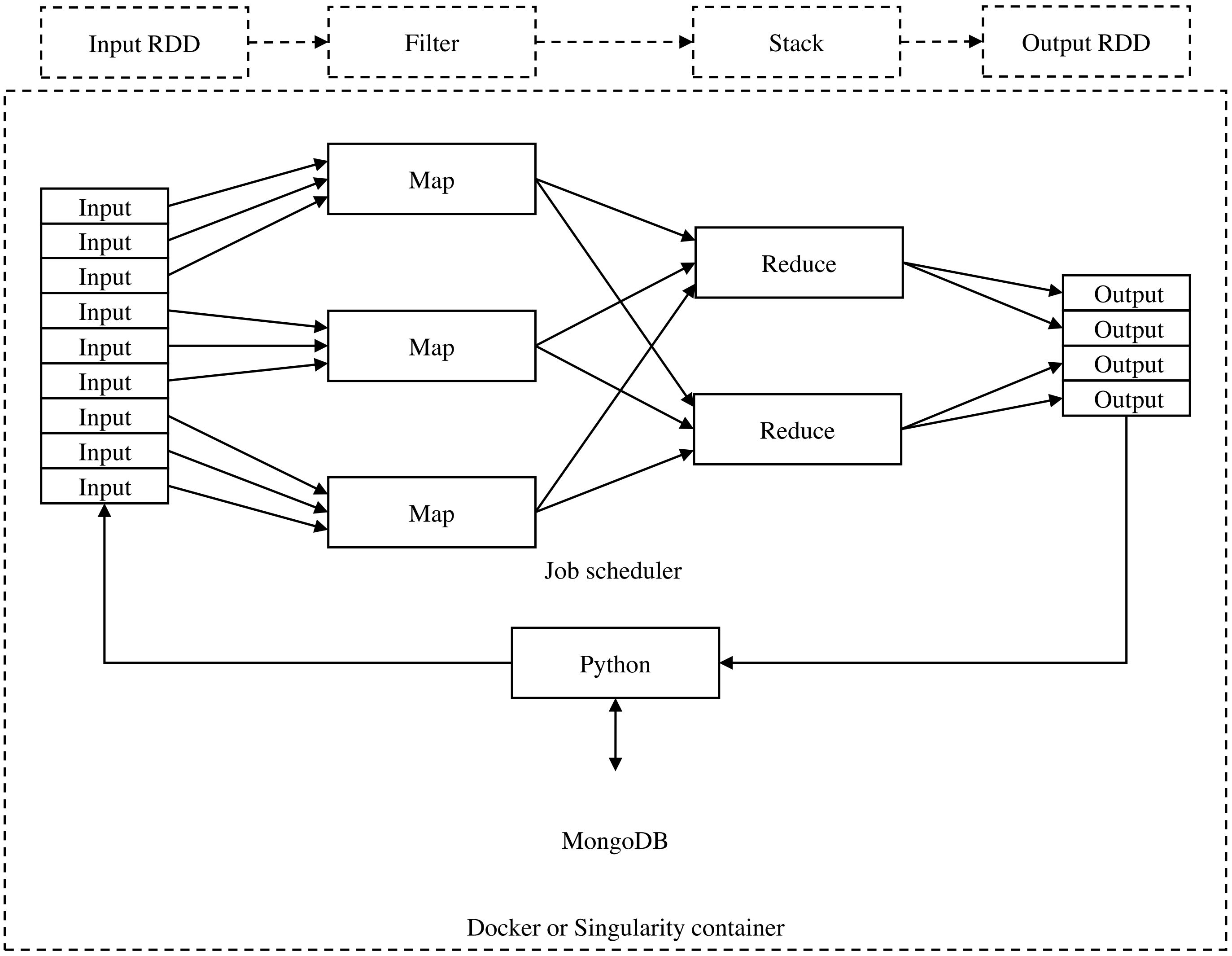

Figure 1. The Massive Parallel Analysis System for Seismologists (MsPASS) architecture illustrated in an example processing workflow. The top is the user abstraction of data flow, and the bottom is the actual execution graph. A Python script drives the workflow by selecting data from MongoDB and sending all the inputs to the map operation. Each map filters the local copy of data and creates outputs for the next step. The reduce step read either locally or remotely from the previous step, apply stacking, and produce outputs as intermediate data for either storing in the database or the next iteration. All components of MsPASS are containerized.

the rich array of Python libraries in existence, and (2) it likely means the language will be around for the foreseeable future.

