OpenMSI: A High-Performance Web-Based Platform for Mass Spectrometry Imaging

Oliver Rübel, Annette Greiner, Shreyas Cholia, Katherine Louie, E. Wes Bethel, Trent R. Northen, and Benjamin P. Bowen

Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA, USA

Acknowledgements: This work was supported by and used resources of the National Energy Research Scientific Computing Center (NERSC), Ecosystems and Networks Integrated with Genes and Molecular Assemblies (ENIGMA), and the Low Dose Radiation Programs, which are supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank K. Yelick, G. Karpen, and M. Maxon for valuable discussions, guidance, and support. We thank D. Skinner and the Outreach, Software and Programming Group at NERSC for their ongoing efforts and support to help deliver scientific data and high-performance computing to science communities. We thank R. E. Lance-Rübel for her patience, support, and advice.

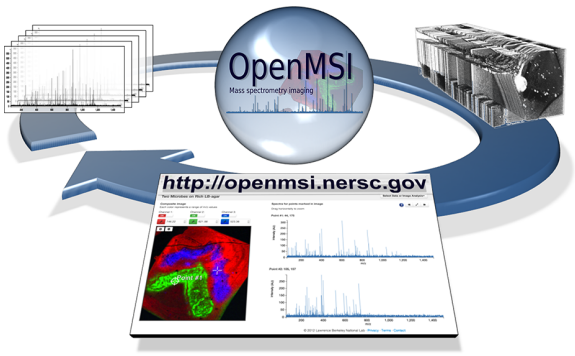
OpenMSI: A High-Performance Web-Based Platform for Mass Spectrometry Imaging

Oliver Rübel,* Annette Greiner, Shreyas Cholia, Katherine Louie, E. Wes Bethel, Trent R. Northen, and Benjamin P. Bowen*

Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, California, 94720, United States

Supporting Information

ABSTRACT: Mass spectrometry imaging (MSI) enables researchers to directly probe endogenous molecules directly within the architecture of the biological matrix. Unfortunately, efficient access, management, and analysis of the data generated by MSI approaches remain major challenges to this rapidly developing field. Despite the availability of numerous dedicated file formats and software packages, it is a widely held viewpoint that the biggest challenge is simply opening, sharing, and analyzing a file without loss of information. Here we present OpenMSI, a software framework and platform that addresses these challenges via an advanced, high-performance, extensible file format and Web API for remote data access (http://openmsi.nersc.gov). The OpenMSI file format supports storage of raw MSI data, metadata, and derived analyses in a single, self-describing format based on HDF5 and is supported by a large range of analysis software (e.g., Matlab and R) and programming languages (e.g., C++, Fortran, and Python). Careful optimization of the storage layout of MSI data sets using chunking, compression, and data replication accelerates common, selective data access operations while minimizing data storage requirements and are critical enablers of rapid data I/O. The OpenMSI file format has shown to provide >2000-fold improvement for image access operations, enabling spectrum and image retrieval in less than 0.3 s across the Internet even for 50 GB MSI data sets. To make remote high-performance compute resources accessible for analysis and to facilitate data sharing and collaboration, we describe an easy-to-use yet powerful Web API, enabling fast and convenient access to MSI data, metadata, and derived analysis results stored remotely to facilitate high-performance data analysis and enable implementation of Web based data sharing, visualization, and analysis.



Mass spectrometry imaging (MSI) as an analytical technique is rapidly finding widespread application in life sciences,[1−3] bioengineering, medicine,[4] drug development,[5,6] and studies of metabolic processes and promises to enable transformative medical diagnostics and large-scale scientific experiments.[7,8] In recent years, sample preparation methodologies, desorption-ionization techniques, and MSI instrumentation have advanced to a point where standard practices can be followed to yield high-quality data, enabling direct interrogation of the spatial distribution of metabolites and proteins within cells and tissues.

In MSI, many spatially defined mass spectra are acquired across a sample. In the raw form, the data for each position is represented as a profile of intensity values over a corresponding range of mass-to-charge (m/z) values. Modern mass spectrometers are capable of accurately measuring the m/z to approximately the mass of a single electron, generating massive and highly complex data sets.[9]

Despite numerous advances in analysis of MSI data sets, widespread adoption of MSI is hindered by a lack of fast and easy-to-use approaches for sharing, management, access, and provenance of raw MSI data and derived analyses.[10−18] While numerous open standards have been proposed for storage of MSI data, e.g, imzML or mzML,[19,20] none of the current formats efficiently support standard data access patterns, such as reading of ion images, and they often introduce large storage overheads. This lack in performance already at the file-format level unnecessarily hinders visual data exploration and high-performance, complex data analysis. These formats came about due to the urgent need for a standardized way to store data and have been adopted by many laboratories as their preferred file format. However, as Web-based technologies and high-performance, parallel data analysis and computing become mainstream in today's laboratories, it is essential that beyond standardized data storage that data formats support efficient parallel I/O for fast read and write, compact data storage, and storage and management of metadata and data provenance information to facilitate complex analysis workflows.

Storage and management of MSI data is challenging; the data is extremely large, shows large (3−4 orders of magnitude)

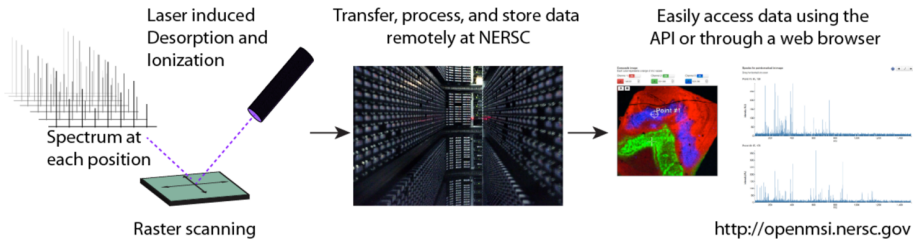


Figure 1. Illustration showing the main processing paradigm of OpenMSI. MSI data is acquired at the lab and transferred to NERSC for processing and storage. This enables us to take advantage of large-scale, high-performance compute resources to perform more complex analyses than possible using limited local compute capabilities. Raw MSI data and derived analyses results are then visualized and analyzed via the Web using the interactive OpenMSI online viewer providing fast feedback to the user.

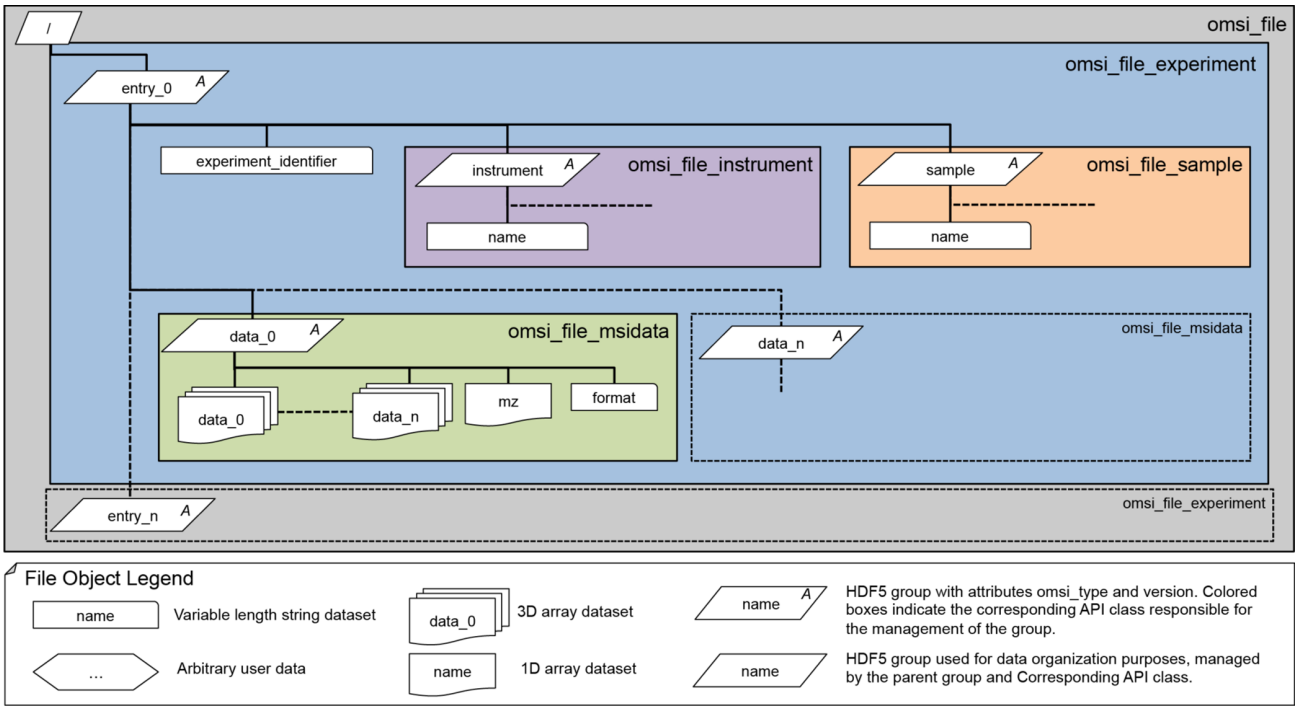


Figure 2. Overview of the hierarchy of the OpenMSI file format. The omsi file format and API follow the same semantic hierarchy. Each main HDF5 group is managed in the omsi file format API by a corresponding class (colored boxes) responsible for providing access to and creating the direct content of the group in the file.

differences between spatial and mass resolution and requires fast orthogonal access to spectra and ion images. A 2D MSI data set can be described as a three-dimensional cube of (x,y,m/z) typically containing hundreds of thousands of positions (x,y), with each position containing one or more spectra. Each spectrum describes the distribution of masses (m/z) at a given image location (x,y) and typically consists of 10^5 to 10^7 integer intensity values. See Part 1 in the Supporting Information for a detailed discussion of the MSI data requirements.

Here we describe a new paradigm in MSI data storage and processing based on a central data repository with Web-based processing and standardization of data formats. This approach takes advantage of advanced computing to make MSI data analysis rapid and accessible such that researchers can easily share and compare data. The OpenMSI file format, file API, and Web API described here build the foundation of the OpenMSI science resource (http://openmsi.nersc.gov). Overall the OpenMSI platform addresses many of the data challenges to MSI by making advanced, high-performance data analysis and computing easily accessible to MSI scientists, by enabling fast sharing and access to raw MSI data and derived analyses via the Web (see Figure 1). All software has been implemented in Python using the h5py library to interact with HDF5 and using the Django Web application framework for Web-related tasks.[21,22]

OPENMSI FILE FORMAT AND API

In recent years, there have been significant advancements in methods for management, access, and storage of big data sets. One of the most widespread file formats for storing and accessing scientific data on massive parallel file systems is the HDF5 format.[23] HDF5 is a suite of technologies, consisting of a versatile data model, portable data format, and a widely accessible software library and API, which includes a rich set of integrated features for optimization of I/O performance and tools for managing, viewing, and manipulating HDF5 data collections.[23] We here describe the newly developed OpenMSI file format, a novel, extensible, portable, self-describing, parallel-aware MSI file format based on HDF5.