Data containers

Python is an object-oriented programming language. In Python, all data, including simple types like integers and floats, are treated as classes. On the other hand, a major weakness of Python for numerical applications is that it is an interpreted language, which can cause serious performance problems if not handled properly. ObsPy works around this problem by making extensive use of NumPy for any of its compute-intensive algorithms. NumPy libraries are compiled C code so they can operate efficiently on large data arrays that are the norm for handling seismic data. They are linked to the Python interpreter through binding code that provides a mechanism to call C functions from Python. In MsPASS, we went a step beyond the ObsPy approach and built a core set of containers written in C++ with Python bindings constructed through a package called pybind11 (Jakob *et al.*, 2017). These containers are abstractions of four primary concepts we suggest define all seismic data:

1. We use the name *TimeSeries* to refer to a single channel of data. For ObsPy users, a *TimeSeries* can be viewed as an alternative implementation of an ObsPy *Trace* object.
2. A *Seismogram* in MsPASS means a bundled set of three-component data. The *Seismogram* object has native methods that naturally treat the data as vectors. This mostly

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

means methods for handling transformation matrices automatically.
3. A *TimeSeriesEnsemble* refers to a group of *TimeSeries* objects that have some generic relationship. This object can be thought of as a generalization of the concepts of various times of "gathers" in seismic reflection processing (i.e., shot gather, common midpoint gather, etc.).
4. A *SeismogramEnsemble* is exactly like a *TimeSeriesEnsemble* except the members of the group are all *Seismogram* objects.

The storage of the sample data is conventional and is notable only for the way the Python bindings allow the data to be manipulated with NumPy and SciPy libraries similar to ObsPy. The piece we call *Metadata* matches the modern definition of that word; metadata summarizes basic information about data. For many users, our *Metadata* class is best thought of as a generalized header. Any data can be inserted into a *Metadata* container and accessed with a key string. Our implementation is a rewrite of a C++ class written by one of us (Pavlis) distributed through Antelope's contrib library (see Data and Resources) and used as a component of dbxcor (Pavlis and Vernon, 2010), generalized iterative deconvolution (Wang and Pavlis, 2016), and the plane-wave migration package (Pavlis, 2011). For Python programmers, the *Metadata* container is perhaps best thought of as a C++ implementation of a Python dictionary.

ErrorLogger and *ProcessingHistory* are extensions to the core data objects that implement two completely different concepts. *ErrorLogger* is used to solve a fundamental problem in handling error conditions in a parallel environment. When data processing is distributed between multiple compute nodes, a mechanism is needed to centralize error messages. In MsPASS, we solve this problem using an *ErrorLogger* class that is a container that is part of the data and holds messages of varying levels of severity. Database saves operations, which are described later, automatically save all error messages to the database with links to the data to which they are associated. Because the error log lives with the data with which it is associated, a workflow can easily filter out data that have errors of any specified level of severity through database queries.

ProcessingHistory is an optional but novel component of MsPASS. The purpose of *ProcessingHistory* is to promote reproducible science. Our goal was to build a system that would assist in the publication of a workflow that would allow the reader of a seismology article to readily reproduce the same processed waveform data used as a basis for that article. *ProcessingHistory* is an implementation of what we call object-level history. Its purpose is to preserve the original parents of each processed waveform and the chain of algorithms they were passed through to produce the final result. History in MsPASS can always be preserved as a general tree. The leaves of the tree are the parent data objects, the nodes are defined by individual algorithms, and the base trunk is the final processed waveform object. The object-level history tree is, in fact, a representation of directed acyclic graph (Daniel, 2019) through which the final result was passed. Our implementation, however, is different from algorithms commonly found in textbooks for storing general trees. The reason is that tree structures are commonly handled efficiently by linked lists of pointers. Pointers turn out to be a big problem in parallel systems because the data get moved between nodes by schedulers (i.e., Spark or Dask), and resolving the pointers would be cumbersome at best. We solved this problem by saving the trees in a C++ multimap container keyed by a UUID (Universal Unique IDentifier). That approach allows the data defining a tree to be stored in a flat data structure that is easily serialized.

Data management system

Arguably, the most novel component of MsPASS is the way we handle data management. Relational databases have a long history in seismology. The original idea of using relational databases happened when relational databases were the hot new idea in information technology. The CSS3.0 schema was first drafted in 1990 and remains a widely used standard (Anderson *et al.*, 1990). Relational databases are now core elements of archival systems in all seismology data centers.

Although relational databases are workhorses everywhere, we found from experience that they are problematic in a data processing system for a long list of reasons. We will not inflict the reader with our list of reasons but state we found a solution in the NoSQL database system called MongoDB. MongoDB provides the following functionality that proved critical in implementing MsPASS.