Specifically, what does this mean for the end user? *Extensibility* of the format means that the object-oriented design of the file structure and API enables flexible extension of the file format to handle custom application use-cases that may not yet be fully supported. *Portability* of the format means that OpenMSI files can be manipulated and viewed using standard HDF5 APIs and tools and used directly without change on all architectures and operating systems for which HDF5 is available. HDF5 is available for Windows and Unix-based systems, including Linux and MacOS, and well-supported HDF5 APIs exist for common programming languages, e.g., C, C++, Fortran, or Python. Also, many advanced visualization and analysis systems, e.g., Matlab, R, and VisIt, support HDF5 natively. *Self-describing* of the format means that all information about the data hierarchy, data types, etc. are directly encoded in the HDF5 files so that a user can, without prior knowledge about the file, explore the file hierarchy and load data similar to how one browses files and directories on a file system. Finally, in being *parallel-aware*, the HDF5 file format has been designed with parallel applications in mind. HDF5 provides many optimizations to enable and accelerate parallel I/O to single HDF5 files. This is fundamentally important to enable parallel algorithms to scale well on modern computing platforms, with I/O being one of the main bottlenecks in many parallel analyses. The OpenMSI file format addresses in this way many of the shortcomings of current MSI data formats. In the following we first describe how data is organized in the OpenMSI data format and then the optimization of the data layout.

In HDF5, data is stored as multidimensional data arrays. HDF5 supports a large range of standard data types as well as complex user-defined compound data types. Similar to directories in file systems, data sets can be organized via so-called groups in HDF5. In addition, groups and data sets may be assigned additional attributes, defining small metadata objects that can be used, e.g., to describe the nature and/or intended usage of a primary data object, e.g., data set or group.

Figure 2 describes the organization of raw MSI data and metadata via groups and data sets in OpenMSI HDF5-based data files. The file format also supports storage and management of derived analyses results, data provenance information, and data from other imaging modalities; this, however is beyond the scope of this manuscript.

All OpenMSI files contain a root group, /. Data associated with a particular imaging experiment is then stored in a corresponding /entry_# group. This allows for convenient storage of data from multiple related experiments in a single OpenMSI data file. Each /entry_# group contains a simple string data set /entry_#/experiment_identifier used to name and uniquely identify the experiment. Metadata describing the instrument and sample associated with the experiment are then stored in separate /entry_#/instrument and /entry_#/sample groups. Since the HDF5 format is self-describing, custom metadata may be added to the sample and instrument groups without violating the OpenMSI file format. Raw MSI data and derived analysis results are then stored in dedicated /entry_#/data_# groups. This data organization allows us to store an arbitrary number of raw MSI data sets and derived analyses for each experiment represented by a /entry_# group.

Storing raw MSI data in a /entry_#/data_# group, rather than directly in an HDF5 data set, provides us with great flexibility with respect to the data layout and storage of additional associated data sets. In the basic case where the raw data defines a complete 3D MSI data cube, each data_# group contains (i) a string data set data_#/format indicating the data format used, (ii) a 1D floating-point array data_#/mz with the m/z values for the spectrum dimension, and (iii) an 3D array data_#/data_# with the 3D MSI data cube. With this design it is simple to organize MSI data in different formats (indicated by the data_#/format string) each optimized for different practical use cases. For example, to avoid possibly large storage overheads by storing an uncompressed, full 3D cube even if only a small region of interest has been imaged, the current implementation of the OpenMSI file API supports storage of mass spectra as a 2D data set of spectra along with additional small index data sets to record the relationship between spatial (x,y) locations and spectra. The design of the file format and API enables us to flexibly extend the file format to accommodate other optimized MSI storage formats as well as to integrate data from other imaging modalities, such as light microscopy. Organizing raw MSI in an HDF5 group allows us to also store multiple copies of the same data as numbered instances of data_#/data_# data sets. As described later, storing multiple copies of the same data using different data layouts can significantly accelerate orthogonal selective data accesses.

The OpenMSI file format API then follows an object-oriented design that models the group hierarchy of the file format. Each main HDF5 group (here called a managed group) is represented in the API by a corresponding class responsible for creation, management, and access of the corresponding HDF5 group type. The type of a managed groupd is uniquely determined by the naming scheme described above. For increased flexibility and extensibility of the file format, optional HDF5 attributes are associated with all main groups to indicate the interface class and version. In particular, the omsi_file_msidata interface class is designed to provide convenient access to raw MSI data stored in data_#/ groups independent of the data format used. The class provides a convenient array-based interface which allows a user to interact with the data as a 3D data cube independent of whether the MSI data set is stored as a full 3D cube or in a reduced data format. In the case where multiple copies of the same data set are available, the interface also automatically determines the data copy that is best suited to resolve a given data request. Providing a consistent data interface, independent of the underlying storage format, significantly simplifies the access to the data and eases development of data analysis algorithms. At the same time, the omsi_file_msidata interface allows developers to directly access all HDF5 data objets associated with the corresponding data_# group, enabling development of algorithms that are optimized to take advantage of different data organizations.

Data Layout Optimization. While HDF5 natively supports multidimensional arrays, on disk the data must be linearized to a one-dimensional data stream. The data layout describes the strategy by which the data is linearized. Traditional binary formats typically flatten MSI data into a single monolithic block on disk by storing the MSI data one spectrum at a time. These types of data layouts, in which the entire data is serialized into a monolithic block on disk that maps directly to a memory buffer of the size of the data sets, are referred to as contiguous data layouts. The traditional, one-spectrum-at-a-time continuous data layout is well suited to access single full spectra but shows very poor performance for access of ion images (see part 3 in the Supporting Information). To achieve optimal performance for the typical selective read operations on MSI data, in particular read of (i) spectra, (ii) ion

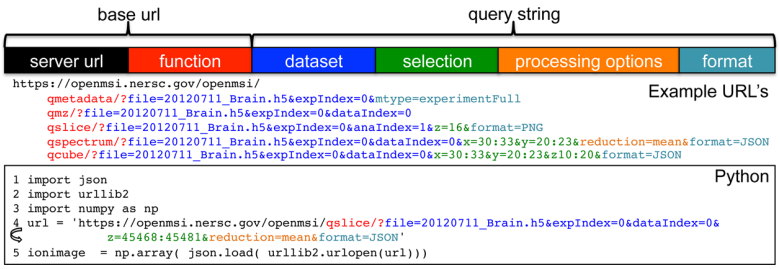


Figure 3. Illustration of the design of OpenMSI URL data requests (top), example URLs for data retrieval (middle), and a simple Python application example (bottom). Query string parameters may appear in arbitrary order.

images, and (iii) subcubes, our file format and API supports a number of data layout optimizations, including chunking, compression, and data replication, described in detail in the following. All data layout optimizations are implemented transparently for the user directly by HDF5 (chunking and compression) and the omsi_file_msidata API (data replication), allowing the user to interact with the data in a consistent manner independent of the data layout used to store the data on disk.

Accelerating Selective Data Access Operations Using Chunking. Chunked data layouts are an important alternative to contiguous data layouts. Chunking splits the data into multiple independent subparts, so-called, chunks, which are stored separately in the file. In HDF5, chunks may be stored in any order and at any position in the file, allowing chunks to be written and read independently, enabling efficient parallel read/write and improved I/O performance when operating on subsets of the data. Using chunking allows us to optimize the data layout to enable fast access to select data portions by improving data locality, hence, reducing the number of I/O operations needed and the size of the data that has to be traversed. Which chunked data layout is best depends greatly on the data access patterns to be optimized. Considering the most common data access patterns in MSI, we focus on the following main chunking strategies: (i) spectra aligned chunking, i.e., store a single full or partial spectrum per chunk; (ii) image-aligned chunking, i.e., store a single full or partial ion image per chunk; and (iii) hybrid chunking, i.e., store a 3D subcube describing a subset of multiple spectra and ion images (see part 5 in the Supporting Information).

Reducing Storage Cost and Accelerating I/O Using Compression. For chunked data layouts, HDF5 allows the data, i.e., the individual chunks, to pass through a series of user-defined I/O filters while being written to or read from disk. I/O filters are applied transparently by HDF5 whenever necessary, allowing the user to interact with the data in a consistent manner independent of the I/O filters used. Here, we focus on the use of compression filters with the goal to reduce storage cost and to accelerate data read operations by reducing the amount of data that needs to be transferred via the system bus and network. To ensure broad applicability of the OpenMSI file format, we focus on the use of gzip compression, which is (in contrast to, e.g., szip and LZF) available by default as part of HDF5. Gzip defines a lossless compression scheme, i.e., no information is lost in the compression process.

Accelerating Orthogonal Data Accesses Using Data Replication. Linearization of the data on disk makes it impossible to achieve optimal performance for orthogonal data access operations, here access to spectra and ion images. Data layouts that are optimal for access of spectra are worst for access to ion images and vice versa. While it may seem undesirable at first sight, replicated storage of MSI data using different optimized data layouts can significantly improve selective read performance, improve responsiveness of interactive applications, and substantially reduce the compute cost for parallel data analyses. Support for replicated data storage is implemented transparently in OpenMSI via the omsi_file_msidata API class, which allows users to interact with the data as a single, regular MSI data set. In the case where multiple copies of a MSI data set are available, the API automatically selects the data set that is most efficient to resolve a given data request and retrieves the data. In the context of OpenMSI, we often store two copies of the data, one optimized for access of spectra and one optimized for access of ion images with the chunking strategies automatically determined by the API. Even when storing the data twice, the resulting compressed MSI HDF5 files are in practice still substantially smaller (typically half the size or less) than the original raw binary data.

WEB API

The primary goals during the design of the OpenMSI Web API have been simplicity and usability. One primary objective has been to efficiently support exploratory analyses of the data via the Web while computationally intensive analyses are executed on high-performance compute resources at NERSC. We observe that most data analyses and visualization tasks are based on the following three data access pattern: (i) read spectra; (ii) read ion images; and (iii) read arbitrary subcubes of the data. We furthermore observe that while MSI data sets are large, the data required during individual data requests for data exploration are typically small.