1. The data model for MongoDB maps exactly into our Metadata container. In fact, the Python API normally uses a variation of a dictionary, which they call a document, as the abstraction of the equivalent of a relational database tuple.
2. One of the biggest problems in a research environment for a relational database is that the schema is a rigid thing that imposes constraints on extensions. Our experience is that with a relational database, most new algorithms require a new relation (table) to save any metadata generated by that algorithm. (see, e.g., the implementation by Pavlis and Vernon, 2010). MongoDB largely eliminates this problem. MongoDB is completely agnostic about what is saved in an individual tuple (document). A simple way to say it is this if it can be stored in a Python dictionary container, then it can be put into the MongoDB database. The caveat is that MongoDB supports only a limited set of native types, but we have not found a case we could not work around this restriction. For example, we store ObsPy's *Inventory* and *Catalog* objects as by serializing them and storing them as byte strings.
3. A database engine can easily become the bottleneck in a large processing workflow. Database transactions are nearly

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

always an expensive operation in time. Blocking IO to a database server is almost guaranteed to be the bottleneck in any processing workflow. That is especially true if the database server is isolated to a single processing node or, worse yet, is single-threaded like the Antelope database. MongoDB can leverage sharding (horizontal scaling) that provides a mechanism for distributing data in multiple nodes of a cluster to improve throughput.

4. MongoDB has an integrated, internal file storage system they call GridFS (see Data and Resources). Any reader who has handled a large data set with millions of data files will understand firsthand that all file systems can become a bottleneck in performance when the number of files gets too large, especially on HPC clusters with huge disk arrays. Handling files on node local storage is usually the solution to overcome this issue, but the limited space of local storage makes the data distribution and collection difficult with large datasets. GridFS could be the alternative solution, which can leverage sharding to forge the local storages of multiple nodes into a larger filesystem. The data input/output implemented in MsPASS is flexible such that we can also swap in more powerful transient shared object store implementation (e.g., Zhang *et al.*, 2018) when needed.

Parallel scheduling

Parallel is one of the keywords in MsPASS. Because of the fundamental role of parallelism in pushing the current generation of HPC, there is a large range of solutions to handling parallelism. In MsPASS, we implement parallelism through either Spark (Zaharia *et al.*, 2016) or Dask (Dask Development Team, 2016). Dask and Spark are similar with different strengths and weaknesses (see Data and Resources). Both provide the functionality of schedulers that distribute common computations across multiple processors and/or nodes.

Dask and Spark are sophisticated packages with lengthy options described in the documentation. For most components of MsPASS, however, the use amounts to one of two concepts we illustrate here with a simple example using pseudocode. Suppose we have a data set consisting of N data objects (*TimeSeries*, *Seismogram*, or one of the *Ensemble* types). Suppose also we have an algorithm *myapplication* that is expressed as a function that takes as input one data object, d, and emits another object known to MsPASS. A serial form of a simple workflow using *myapplication* could be expressed in pseudocode as follows:

```
1. for data in reader():
2. output = myapplication(data)
3. writer(output)
4. end for
```

This is a standard data driven algorithm in which the reader reads one object at a time until it runs out of data. The loop handles calling *myapplication* and saving the results one by one.

A key abstraction in parallel schedulers is to treat the entire data set as a thing instead of immediately parsing it into pieces. With this idea in mind, let the symbol d represent the entire data set of N data objects. A parallel version of the above using a simplification of the MsPASS API is the following:

```
1. d = read_distributed_data(db, query)
2. d = d.map(myapplication)
3. d = d.map(db.save_data)
4. d.compute()
```

In this snippet, *read_distributed_data* is an MsPASS function that creates the dataset container using a handle to MongoDB (*db*) and a query that uniquely defines that dataset. The map method of the dataset container applies a single function to each object that defines the data set. In our case, this abstraction is used twice: once to run *myapplication* and then to call the MsPASS *save_data* method of the MongoDB handle.

Line 4 defines a key concept of asynchronous parallelism in systems like Dask and Spark. They all use "lazy computation" meaning processing does not begin until requested explicitly or implicitly. Once started, the scheduler will decide which processor runs each of steps 1–3 on each datum of the dataset.

Containerization

Containers have emerged as a central component for running applications in a cloud environment. The problem containers aim to solve is insulating application software from any operating system dependency. Modern container hosts like those we use in MsPASS make it possible to build a system that can run on any computer of compatible hardware architecture. The practical implication of that is that MsPASS can run on all machines regardless of their operating systems. It also makes porting the system to HPC and cloud systems much simpler than building the entire software stack from source code. For this reason, container images are the recommended distribution mechanism for MsPASS. The container image has prebuilt, working version of key elements of MsPASS: the schedulers (Dask and Spark), the MongoDB database server and client, the MsPASS code base, and a Jupyter notebook running as the frontend. The MsPASS container image is hosted on the Docker hub (see Data and Resources), and the container is built to support two container hosts: Docker (Merkel, 2014) and Singularity (Gannon and Sochat, 2017). Docker is the recommended form for desktop use. Singularity is the most commonly available containerization tool on HPC systems. The container image can also serve as the base image to build any other packages to extend MsPASS (e.g., Wang *et al.*, 2019) or add completely independent packages needed to run a given workflow. Common examples are SAC and the Generic Mapping Toolbox (Wessel *et al.*, 2019).