The OpenMSI Web API consists of just five simple functions, qmetadata, qmz, qslice, qspectrum, and qcube, which together provide full access to the data, including metadata and raw MSI and derived analysis data. The basic methods are simple and can be effectively encoded in URL patterns (see Figure 3).

The qmetadata call is used to retrieve metadata information about which files are available on the server and which information is available in the files. The qmz call is used to retrieve information about the m/z data axis. Information about the m/z axis is in practice frequently reused. To avoid large overheads due to repeated transfer of the m/z data, we separate this into an independent call that is usually executed once at the beginning of any analysis.

The qslice, qspectrum, and qcube patterns are designed to provide easy-to-use support for the three most common selective access patterns, i.e., read ion image slices, read m/z spectra and read arbitrary subcubes of the data. To minimize

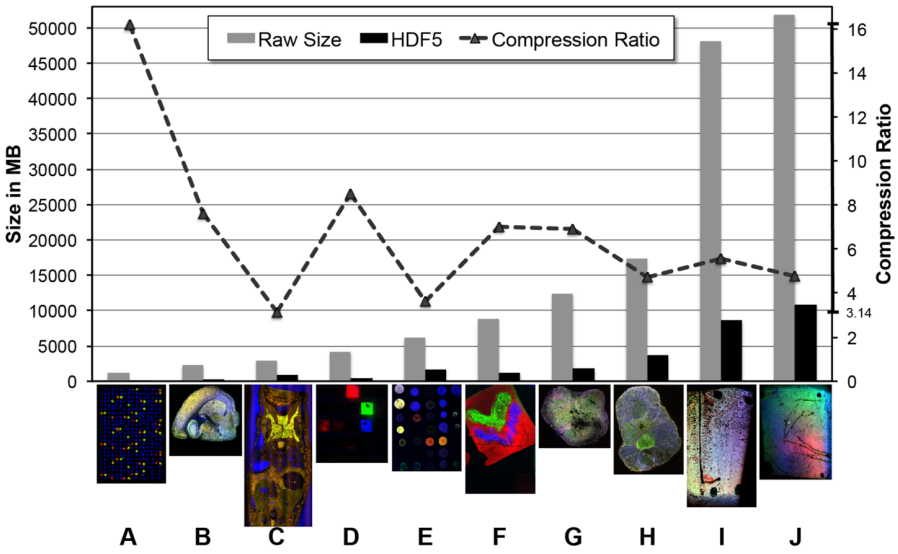


Figure 4. Size of original img data (gray bars) compared to the same data stored using the OpenMSI HDF5 data format (black bars) using gzip compression and a hybrid chunking of 4 × 4 × 2048. We observe very good compression ratios (curve, right axis) in all cases. See also Figure 6 for more details about the data sets.

the amount of data that needs to be transferred via the Web, all access patterns support common data reduction operations, including maximum, minimum, average, standard deviation, variance, etc., which are applied on the server side prior to transfer of the data. This allows one to conveniently access, e.g., maximum projection ion images of selected m/z ranges or mean spectra for arbitrary sets of spectra, while only the final image or spectrum needs to be transferred via the Web. Further detailed descriptions of the five URL patterns are provided in part 4 in the Supporting Information.

We implemented the Web API in Python using the Django Web application framework, and like the file format, the Web service is cross platform compatible. In the current implementation of the Web API, we typically transfer all data either as easy-to-use JSON objects or as images (e.g., ion images of curve plots of spectra). To also support efficient retrieval of larger subsets of MSI data sets, we plan in the near future to also support retrieval of data directly in binary HDF5 format.

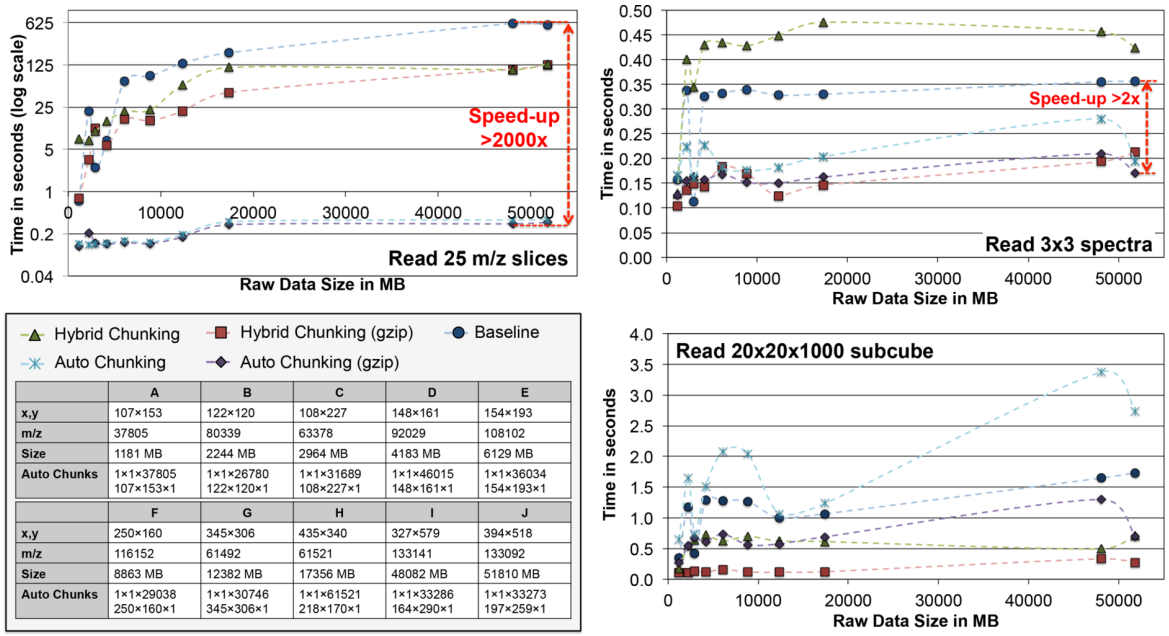
WEB-BASED DATA EXPLORATION

To demonstrate the applicability of the OpenMSI platform, we have developed an interactive Web-based, HTML5 data viewer based on the OpenMSI Web API. Using this viewer, a user can interactively define ion images and spectra to be displayed. Ion images and spectra are directly retrieved from the file from the complete, raw MSI data during each data request. As we show next, the OpenMSI data format and Web API can resolve these data requests in less than ~0.25 s via the Web even for large 50 GB MSI data sets. The viewer uses the standard URL patterns without any knowledge about the specific names of data sets or organization of the data in the HDF5 files. The Web client is in this way isolated from any specific implementation details on the server end and can flexibly display images and spectra for raw data and all derived analyses and their dependencies. The OpenMSI viewer is available online at http://openmsi.nersc.gov/ (see also part 8 in the Supporting Information).

EVALUATION AND RESULTS

Data Layout Optimization and Performance. The goal of this study is to evaluate the effectiveness of the various data layout optimizations available as part of the OpenMSI file format and API. Although text-based formats (e.g., XML) are very common in MSI,[19,20] such formats are optimized for ease of use, not efficient data storage and fast data access. We, therefore, compare the various optimized data layouts to the common and much more efficient continuous binary data layout. However, we would like to note that due to the large storage and data read overheads of XML-based formats, the improvements in read performance and storage requirements would be in practice one or several orders of magnitude greater, if we were to take XML-based formats into account.

Identifying a Suitable Hybrid Chunked Data Layout. In practice, we expect spectra-aligned chunking to provide optimal performance for access of single, complete spectra while providing poor performance for access of ion images and vice versa for image-aligned chunked data (see parts 2, 3, and 5 in the Supporting Information). Hybrid chunked data layouts promise to provide fast read performance to arbitrary subcubes of the data while providing a compromise in performance for access to ion images and spectra. However, the large differences in resolution in physical space (x,y) and the spectra (m/z) in MSI data, make finding a well-performing hybrid chunking challenging. To identify a good hybrid chunked data layout we performed a large-scale autotuning-type experiment in which we explored: (i, ii, iii) the read performance of spectra, ion images, and subcubes; (iv) the data write performance; and (v) the storage requirements of all $k \times k \times l$ hybrid chunked data layouts with $k \in [1, 2, 4, 8, 16, 32]$ and $l \in [128, 256, 512, 1024, 2048, 4096, 8192]$ using a 100 × 100 × 100 000 sized data set as reference. These experiments have shown that a chunked layout of 4 × 4 × 2048 may provide good performance in all above-mentioned evaluation criteria, and we use this configuration in the following to exemplify the performance characteristics of a hybrid chunked data layout. A detailed discussion of these experiments is available in part 5 in the Supporting Information.



	A	B	C	D	E
x,y	107×153	122×120	108×227	148×161	154×193
m/z	37805	80339	63378	92029	108102
Size	1181 MB	2244 MB	2964 MB	4183 MB	6129 MB
Auto Chunks	1×1×37805 107×153×1	1×1×26780 122×120×1	1×1×31689 108×227×1	1×1×46015 148×161×1	1×1×36034 154×193×1

	F	G	H	I	J
x,y	250×160	345×306	435×340	327×579	394×518
m/z	116152	61492	61521	133141	133092
Size	8863 MB	12382 MB	17356 MB	48082 MB	51810 MB
Auto Chunks	1×1×29038 250×160×1	1×1×30746 345×306×1	1×1×61521 218×170×1	1×1×33286 164×290×1	1×1×33273 197×259×1

Figure 5. Serial read performance for the three most common data access patterns using the following data layouts (i) default monolithic layout (baseline), (ii,iii) hybrid (4 × 4 × 2048) chunking with and without compression, and (iv, v) auto chunking with and without compression. The auto chunking option uses two data copies optimized for retrieval of ion images and full spectra (see *Auto Chunks* row table above). The results were generated using a desktop workstation equipped with two Intel Xeon CPU, E5630 @ 2.53 GHz (only one core used here) and 18 GB of RAM.

Data Compression. Next we examined the ability of compression to reduce file size. Figure 4 shows a comparison of the size of a diverse set of MSI data sets stored using the OpenMSI file format compared to the standard raw binary data. We achieved 3−16 times compression without loss of data, e.g., a 3 GB image could be compressed to only 0.5 GB. This means, even when replicating the MSI data to accelerate data access, the resulting OpenMSI files are still much smaller than the raw binary data.