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

The runtime environment for the container is set up by what is commonly called an "entrypoint script." For desktop use and prototyping, we recommend what we call the all-in-one mode using docker. The entrypoint script for this mode sets up MongoDB, a Spark or Dask local cluster, and a Jupyter notebook. This configuration is "all in one" because all the functions of the system are contained in a single container instance. On a single workstation, the workload is distributed into multiple cores automatically. The container runs a flavor of UNIX, so we use environment variables to define whether a container should run in one of five roles: database manager, shard (distributed database component), scheduler, worker, and/or frontend. On a distributed system like any of the HPC centers, these various roles need to be coordinated to leverage the performance by balancing the load requirements of the different roles. Users can deploy an arbitrary number of database shards and Dask or Spark workers tailored to the data processing need by only changing the environment variables. In the official GitHub repository of MsPASS, we provide exemplary job scripts to launch distributed MsPASS on HPC systems as well as docker compose files to launch it in a cloud environment.

Demonstrative Workflow

To demonstrate how MsPASS performs in a distributed environment, we downloaded a USArray data set. The data set was defined by teleseismic events recorded by USArray in 2012 and limited to stations that had a *P*-wave pick made by the Array Network Facility. The waveform segments are one-hour records with the event origin time as the start time.

For the initial performance tests presented here, we arbitrarily reduced the data to the first 30,000 records, yielding a modest data set of only ~5.34 GB. The test results we present here was designed primarily to appraise performance scaling with the number of processors dedicated to processing. The workflow contains four steps: (1) read the waveform data, (2) apply a demean operator, (3) band-pass filter between 0.01 and 2.0 Hz, and (4) window the waveform to −20 s and 150 s around the *P*-wave arrival time. This is a common initial processing step for most arrival-driven processing but is intentionally a lightweight calculation because of the expected scaling of this system. The following model provides a background and may help the reader better understand the limitations of parallelization with a scheduler such as Spark and Dask.

For a serial job, the time spent to process a single waveform with our test workflow is a simple sum of two terms:

$$T_{\text{serial}} = T_{\text{read}} + T_{\text{process}}, \tag{1}$$

in which T_{read} is the time spent in reading a given waveform into memory and T_{process} is the time spent in the calculations. (If we did a typical save at the end of the workflow, we could represent that by yet another additive term.). Because T_{serial} is an average per waveform process time, the time to process N waveforms is approximately NT_{serial}. For a parallel job, the elapsed time will scale per waveform as follows:

$$T_p = T_{\text{read}'} + T_{\text{process}'} + T_{\text{scheduler}} + T_{\text{serialization}}, \tag{2}$$

in which T_p is an average time to process a single waveform. Overhead is created by two terms here characterized by $T_{\text{scheduler}}$ and $T_{\text{serialization}}$. The first is processor time spent by the scheduler (Spark or Dask) in deciding what should be done when. The second is the average time spent moving data between nodes by serializing and reassembling the data objects. We assume $T_{\text{read}'}$ and $T_{\text{process}'}$ do not differ significantly from the serial version. Furthermore, for a summary, we combine the scheduling and serialization terms as $T_{\text{overhead}} = T_{\text{scheduler}} + T_{\text{serialization}}$ yielding the following scaling ratio for serial to parallel jobs:

$$T_p/T_{\text{serial}} = 1 + T_{\text{overhead}}/T_{\text{serial}}. \tag{3}$$

This shows that if the overhead is zero, this ratio will be one, and the elapsed time for a parallel job will be $1/N$ of the total elapsed time for a serial job. Any overhead will increase the lapsed time. This simple model also shows that to evaluate overhead we need to minimize T_{read} and T_{process}. This is why our test workflow does only minimal calculations.

Minimizing T_{read} is a different challenge. MongoDB provides an internal mechanism for storing raw data outside a file system using a mechanism they call GridFS. MsPASS supports GridFS natively, and saving data to GridFS is, in fact, the default input/output mode, but GridFS is not ideal for this overhead test. The reason is that we also wanted to limit the test to a single instance of the MongoDB server to avoid the additional potential complexity of sharding the database across multiple nodes. Initial tests showed that a single instance of the MongoDB would be a bottleneck and increase the T_{read} term significantly. Hence, for the tests shown here, we prestaged the waveform data to local storage of the compute nodes. This functionality is built in to the entrypoint script.