The combination of chunking and compression has also shown itself to be a viable solution for efficient storage of partial MSI data cubes and processed spectra. In this case, the data is still described as a complete MSI data cube. However, chunks are allocated by HDF5 during the first write, i.e., empty data chunks are never allocated by HDF5, while missing data values are automatically completed with zero values upon read. Furthermore, partial chunks are completed with zeros, which can be compressed very efficiently with very little overhead. To illustrate the effectiveness of this approach, we chose as an example an MSI data set of a lung with a resolution of 132 × 149 × 300 000. In the data set, an arbitrary region of interest consisting of 12 654 spectra has been imaged, and the spectra were preprocessed to remove background noise. From the total 5 900 400 000 data values (i.e., ∼11 800 MB) only 107 007 401 values (i.e., ∼214 MB) are nonzero. Using a hybrid chunked layout of 4 × 4 × 2048 in combination with compression, we require only ∼196 MB to store the complete 132 × 149 × 300 000 data cube while allowing the user to seamlessly interact with the data as if it were a complete MSI data cube.

Optimizing Data Read Performance. The goal of the following tests has been to evaluate the performance of our file format and to identify the best suited data layouts. To evaluate the performance of different data layouts for the most common selective read patterns, we defined the following three representative test cases: (i) read 25 consecutive m/z slices, (ii) read a 3 × 3 subset of complete spectra, and (iii) read a 20 × 20 × 1000 subcube of the data. We here compare the performance of the following five data layouts: (i) the default monolithic layout (baseline), (ii) a hybrid (4 × 4 × 2048) chunking with compression, (iii) a hybrid (4 × 4 × 2048) chunking without compression, (iv) an autochunked data layout with compression, and (v) the same autochunked layout without compression. The autochunked data layout uses data replication in addition to chunking and compression to further optimize data read performance. Here, the data is stored twice using a spectrum-aligned and an image-aligned data chunking (see Figure 5), while the OpenMSI file API automatically chooses the best-suited data layout for a given data read.

To demonstrate the performance across a broad range of MSI data sets, we have chosen 10 MSI data sets that show varying spatial and m/z resolution and range in size between 1 GB up to 50 GB (see Figure 5). All tests were performed on a local desktop workstation equipped with two quad-core Intel Xeon E5630 CPUs running at 2.53 GHz and 18 GB of RAM. All data was stored on a local 1 TB regular spinning-disk hard drive. The tests were performed in serial, i.e., only one of the available compute cores was used in the tests. The tests were implemented in Python, and the source code of the tests is available in part 6 in the Supporting Information. We performed 50 random read operations for each of the 150 test cases, while we randomized (i) the m/z value for the image read, (ii) the (x/y) location for spectra read, and (iii) the $(x,y,m/z)$ origin for the subcube read. We report the 95th percentile of all measurements to demonstrate the expected sustained read performance for the different data layouts.

Figure 5 summarizes the results from all selective read performance tests. We observed that the baseline data layout shows particularly poor performance for the read of ion images (Figure 5, top left), requiring more than 600 s to retrieve just 25 consecutive images for data set I. Even though 25 ion images

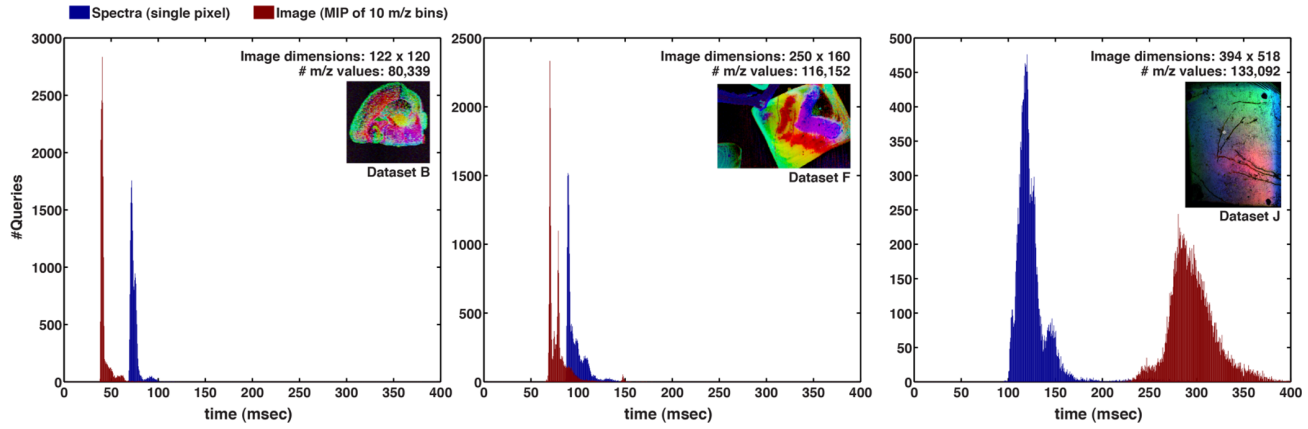


Figure 6. Web-based read performance. For each file, we evaluated 20 000 random single spectrum and 20 000 random ion-image data requests returning a maximum intensity projection over 10 m/z bins. The remote data requests were performed using Matlab (see also part 6 in the Supporting Information for further details).

constitute only ~9.5 MB of binary data, the entire ~48 GB data volume needs to be traversed to retrieve the data using the baseline layout. For the hybrid-chunked data layout (with compression) we observed speed-ups of up to ~6.3× for the image read, ~2.6× for the spectrum read, and ~11.2× for the subcube read compared to the baseline data layout. While this improvement in performance is significant, the read performance of the hybrid-chunked data layout is still insufficient for many time-critical analysis tasks and interactive data applications on large MSI data sets. This is due to the compromise the hybrid chunking is making in terms of performance to support orthogonal data access patterns.

Using the autochunked data layout (with compression) of the OpenMSI platform yielded speed-ups of more than 2000× for the image read, enabling data read of ion images and spectra in less than 0.3 s even for the largest test data set. We also found that the performance for reading ion images using the autochunked approach depends mainly on the spatial resolution (i.e., number of pixels) of the images and is mostly independent of the resolution of the data in m/z (i.e., the total number of images). Similarly, the read performance of spectra is largely independent of the spatial resolution of the data in the autochunked case. These results suggest that this approach is scalable to meet the needs of data at scales higher than what is typically generated today. Using this approach enables for the first time fast retrieval of both spectra and ion image directly from the file without requiring caching of the data in memory. The performance we observe is sufficient to support interactive data exploration tasks even for very large MSI data sets.

We observe that the compressed data layouts perform significantly better even for reads from local disk when compared to the corresponding uncompressed data layouts. In cases where the data is stored on external storage systems, we would expect this behavior to be further amplified due to the reduced amount of data that needs to be transferred via the network when the data is compressed.

Performance of Web-Based Data Access Operations. To test the performance of the OpenMSI platform's ability to access data across the Internet, images and spectra were programmatically retrieved from the server to a laptop computer. With the expectation that the size of a particular MSI file would significantly affect the time required to transfer results, we chose data sets J, F, and B (sizes are shown in Figure 6). We retrieved for each file 20 000 images and 20 000 spectra from the server at random spatial coordinates and m/z ranges and using the qslice and qspectrum commands, respectively. These requests were implemented in Matlab using the urlread command to retrieve the data as JSON structured text (the source code is provided in part 8 in the Supporting Information). The computer requesting the data was a MacBook Pro laptop with a 2.2 GHz Intel Core i7 processor and 8 GB of 1333 MHz RAM. All requests were made to a server at NERSC https://openmsi.nersc.gov. All of the files were stored on the physical, regular spinning disk of that server. The laptop was connected to a standard (1 Gb/s) office Ethernet connection in Berkeley, California.

The tests showed that the OpenMSI platform reliably supports subsecond data retrieval times for a wide range of MSI file sizes. For these data sets, the average time to retrieve a spectrum ranged between 74 and 126 ms. The average time to retrieve an image ranged between 43 and 294 ms. These results are consistent with the performance we have observed in the previous section for read performance directly from file. Figure 6 shows the histogram of the 20 000 requests for each of the six test cases. We observed very reliable read performance in all cases, indicated by the compact distribution of response times in the histograms.

URL-Based Data Analysis Sharing. Using the OpenMSI Web viewer prototype, a URL can be shared that presents the user with an interactive view based on specified visualization parameters, e.g., http://openmsi.nersc.gov/openmsi/client/viewer/?file=20120711_Brain.h5&expIndex=0&dataIndex=0&redValue=868.6&greenValue=840.6&blueValue=824.6&rangeValue=0.2&cursorX1=40&cursorY1=40&cursorX2=80&cursorY2=80

In this example, the m/z values and range are specified for creating an RGB image of three distinct ions and the spatial locations of two cursors are defined, selecting two spectra of interest plotted separately (see part 8 in the Supporting Information, OpenMSI Viewer).

CONCLUSION

We have described the OpenMSI platform and shown that it addresses many of the data challenges to MSI by making advanced, high-performance data analysis and computing easily accessible via the Web (http://openmsi.nersc.gov). The use of

the OpenMSI HDF5-based file format was found to be highly suited for this application. Optimization of data layouts using chunking, compression, and data replication were found to be critical enablers of rapid data access and resulted in >2000-fold improvement in image access. The Web-based API design enables easy-to-implement data access patterns with data retrieval speeds of less than 0.3 s across the Internet even for large 50 GB MSI data sets. By making MSI easily accessible, without the need for advanced knowledge in high-performance data analysis and computing, OpenMSI promises to transform how MSI is used in practice and promotes the widespread adoption of MSI as a novel imaging approach. While this has been beyond the scope of this manuscript, we would like to note that OpenMSI also supports management, provenance, and visualization of derived data analyses. In addition, the format is well suited to three-dimensional imaging, SIMS, and other mass spectrometry imaging approaches.

■ ASSOCIATED CONTENT

Supporting Information

Additional information as noted in text. This material is available free of charge via the Internet at http://pubs.acs.org.

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: oruebel@lbl.gov.

*E-mail: bpbowen@lbl.gov

Author Contributions

O. Rübel and B. Bowen are the project leads of OpenMSI and are responsible for conception, overall design, and writing of the manuscript. A. Greiner and K. Louie contributed to design, development, and testing of the Web-based viewer and reviewed the manuscript. S. Cholia contributed to the design, development, and setup of the OpenMSI Web services and server and reviewed the manuscript. T.R.N. and E.W.B. contributed to the design and manuscript editing.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

This work was supported by and used resources of the National Energy Research Scientific Computing Center (NERSC), Ecosystems and Networks Integrated with Genes and Molecular Assemblies (ENIGMA), and the Low Dose Radiation Programs, which are supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank K. Yelick, G. Karpen, and M. Maxon for valuable discussions, guidance, and support. We thank D. Skinner and the Outreach, Software and Programming Group at NERSC for their ongoing efforts and support to help deliver scientific data and high-performance computing to science communities. We thank R. E. Lance-Rübel for her patience, support, and advice.

■ REFERENCES

(1) Garden, R. W.; Sweedler, J. V. *Anal. Chem.* **2000**, *72* (1), 30−36.

(2) Caprioli, R. M.; Farmer, T. B.; Gile, J. *Anal. Chem.* **1997**, *69* (23), 4751−4760.

(3) Stoeckli, M.; Chaurand, P.; Hallahan, D. E.; Caprioli, R. M. *Nat. Med.* **2001**, *7* (4), 493−496.

(4) Monroe, E. B.; Annangudi, S. P.; Hatcher, N. G.; Gutstein, H. B.; Rubakhin, S. S.; Sweedler, J. V. *Proteomics* **2008**, *8* (18), 3746−3754.

(5) Wiseman, J. M.; Ifa, D. R.; Zhu, Y.; Kissinger, C. B.; Manicke, N. E.; Kissinger, P. T.; Cooks, R. G. *Proc. Natl. Acad. Sci. U.S.A.* **2008**, *105* (47), 18120−18125.

(6) McLean, J. A.; Ridenour, W. B.; Caprioli, R. M. *J. Mass Spectrom.* **2007**, *42* (8), 1099−1105.

(7) Louie, K. B.; Bowen, B. P.; McAlhany, S.; Huang, Y.; Price, J. C.; Mao, J. H.; Hellerstein, M.; Northen, T. R. *Sci. Rep.-U.K.* **2013**, *3*, 1656.

(8) Reindl, W.; Bowen, B. P.; Balamotis, M. A.; Green, J. E.; Northen, T. R. *Integr. Biol.* **2011**, *3* (4), 460−467.

(9) Rompp, A.; Guenther, S.; Takats, Z.; Spengler, B. *Anal. Bioanal. Chem.* **2011**, *401* (1), 65−73.

(10) Suits, F.; Fehniger, T. E.; Vegvari, A.; Marko-Varga, G.; Horvatovich, P. *Anal. Chem.* **2013**, *85* (9), 4398−4404.

(11) Parry, R. M.; Galhena, A. S.; Gamage, C. M.; Bennett, R. V.; Wang, M. D.; Fernández, F. M. *J. Am. Soc. Mass Spectrom.* **2013**, *24* (4), 646−649.

(12) Oetjen, J.; Aichler, M.; Trede, D.; Strehlow, J.; Berger, J.; Heldmann, S.; Becker, M.; Gottschalk, M.; Kobarg, J. H.; Wirtz, S.; Schiffler, S.; Thiele, H.; Walch, A.; Maass, P.; Alexandrov, T. *J. Proteomics* **2013**, *90*, 52−60.

(13) Alexandrov, T.; Becker, M.; Guntinas-Lichius, O.; Ernst, G.; von Eggeling, F. *J. Cancer Res. Clin. Oncol.* **2013**, *139* (1), 85−95.

(14) Trede, D.; Schiffler, S.; Becker, M.; Wirtz, S.; Steinhorst, K.; Strehlow, J.; Aichler, M.; Kobarg, J. H.; Oetjen, J.; Dyatlov, A.; Heldmann, S.; Walch, A.; Thiele, H.; Maass, P.; Alexandrov, T. *Anal. Chem.* **2012**, *84* (14), 6079−6087.

(15) Trede, D.; Kobarg, J. H.; Oetjen, J.; Thiele, H.; Maass, P.; Alexandrov, T. *J. Integr. Bioinf.* **2012**, *9* (1), 189.

(16) Moree, W. J.; Phelan, V. V.; Wu, C.-H.; Bandeira, N.; Cornett, D. S.; Duggan, B. M.; Dorrestein, P. C. *Proc. Natl. Acad. Sci. U.S.A.* **2012**, *109* (34), 13811−13816.

(17) Alexandrov, T. *BMC Bioinf.* **2012**, *13* (Suppl 16), S11.

(18) Bruand, J.; Alexandrov, T.; Sistla, S.; Wisztorski, M.; Meriaux, C.; Becker, M.; Salzet, M.; Fournier, I.; Macagno, E.; Bafna, V. *J. Proteome Res.* **2011**, *10* (10), 4734−4743.

(19) Römpp, A.; Schramm, T.; Hester, A.; Klinkert, I.; Both, J.-P.; Heeren, R. M. A.; Stöckli, M.; Spengler, B. *Methods Mol. Biol.* **2011**, *696*, 205−224.

(20) Martens, L.; Chambers, M.; Sturm, M.; Kessner, D.; Levander, F.; Shofstahl, J.; Tang, W. H.; Römpp, A.; Neumann, S.; Pizarro, A. D.; Montecchi-Palazzi, L.; Tasman, N.; Coleman, M.; Reisinger, F.; Souda, P.; Hermjakob, H.; Binz, P. A.; Deutsch, E. W. *Mol. Cell. Proteomics* **2010**, *10* (1), R110.000133−R110.000133.

(21) McKinney, W. *Python for Data Analysis*; O'Reilly Media, Inc.: Sebastopol, CA, 2012; p 452.

(22) Forcier, J.; Bissex, P.; Chun, W. *Python Web Development with Django*; Pearson Education, Inc.: Boston, MA, 200p; p 408.

(23) The HDF Group. *HDF5 User's Guide*; November 2011.

Supplemental Material

1. MSI Data Requirements

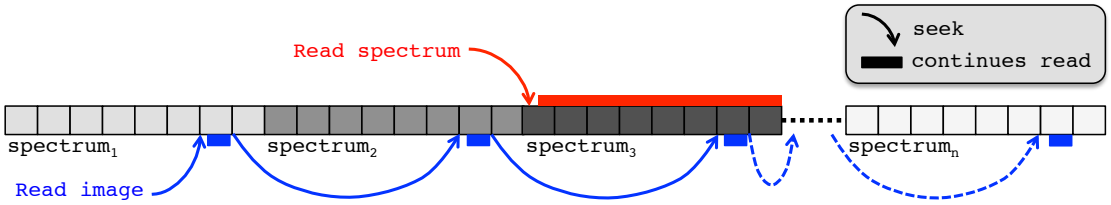
Storage and management of MSI data is challenging; the data is extremely large, shows large (three to four orders of magnitude) differences between spatial and mass resolution, and requires fast orthogonal accesses to spectra and ion images. A 2D MSI dataset can be described as a three-dimensional cube of (x,y,m/z). Current 2D raw MSI datasets contain spectra from hundreds of thousands of positions, with each position containing one or more spectra. Each spectrum describes the distribution of masses at a given image location (pixel) and typically consists of 10^5 to 10^7 integer intensity values. Using currently available instrumentation, MSI datasets with 10^6 pixels and 10^6 mass bins could be easily acquired, resulting in a raw size of 4TB. Unfortunately, acquisition of such datasets is largely limited by the inability of current tools and methods to store and process MSI datasets of this magnitude.

In practice, MSI data is used in a write-once read-many fashion, i.e., the data is written once during data acquisition and read repeatedly during the visualization and analysis process. For instance, many visualization and analysis algorithms do not process the full MSI data cube at once but rely on repeated selective read of (i) spectra, (ii) ion images, and (iii) arbitrary 3D subsets of the data. The performance of current tools for visual exploration of MSI data is largely limited due to the poor performance of current MSI data formats in resolving selective data accesses efficiently. Accelerating these most common data access patterns and enabling efficient parallel read are fundamental requirements for enabling fast and efficient analysis of large MSI datasets.

A second main requirement is the need for flexible storage of metadata describing (among others) the sample, instrument, and imaging settings. Current MSI data formats often require the use of secondary data files for storage of metadata and often do not allow for flexible extension of metadata storage.

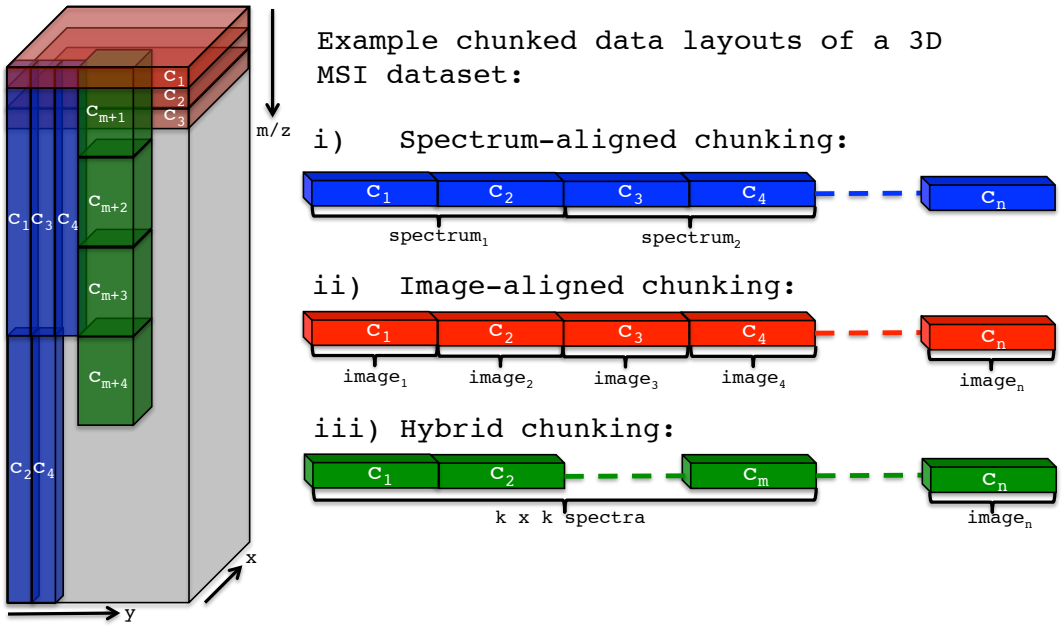
Finally, to facilitate distributed, inter-disciplinary, collaborative analyses of MSI data and to enable the scientific community to share and benefit from results from other MSI experiments, researchers must be able to share data and analysis results with other researchers around the world and the MSI community at large. An appropriately designed file format must therefore be self-describing, platform independent, and easily accessible via a large range of programming languages and analysis systems. In the following we describe the OpenMSI file format, file format and analysis APIs, and web API, which we have developed to meet the abovementioned fundamental data and analysis requirements and needs.

2. Data Read



The figure shown above illustrates the layout most commonly used for storing MSI data in binary form. The data is stored as a single monolithic block arranged on disk one spectrum at a time. This layout is well suited for retrieval of single full spectra from disk (red) but requires a large number of seek and small read operations to retrieve a single ion-image (blue). In order to retrieve an ion image, the full dataset has to be traversed, leading to poor performance in particular for large MSI datasets.

3. Data Layout



The figure shown above illustrates different basic chunked data layouts for storage of MSI data. Use of chunking enables independent read/write access to individual data chunks and can significantly improve the locality of data as it is linearized on disk. Chunking allows in this way optimization of the data layout to improve the performance of a select set of I/O patterns.