We ran this simple workflow on the Stampede2 supercomputer at the Texas Advanced Computing Center. Stampede2 hosts 4200 Knights Landing nodes and 1736 Intel Xeon Skylake (SKX) nodes. We chose to use the SKX nodes to run all the tests to achieve better performance. Each of the SKX nodes has two Intel Xeon Platinum 8160 processors and a total of 192 GB DDR4 memory, so this is unquestionably a test with a high-end system. We scaled the workflow from a single worker to 384 workers distributed across eight nodes. For this system, more than one node is used when the worker number goes beyond 48, which is the number of cores on a single node. We ran the tests using both the Dask and Spark scheduler to appraise their relative performance. We also

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

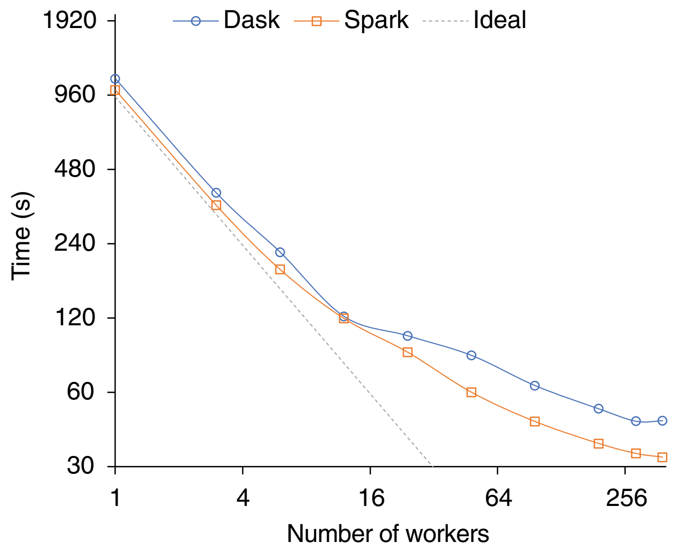

Figure 2. Parallel runtime scaling relations for overhead test. The *y* axis is the elapsed time for a job in seconds, and the *x* axis is the number of workers. The gray dashed line is the ideal scaling performance computed from a single-threaded run with a for loop. Note that both axes are logarithmic (ticks are at power of 2 intervals). The color version of this figure is available only in the electronic edition.

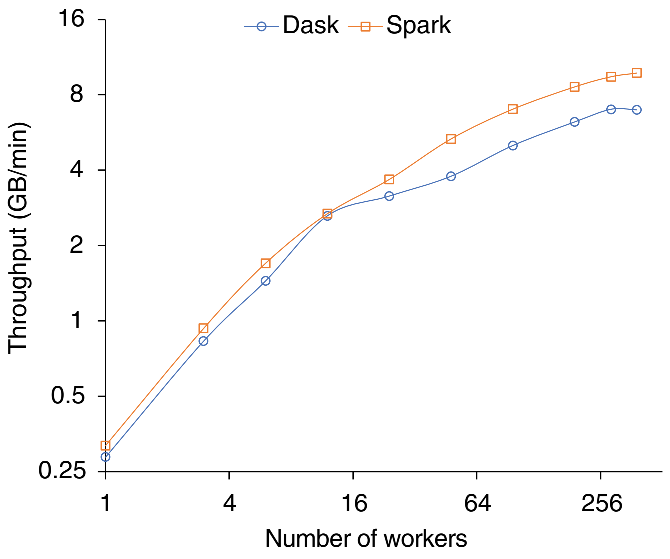

Figure 3. A comparison of average throughput between Dask and Spark. The *y* axis is the throughput of data measured by the size of the original waveform, and the *x* axis is the number of workers, in which multiple nodes are used for runs of more than 48 workers. Note that both axes are logarithmic (ticks are at power of 2 intervals). The color version of this figure is available only in the electronic edition.

run a single-threaded for-loop version of the workflow to estimate what we called T_{serial} above. The ideal performance computed as T_{serial}/N is illustrated in Figure 2.

Figure 2 shows the time to completion of the scaling runs with different numbers of workers. The plot shows that both Dask and Spark maintained near-ideal scaling for up to 24 workers. After that, the performance degrades. This indicates the overhead term increases with the number of workers. Although performance tuning is not the focus here, we attribute the performance degradation to two problems: (1) the scheduler performance is likely degrading with the number of processors it needs to handle for the same reason a human manager gets frantically busy as the person's number of people increases, and (2) for this system, passing over 48 cores likely increases the fraction of the data that must be serialized to move between nodes. Previous study showed that longer tasks or proper tuning can effectively mitigate such overheads (e.g., Dugré *et al.*, 2019; Böhm and Beránek, 2020). An additional insight on the scheduler overhead can be made by comparing processing of these data with a simple for loop versus a single worker run through Dask and Spark. We found the single-worker runs of Dask and Spark are ~18% and ~6% (respectively) slower than the for-loop version. The performance difference between Dask and Spark is within 15% with less than 24 workers. With more workers, Spark's performance is then significantly better than Dask's by about 30%.