4. Description of the OpenMSI WebAPI URL Patterns

The basic URL patterns are constructed as follows:

`<baseURL>/<command>/?<querystring>`

and consist of the following three main components:

- `<baseURL>` : The basic URL where the server is running, e.g. https://openmsi.nersc.gov/openmsi/
- `<command>` : Depending on which data/action we are requesting a different command is used. The main available commands are :
    - `qmetadata`: Request metadata information.
    - `qmz`: Request information about the m/z axis of the data.
    - `qslice`: Request ion-slices (raw or derived) from the data.
    - `qspectrum`: Request spectra (raw or derived) from the data.
    - `qcube`: Request arbitrary structured subsets of the data.
    - `(client`: Request client webpages e.g, the OpenMSI viewer.)
- `<querystring>`: List of different function parameters.

In the following we first describe the different URL patterns and briefly describe the syntax for specifying data selections.

qmz: Requesting m/z data

Request information about the static m/z axes. This function is provided to avoid repeated transfer of the usually static m/z axes information. In most cases the m/z axes data is requested once at the beginning and reused afterwards. In cases where the m/z axes for spectra is not static—e.g., in the case of processed spectra— the qslice pattern returns the intensity values as well as the corresponding m/z values for the spectra, otherwise the m/z values are omitted.

Base pattern:

https://openmsi.nersc.gov/openmsi/qmz/?<querystring>

Query string parameters:

- Required query string parameters:
    - `file`: The filename/path of the OpenMSI HDF5 datafile to be used.
    - `expIndex` : The index of the experiment stored in the file.
- Required query parameters when requesting from raw MSI data:
    - `dataIndex` : The index of the MSI dataset to be used.

Returns:

- Returns error message or JSON object with the following entries:
    - `values_spectra` : Axes values for the spectra or null if missing in the data.

- `label_spectra` : Axes label to be used for the spectrum axes.
- `values_slice` : Values for the z axis to be used for identifying image slices or null if missing in the data. This return value is optional and is only present if different from `values_spectra`.
- `label_slice` : Label for the z axis to be used for image slices. This return value is optional and is only present if different from `label_spectra`.

`qmetadata`: Requesting metadata information

Request JSON object with metadata information pertaining to the list of available files, a file, an experiment, an analysis, an instrument or a sample.

Base Pattern:

https://openmsi.nersc.gov/openmsi/qmetadata/?<querystring>

Query string parameters:

- Required query arguments:
  - `mtype` : Type of metadata requested, one of:
    - `filelist`,
    - `file`,
    - `experiment`,
    - `experimentFull`,
    - `analysis`,
    - `instrument`,
    - `sample`,
    - `dataset`.
- Additional required query arguments for `mtype experiment, experimentFull, instrument, and sample`:
  - `filename` : The filename/path of the OpenMSI HDF5 datafile.
  - `expIndex` : The index of the experiment stored in the file.
- Additional required query arguments if `mtype` is `analysis`:
  - `filename` : The filename/path of the OpenMSI HDF5 datafile.
  - `expIndex` : The index of the experiment stored in the file.
- Additional required query arguments if `mtype` is `dataset`:
  - `filename` : The filename/path of the OpenMSI HDF5 datafile.
  - `expIndex` : The index of the experiment stored in the file
  - Raw MSI data indicator (only when requesting information for a raw MSI dataset):
    - `dataIndex` : Index of the MSI dataset

Returns:

- The function returns a JSON object with a dictionary describing the requested metadata information in a structured fashion.

`qspectrum`: Requesting spectra

Request JSON object or PNG image plot of: i) a single spectrum, ii) multiple spectra or iii) the difference of two or multiple spectra.

Base Pattern:

https://openmsi.nersc.gov/openmsi/qspectrum/?<querystring>

Query string parameters:

- Required query arguments:
  - `filename` : The filename/path of the OpenMSI HDF5 datafile.
  - `expIndex` : The index of the experiment stored in the file.
  - `format` : Output format of the returned data, one of: `JSON` or `PNG`
  - `x` : x-index(s) of the pixel/spectrum to be loaded. See Section *Data Selection* below.
  - `y` : x-index(s) of the pixel/spectrum to be loaded. See Section *Data Selection* below.
- Required query arguments when requesting from raw MSI data:
  - `dataIndex` : Index of the MSI dataset
- Additional optional query parameters:
  - `findPeaks` : Execute peak finding for the retrieved spectra (only used if format==JSON). Valid values are 0 (False) and 1 (True).
  - `reduction`: String indicating the reduction operation to be executed on the first set of spectra defined by `x`, `y`. (Default is `mean`). Reduction operations are defined as strings indicating the numpy function to be used for data reduction. Valid reduction operations include e.g.: `min`, `max`, `mean`, `median`, `std`, `var` etc..
- Optional query parameters when requesting difference spectra:
  - `x2` : x-index of the second pixel/spectra to be loaded. See Section *Data Selection* below.
  - `y2` : y-index of the second pixel/spectra to be loaded. See Section *Data Selection* below.
  - `reduction2` : String indicating the reduction operation to be executed for the second set of spectra selected by `x2`, `y2` (default is None). Reduction operations are defined as strings indicating the numpy function to be used for data reduction. Valid reduction operations include e.g.: `min`, `max`, `mean`, `median`, `std`, `var` etc.. Note if no reduction operation is applied, then the `(x,y)` shape of the first selection and second selection have to match in order to allow for the two arrays to be subtracted from each other.

Returns:

- If `format==JSON`:

    - JSON object of the raw spectrum data (or multiple spectra if no reduction is applied), if only `x`, `y` (but not `x2`, `y2`) are specified and `findPeaks` is set to 0.
    - JSON object of the difference spectrum (or multiple difference spectra if no reduction is applied), if `x`, `y` and `x2`, `y2` are specified `findPeaks` is set to 0.
    - JSON object of the raw spectrum (or difference spectra) data including additional fields with the results from the local peak finding (`spectrum`, `peak_value`, `peak_pz`) if `findPeaks` is set to 1.
    - In case that the *m/z* axis should be not static but change dynamically between spectra, then additional `spectrum_mz` key value with the m/z data is returned.
- If format==PNG:
    - PNG plot of the raw spectrum data, if only `x`, `y` are specified or PNG plot of the difference spectrum data if `x`, `y` and `x2`, `y2` are specified.

`qslice`: Requesting z data slices

Request JSON object (or gray-scale PNG image) of a single or multiple m/z image slices of the data.

Base Pattern:

https://openmsi.nersc.gov/openmsi/qslice/?<querystring>

Query string parameters:

- Required query arguments:
    - `filename` : The filename/path of the OpenMSI HDF5 datafile.
    - `expIndex` : The index of the experiment stored in the file.
    - `format` : Output format of the returned data, one of: `JSON` or `PNG`
    - `z` : z-index(s) of image slices to be loaded. See Section *Data Selection* below.
- Required query arguments when requesting from raw MSI data:
    - `dataIndex` : Index of the MSI dataset
- Additional optional query parameters:
    - `normalize`: Binary value (0=False, 1=True) indicating whether the data retrieved should be normalized by dividing by the maximum value retrieved. (Relevant only if `format==JSON`).
    - `reduction` : String indicating the reduction operation to be executed for the selected image slices (axis=2). Reduction operations are defined as strings indicating the numpy function to be used for reduction. Valid reduction operations include e.g.: `min`, `max`, `mean`, `median`, `std`, `var` etc..

Returns:

- JSON object or PNG image of the selected image slice(s).

qcube: Requesting arbitrary structured subsets of the data

Request JSON object of a general subset of the original MSI data or derived analysis data.

Base Pattern:

https://openmsi.nersc.gov/openmsi/qcube/?<querystring>

Query string parameters:

- Required query arguments:
  - `filename` : The filename/path of the OpenMSI HDF5 datafile.
  - `expIndex` : The index of the experiment stored in the file
- Required query arguments when requesting from raw MSI data:
  - `dataIndex` : Index of the MSI dataset
- Optional query arguments required for specification of data selections:
  - `x` : Selection string for x. Default value is `":"` (i.e. all). See Section *Data Selection* below.
  - `y` : Selection string for y. Default value is `":"` (i.e. all). See Section *Data Selection* below
  - `z` : Selection string for z. Default value is `":"` (i.e. all). See Section *Data Selection* below
- Additional optional query arguments:
  - `normalize` : Normalize the data by dividing by the maximum retrieved data value.
  - `reduction` : String specifying the first data reduction to be applied to the data. Reduction operations are defined as strings indicating the numpy function to be used for reduction. Valid reduction operations include e.g.: `min`, `max`, `mean`, `median`, `std`, `var` etc..
  - `axis` : The data axis along which `reduction` should be applied (default value 2, i.e., the z axis).
  - `reduction2` : Second reduction operation to be applied to the data.
  - `axis2` : Axis along which the second reduction operation should be applied. Note that the dimensionality of the data is reduced by 1 by any prior data reduction operations (default value is 0).
  - `reduction3` : Third reduction operation to be applied to the data.
  - `axis3` : Axis along which the third reduction operation should be applied. Note that the dimensionality of the data is reduced by 1 by each prior data reduction operation (default value is 0).

Returns:

- JSON object defining the array of data retrieved.

Data Selection

Basic Slicing

The data request URL's commonly support data selection parameters—e.g., `x`, `y`, or `z` —which are used to select the data that should be retrieved. There are several basic ways in which a user may specify data selections:

- **Range selection:** "**`a:b`**" indicate that all values in the range of `a` and `b` should be selected. The upper bound `b` is not included in the selection, i.e., the selection `1:10` selects the elements `1,2,3,4,5,6,7,8,9`.
- **Index selection: "`a`"** specifies a single index `a` that should be selected. NOTE: Specifying a single index usually implies that the dimensionality of the returned array is reduced by 1. E.g., a selection of `[1,4,5]` usually results in the retrieval of a single scalar corresponding to the item with index (`1,4,5`).
- **All: ":"** indicates that all values, i.e., the full range for the given dimension, should be selected.
- **Index list: "`[a,b,c,d]`"** indicates that the indices `a,b,c,d` should be selected.

Multi-dimensional Slicing

Several of the data URL patterns support multiple selection parameters, e.g., `x` and `y` in the case of `qspectrum`. These parameters are combined as `[x,y,z]` to allow retrieval of data from multi-dimensional arrays. The semantic for different combinations follows the same strategy as used by numpy (and h5py) :

- **All-to-all**: Most combinations of selections follow the all-to-all combination principal. That is all elements in the selection specified for `x` are combined with the selection specified for `y`. `x=1:4` and `y=1:3`, hence, results in the retrieval of the elements `[(1,1), (1,2), (2,1), (2,2) (3,1), (4,2)]`. All-to-all selection, hence, always result in the retrieval of of a single or multiple rectangular regions.
- **Multiple index lists**: In case that multiple index list selections are specified the lists are matched. This means if multiple lists are specified, then the lists must be of equal length and the lists are merged to define specific index-pairs to be selected. E.g, `x=[1,2]` and `y=[4,5]` results in the retrieval of the elements `[ (1,4), (2,4) ]` compared to an all-to-all matching, which would retrieve `[ (1,4), (1,5), (2,4), (2,5) ]`. This scheme allows for selection of arbitrary regions of interest. NOTE: When specifying multiple index lists, the dimensionality of the returned array may be reduced.

5. Evaluation of Hybrid Chunked Data Layouts

Goal: The goal of this study has been to identify hybrid chunked data layouts that provide a compromise in performance for common data access patterns. In this

initial study we investigated the sustained performance for repeated selective data access operations.

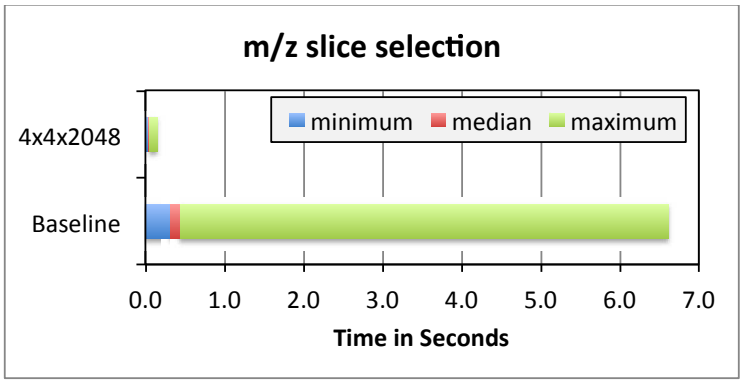
Test Platform: All test were performed using a shared login node of the hopper.nersc.gov compute system equipped with 4 quad-core AMD 2.4 GHz Opteron 8378 processors (16 cores total) and 128 GB of memory using the Lustre-based scratch file system. All test were performed in serial, i.e., only a single processor core was used.

Test Design: To evaluate the performance of different data layouts, we designed a set of test cases modeling the most common data access patterns in the analysis of MSI data.. We report for each selection test case the median time (indicating the sustained performance on an open file) and in select cases also the maximum time (indicating the selection performance after the first opening of the file). We usually repeat each selection test case 50 times for each data layout using randomized selection parameters. All tests are performed using a 100×100×100,000 test dataset. We, evaluate the performance of k×k×l layouts with k=[1, 2, 4, 8, 16, 32] and l=[128, 256, 512, 1024, 2048, 4096, 8192]. We omitted 32×32×128, 32×32×256, 32×32×512 due to the poor spectrum-at-a-time write performance of these data layouts.

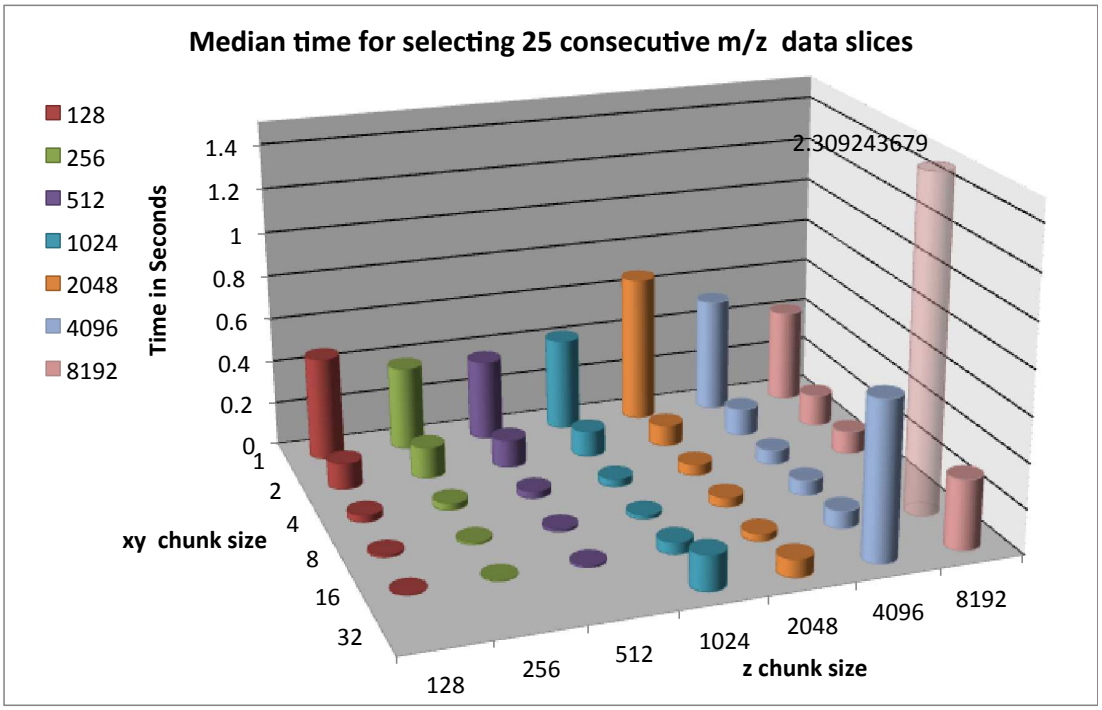
- **Case 1: m/z Slice Selection:** This test case models the selection of a series of m/z-slices of the data, and extracts a set of consecutive, full ion-images of the data.
  - **Selection:** [:,:,zmin:zmax]
  - **Randomized Selection Parameters:** zmin
  - **Dependent Selection Parameters:** zmax = zmin+25
  - **Selection Size:** 100×100×25 = 250, 000 records
    = 500, 000 bytes = 0.5MB
- **Case 2: Spectra Selection** This test case models the selection of a 5 × 5 set of full spectra.
  - **Selection:** [xmin:xmax, ymin:ymax, :]
  - **Randomized Selection Parameters:** xmin , ymin
  - **Dependent Selection Parameters: xmax = xmin+5 , ymax = ymin+5**
  - **Selection Size:** 5×5×100, 000 = 200, 000 records
    = 2, 500, 000 bytes = 5MB
- **Case 3: 3D Subcube Selection:** This selection models the general access to consecutive sub-pieces of the data, e.g., when accessing data from a particular spatial region of the data related to a particular set of m/z data values.
  - **Selection:** [xmin:xmax, ymin:ymax, zmin:zmax]
  - **Randomized Selection Parameters:** xmin , ymin, zmin
  - **Dependent Selection Parameters:** xmax = xmin+5 , ymax = ymin+5, zmax = zmin+1000
  - **Selection Size:** 5×5×1, 000 = 25, 000 records
    = 50, 000 bytes = 0.05MB

Comments:

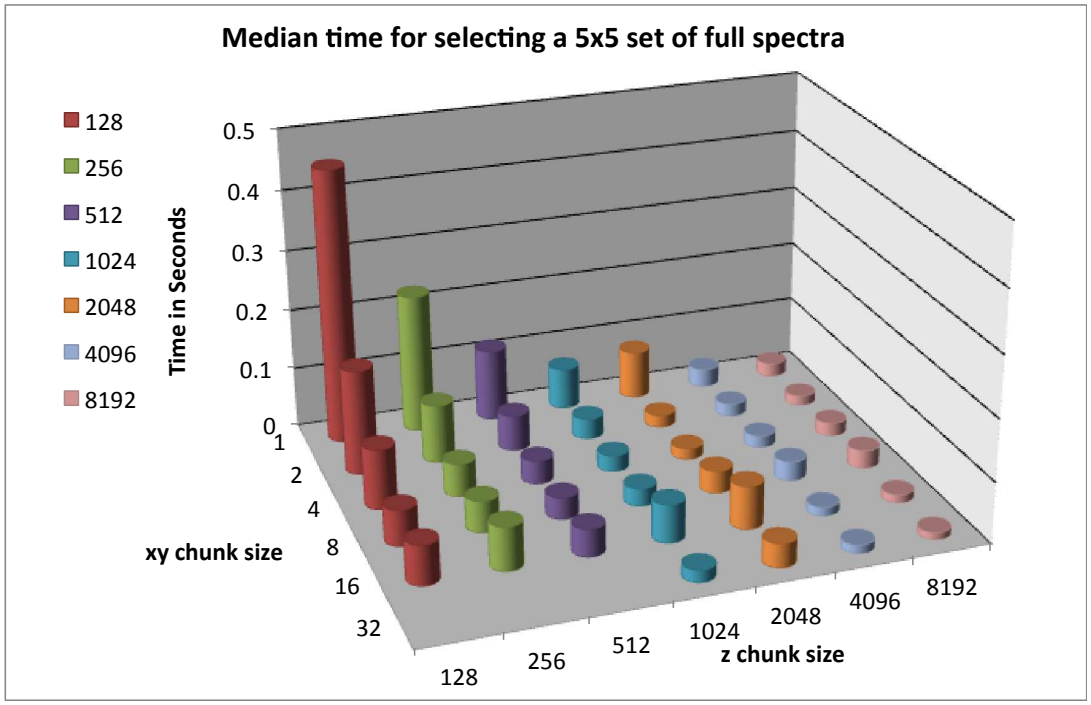
- Note, the amount of data that needs to be read and/or traversed on disk largely depends on the chosen data layout and may be significantly larger than the size of the selection.
- Note, the sustained performance, here measured by the median performance, is in practice often more important for analyses algorithms rather than web-applications, which require good worst-case performance, rather than good median performance. For hybrid chunked data layouts the general performance characteristics are in practice much more stable than for traditional monolithic data layouts (see Plot 4.1) so that the general trends of the median and 95%'il performance characteristics are often very similar.



Plot 4.1: This plot shows the minimum, median, and maximum time for reading 25 consecutive ion-images from a 100x100x100,000 test dataset. The baseline, monolithic data layout requires traversal of the full dataset in order to retrieve ion images. In the baseline case, this behavior causes the full data to be cached in memory after just a few image read operations. This behavior leads to dramatic difference between the maximum and median read performance in the baseline case. Also, in cases where the size of the MSI data exceeds the amount of available memory, the data can no longer be cached so that the median time approaches the maximum time. In contrast, the hybrid chunked data layout used in this example requires the read of typically only 625 independent chunks (i.e., 25*25 chunks in x and y) so that only a subvolume of 100x100x2048 is touched, avoiding traversal of the full data. This characteristic behavior leads to a much more stable read performance.