A similar comparison can be made for the throughput of the workflow achieved by Dask and Spark (Fig. 3). The throughput here is measured as the total size of waveform data being processed over the processing time. This is an approximation because we did not take into account the size of the metadata. For these data (one-hour time blocks with 14,400 samples), the metadata size is negligible, so it is a close approximation that slightly underestimates throughput. Figure 3, similar to Figure 2, demonstrates higher throughput with Spark compared with Dask. More important, it demonstrates that this preliminary implementation of MsPASS can scale the throughput from ~0.3 GB/min to ~9 GB/min. A factor of 30 performance gain is not necessarily useful if one is processing a small data set. No one will care if a job takes 1 s or 1/30 s to complete. However, for a 1 TB data set, this translates to 2 min compared with approximately an hour for a serial job. For a more compute-intensive job, it is a difference between 6 hr and one week. Note, however, that for a more compute-intensive job, our simple model predicts the performance will improve as the fraction of time spent on overhead relative to computing time drops. Ongoing tests we plan to post with the source code repository documentation will evaluate that hypothesis. For this article, the key lesson is that this system works, and performance is solid, but we can likely improve performance with future tuning.

Discussion and Conclusions

An important issue with the current data infrastructure in our field is how to best handle the raw data input to a workflow. That is, should one always download data first and then do

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

processing or depend on webservices to only pull the data you need by request? Previous studies show that the streaming throughput from the IRIS Data Management Center (DMC) is ~1.7 GB/min. That rate is achieved with 100 nodes, and it is already pushing the limit of the connection rate control mechanisms of IRIS DMC (MacCarthy *et al.*, 2020). The throughput we achieved with MsPASS in our test is almost an order of magnitude greater with only a small number of nodes. This test, however, is not definitive for two reasons. First, the workflow in our test is very simple, and the total compute time per object is small (~0.03 s per channel). A more compute intensive workflow would drop throughput proportionally with the total compute time. Second, we ran these tests on an HPC cluster, in which each node has a larger number of cores and each core is as fast as any existing processor. Furthermore, the experiment in this study was done reading data from local storage to minimize input/output overhead from input/output. Hence, the throughput achieved should be close to an optimal setup for MsPASS alone. Therefore, although the database implementation of MsPASS supports stream processing waveform data with URLs, the download-and-process model is advised for now. We expect MsPASS to reach its full potential when cloud-based archives and the use of object stores like that described by Yu *et al.* (2021) become widely available in the near future.

A final point worth emphasizing is that although the core components of MsPASS are fully functional, there are two fundamental limitations that we must acknowledge.

1. MsPASS has a set of basic processing modules that include most ObsPy algorithms, a set of functions we implemented from existing C/C++ code, and a fairly complete suite of receiver function deconvolution algorithms. The system is far from a complete suite of methods matching even something like SAC. The system, however, is fully open source, and our hope is the community will add its algorithms to the system and it will grow naturally in functionality with time.
2. A system like MsPASS has the computing equivalent of a lot of moving parts. For a desktop system, the complexity is reduced, but when porting a workflow to a cluster, our experience is that some degree of tuning may be needed to handle large data sets. A strength of the containerized approach, however, is that the most important balancing seems possible by changing of the setup scripts that define which container runs in what mode.

The MsPASS source code and documentation are made available through GitHub (links are available in Data and Resources). We encourage any users to contribute to the work through the GitHub repo. We hope the open-source MsPASS package to become a useful tool to help seismologists leverage advanced computing infrastructures in the data processing workflows.

Data and Resources

The Massive Parallel Analysis System for Seismologists (MsPASS) source code is available at https://github.com/mspass-team/mspass, and the containerized distribution can be found in https://hub.docker.com/r/mspass/mspass. The USArray data used in this work was obtained from the Incorporated Research Institutions for Seismology Data Management Center (IRIS-DMC) available at www.iris.edu. The event catalog and arrival picks were obtained from the Array Network Facility at http://anf.ucsd.edu/tools/events/. The Antelope's contrib library is available at https://github.com/antelopeusersgroup/antelope_contrib. The information about GridFS is available at https://docs.mongodb.com/manual/core/gridfs. Comparison with Spark is available at https://docs.dask.org/en/latest/spark.html. The documentation of MsPASS is available at https://www.mspass.org. All other software resources used in this article came from published sources listed in the references. All websites were last accessed in July 2021.

Declaration of Competing Interests

The authors acknowledge there are no conflicts of interest recorded.

Acknowledgments

The authors thank Zhao Zhang and Songqiao Wei for discussions at the early stage of the project. The authors also thank an anonymous reviewer for the very helpful comments. This work is supported by the National Science Foundation (NSF) through the OAC-1931352 award.

References

Addair, T. G., D. A. Dodge, W. R. Walter, and S. D. Ruppert (2014). Large-scale seismic signal analysis with Hadoop, *Comput. Geosci.* **66,** 145–154, doi: 10.1016/j.cageo.2014.01.014.

Ajo-Franklin, J. B., S. Dou, N. J. Lindsey, I. Monga, C. Tracy, M. Robertson, V. R. Tribaldos, C. Ulrich, B. Freifeld, T. Daley, *et al.* (2019). Distributed acoustic sensing using dark fiber for near-surface characterization and broadband seismic event detection, *Sci. Rep.* **9,** no. 1, Article Number: 1328, doi: 10.1038/s41598-018-36675-8.

Anderson, J., W. Farrell, K. Garcia, J. Given, and H. Swanger (1990). Center for seismic studies version 3 database: Schema reference manual, *Center for Seismic Studies Tech. Rep., C90-1*, available at https://l2a.ucsd.edu/local/Manuals/CSS3.0_Format_Manual.pdf (last accessed July 2021).

Bates, C. C., T. F. Gaskell, and R. B. Rice (1982). *Geophysics in the Affairs of Man: A Personalized History of Exploration Geophysics and its Allied Sciences of Seismology and Oceanography*, First Ed., Pergamon Press, Oxford, New York.

Beyreuther, M., R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann (2010). ObsPy: A Python toolbox for seismology, *Seismol. Res. Lett.* **81,** no. 3, 530–533, doi: 10.1785/gssrl.81.3.530.

Böhm, S., and J. Beránek (2020). Runtime vs scheduler: Analyzing Dask's overheads, *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 1–8, doi: 10.1109/WORKS51914.2020.00006.

Chen, P., N. J. Taylor, K. G. Dueker, I. S. Keifer, A. K. Wilson, C. L. McGuffy, C. G. Novitsky, A. J. Spears, and W. S. Holbrook (2016). pSIN: A scalable, Parallel algorithm for Seismic INterferometry of large-N ambient-noise data, *Comput. Geosci.***93,** 88–95, doi: 10.1016/j.cageo.2016.05.003.

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user

Choubik, Y., A. Mahmoudi, M. M. Himmi, and L. El Moudnib (2020). STA/LTA trigger algorithm implementation on a seismological dataset using Hadoop MapReduce, *IAES Int. J. Artif. Intell. (IJ-AI)* **9,** no. 2, 269–275, doi: 10.11591/ijai.v9.i2.pp269-275.

Clements, T., and M. A. Denolle (2020). SeisNoise.jl: Ambient seismic noise cross correlation on the CPU and GPU in Julia, *Seismol. Res. Lett.* **92,** no. 1, 517–527, doi: 10.1785/0220200192.

Daniel, J. (2019). *Data Science with Python and Dask*, First Ed., Manning Publications, Shelter Island, New York, 296 pp.

Dask Development Team (2016). Dask: Library for dynamic task scheduling, available at https://dask.org (last accessed July 2021).

Dodge, D. A., and W. R. Walter (2015). Initial global seismic cross-correlation results: Implications for empirical signal detectors, *Bull. Seismol. Soc. Am.* **105,** no. 1, 240–256, doi: 10.1785/0120140166.

Dugré, M., V. Hayot-Sasson, and T. Glatard (2019). A performance comparison of Dask and apache spark for data-intensive neuroimaging pipelines, *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 40–49, doi: WORKS49585.2019.00010.

Eagar, K. C., and M. J. Fouch (2012). FuncLab: A MATLAB interactive toolbox for handling receiver function datasets, *Seismol. Res. Lett.* **83,** no. 3, 596–603, doi: 10.1785/gssrl.83.3.596.

Gannon, D., and V. Sochat (2017). Singularity: A container system for HPC applications, available at https://cloud4scieng.org/singularity-a-container-system-for-hpc-applications/ (last accessed February 2019).

Geer, D. (2005). Chip makers turn to multicore processors, *Computer* **38,** no. 5, 11–13, doi: 10.1109/MC.2005.160.

Goldstein, P., D. Dodge, M. Firpo, and L. Minner (2003). SAC2000: Signal processing and analysis tools for seismologists and engineers, in *International Geophysics*, W. H. K. Lee, H. Kanamori, P. C. Jennings, and C. Kisslinger (Editors), Vol. 81, Elsevier, 1613–1614, doi: 10.1016/S0074-6142(03)80284-X.

Heimann, S., M. Kriegerowski, M. Isken, S. Cesca, S. Daout, F. Grigoli, C. Juretzek, T. Megies, N. Nooshiri, A. Steinberg, *et al.* (2017). Pyrocko—An open-source seismology toolbox and library [Application/octet-stream,application/octet-stream,application/octet-stream,application/octet-stream], GFZ Data Services, doi: 10.5880/GFZ.2.1.2017.001.

Jakob, W., J. Rhinelander, and D. Moldovan (2017). pybind11—Seamless operability between C++11 and Python, available at https://github.com/pybind/pybind11 (last accessed July 2021).

Junek, W. N., C. A. Houchin, J. A. Wehlen III, J. E. Highcock II, and M. Waineo (2017). Acquisition of seismic, hydroacoustic, and infrasonic data with Hadoop and Accumulo, *Seismol. Res. Lett.* **88,** no. 6, 1553–1559, doi: 10.1785/0220170056.

Kong, Q., D. T. Trugman, Z. E. Ross, M. J. Bianco, B. J. Meade, and P. Gerstoft (2018). Machine learning in seismology: Turning data into insights, *Seismol. Res.. Lett.* **90,** no. 1, 3–14, doi: 10.1785/0220180259.

MacCarthy, J., O. Marcillo, and C. Trabant (2020). Seismology in the Cloud: A new streaming workflow, *Seismol. Res. Lett.* **91,** no. 3, 1804–1812, doi: 10.1785/0220190357.

MacCarthy, J. K., and C. A. Rowe (2014). Pisces: A practical seismological database library in Python, *Seismol. Res. Lett.* **85,** no. 4, 905–911, doi: 10.1785/0220140013.

Magana-Zook, S., J. M. Gaylord, D. R. Knapp, D. A. Dodge, and S. D. Ruppert (2016). Large-scale seismic waveform quality metric calculation using Hadoop, *Comput. Geosci.* **94,** 18–30, doi: 10.1016/j.cageo.2016.05.012.

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment, *Linux J.* **2014,** no. 239, 2.

Mohammadzaheri, A., H. Sadeghi, S. K. Hosseini, and M. Navazandeh (2013). DISRAY: A distributed ray tracing by map-reduce, *Comput. Geosci.* **52,** 453–458, doi: 10.1016/j.cageo.2012.10.009.

Morozov, I. B., and G. L. Pavlis (2011). Management of large seismic datasets: I. Automated building and updating using BREQ_FAST and NetDC, *Seismol. Res. Lett.* **82,** no. 2, 211–221, doi: 10.1785/gssrl.82.2.211.

Mousavi, S. M., W. L. Ellsworth, W. Zhu, L. Y. Chuang, and G. C. Beroza (2020). Earthquake transformer—An attentive deep-learning model for simultaneous earthquake detection and phase picking, *Nat. Comm.* **11,** no. 1, 3952, doi: 10.1038/s41467-020-17591-w.

Pavlis, G. L. (2011). Three-dimensional, wavefield imaging of broadband seismic array data, *Comput. Geosci.* **37,** no. 8, 1054–1066.

Pavlis, G. L., and F. L. Vernon (2010). Array processing of teleseismic body waves with the USArray, *Comput. Geosci.* **36,** no. 7, 910–920, doi: 10.1016/j.cageo.2009.10.008.

Wang, Y., and G. L. Pavlis (2016). Generalized iterative deconvolution for receiver function estimation, *Geophys. J. Int.* **204,** no. 2, 1086–1099, doi: 10.1093/gji/ggv503.

Wang, Y., R. T. Evans, and L. Huang (2019). Performant container support for HPC applications, *Proc. of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) - PEARC '19*, ACM Press, Chicago, Illinois, 1–6, doi: 10.1145/3332186.3332226.

Wessel, P., J. F. Luis, L. Uieda, R. Scharroo, F. Wobbe, W. H. F. Smith, and D. Tian (2019). The generic mapping tools version 6, *Geochem., Geophys., Geosys.* **20,** no. 11, 5556–5564, doi: 10.1029/2019GC008515.

West, J. D., and M. J. Fouch (2012). EMERALD: A web application for seismic event data processing, *Seismol. Res. Lett.* **83,** no. 6, 1061–1067, doi: 10.1785/0220110138.

Yu, E., A. Bhaskaran, S. Chen, Z. E. Ross, E. Hauksson, and R. W. Clayton (2021). Southern California earthquake data now available in the AWS Cloud, *Seismol. Res. Lett.* doi: 10.1785/0220210039.

Zaharia, M., R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.* (2016). Apache spark: A unified engine for big data processing, *Comm. ACM* **59,** no. 11, 56–65, doi: 10.1145/2934664.

Zhang, Z., L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney (2018). FanStore: Enabling efficient and scalable I/O for distributed deep learning, available at http://arxiv.org/abs/1809.10799 (last accessed July 2021).

Zhu, W., and G. C. Beroza (2019). PhaseNet: A deep-neural-network-based seismic arrival-time picking method, *Geophys. J. Int.* **216,** no. 1, 261–273, doi: 10.1093/gji/ggy423.

Manuscript received 6 July 2021
Published online 15 September 2021

Downloaded from http://pubs.geoscienceworld.org/ssa/srl/article-pdf/93/1/426/5492324/srl-2021182.1.pdf
by University of Texas at Austin user