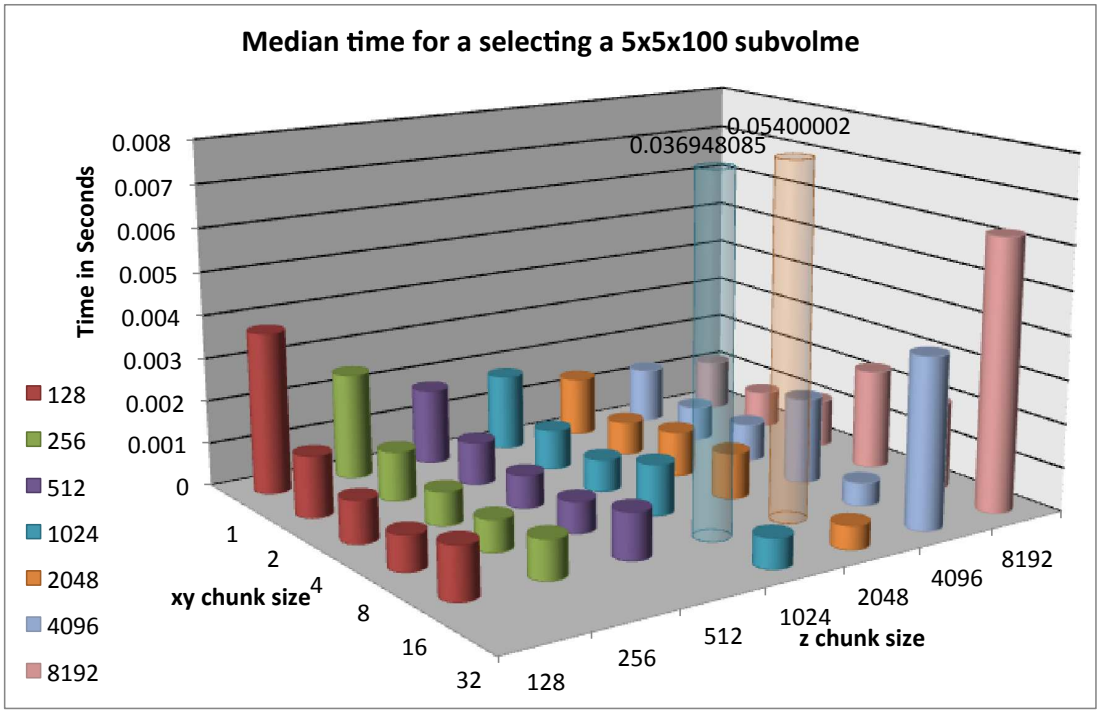


Plot 4.2: For the read of 25 ion-images we observe better read performance for hybrid chunked data layouts with larger spatial xy chunk sizes and smaller z chunk sizes. This behavior is expected; smaller z chunk sizes imply that less data needs to be read while larger xy chunks imply that less chunks need to be read.

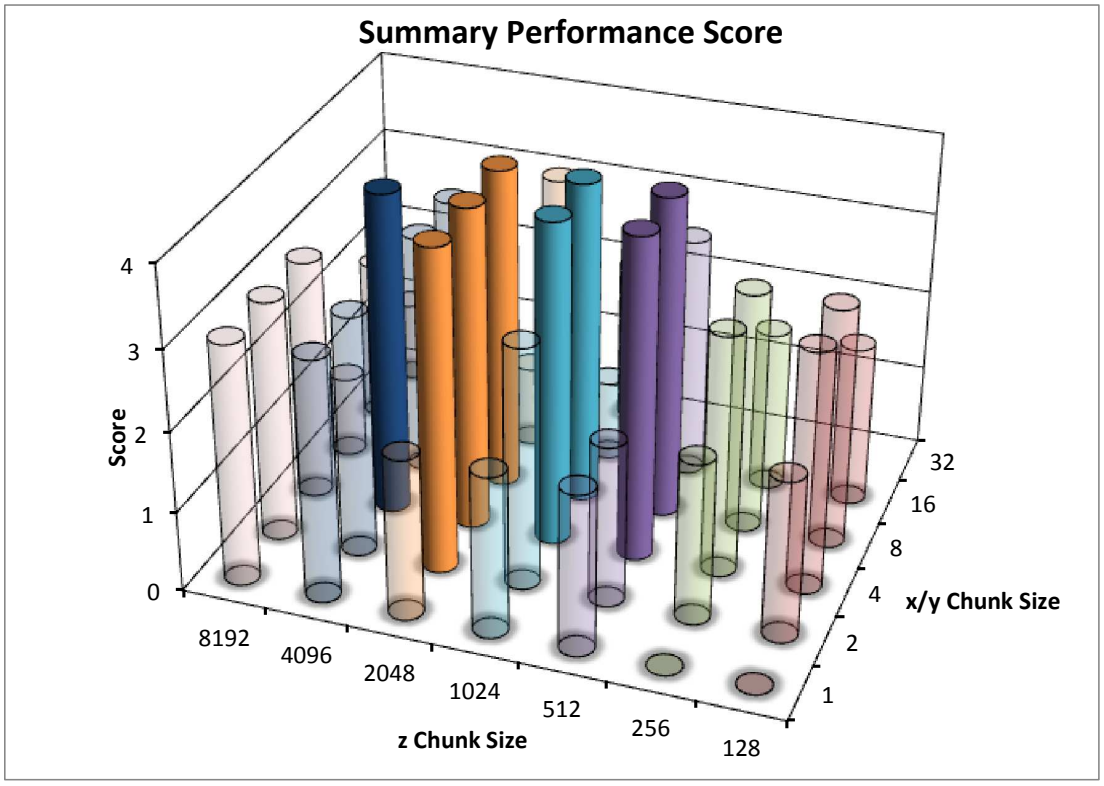


Plot 4.3: For the read of a random 5x5 subset of full spectra we observe better read performance for chunked data layouts with larger spatial z chunk sizes. This

behavior is expected as larger z chunk sizes imply that fewer chunks need to be read.



Plot 4.4: Plot showing the median read performance for the selective read of a random 5x5x100 subvolumes of the test data.



Plot 4.5: To illustrate the overall performance of the different dataset layouts and to identify the "best" layouts, we define the following set of minimum performance criteria a data layout should fulfill:

- The median time for the z-slice selection test case should be <0.1 s
- The median time for the spectra selection test case should be <0.05 s
- The median time for the 3D subcube selection test case should be <0.002 s
- The total file size should be < 2100 MB, limiting the overhead in file size for the test data to a maximum of ≈ 200MB.

Based on these criteria we can determine an overall performance score by evaluating how many of the criteria a particular data layout fulfills (with 4=best (passes all criteria) and 0=worst (does not pass any of the criteria)). We observe a cluster of 8 data layouts that satisfy the four performance conditions. Based on these results and experience with real data in practice, we selected a chunked data layout of 4x4x2048 as reference hybrid chunked data layout.

6. Python script for testing local file read performance

```
"""Simple script for testing the read performance of a set of HDF5
   files. The goal is to evaluate the performance of files with
   different data layouts for the same data. The script performs three
   main read tests: i) ion-images, ii) spectra, and iii) sub cube. To
   evaluate the estimated performance in the context of a web-based
   application, we start a new python interpreter for each data read,
   execute the read, and close the read process."""

from omsi.dataformat.omsi_file import *
import time
import json
import sys
from sys import argv,exit
import numpy as np
import os
import random
import subprocess


def main(argv=None):

    #Check and read input parameters
        if argv is None:
       argv = sys.argv

        if len(argv) < 2 :
       printHelp()
       exit(0)

        #If we are a slave process, then read the requested data #subset and exit
    if len(argv) == 8 :

       infile = argv[1]
```

```
    xmin = int(argv[2])
    xmax = int(argv[3])
    ymin = int(argv[4])
    ymax = int(argv[5])
    zmin = int(argv[6])
    zmax = int(argv[7])
    start = time.time()
    d = omsi_file( infile , 'r' ).get_exp(0).get_msidata(0)
    loaddata = d[xmin:xmax , ymin:ymax, zmin:zmax]
    #content = json.dumps( loaddata.tolist() )
    stop = (time.time() - start)
    print stop
    exit(0)

      #If we are the master process then define the test parameters
repeats = 50
outfolder  = argv[1]
filelist =  [<list_of_files>]

#Initalize the output data structures
data_shapes = {}
results = {}
for filename in filelist :

    #Initialze data shape
    f = omsi_file( filename , 'r' )
    d = f.get_exp(0).get_msidata(0)
    data_shapes[filename] = d.shape
    f.close_file()
    #Initialze output storage
         results[filename] = np.zeros( repeats ,  dtype=[ ('mz-
                    slice','f') , ('spectrum','f') , ('xyz-cube','f'),
                    ('mz-slice-all','f') , ('spectrum-all','f') , ('xyz-
                    cube-all','f')  , ('filesize' , 'f') ] )
    results[filename]['filesize'] = os.stat( filename).st_size

      #Note: We compute each test separately so that we have touched #enough data from other files to
      avoid biases due to data cacheing.
 #Note: Depending on the file system, a significant amount of data
      #and in some cases complete files) can be cached by the file
      #system. In particular for small datasets this can result in slow #initial data access and much faster
      repeated data access during a #given test.

#Compute the slice query test
for filename in filelist :
    print filename+" 25 mz-slices"
    #mz-slice selection 250,000 elements
    sliceWidthZ = 25 #xdim=100 , ydim=100
    for ri in xrange( 0 , repeats ) :

        xmin = 0
        xmax = data_shapes[filename][0]
        ymin = 0
        ymax = data_shapes[filename][1]
        zmin = random.randint(0, data_shapes[filename][2]-
                              sliceWidthZ-1 )
        zmax = zmin + sliceWidthZ
        callCommand = ["python", "testhdf5_file_read.py" , filename
                              , str(xmin) , str(xmax), str(ymin), str(ymax),
                              str(zmin) , str(zmax) ]
        start = time.time()
        p2 = subprocess.Popen(callCommand, stdout= subprocess.PIPE)
```

```
        readTime =  float(p2.stdout.read())
        stop = (time.time() - start)
        results[filename]['mz-slice'][ri] = readTime
        results[filename]['mz-slice-all'][ri] = stop

#Compute the spectra test
for filename in filelist :
    print filename+" 3 x 3  spectra"
    #mz-slice selection 250,000 elements
    sliceWidthX = 3
    sliceWidthY = 3
    for ri in xrange( 0 , repeats ) :

        xmin = random.randint(0, data_shapes[filename][0]-
                              sliceWidthX-1 )
        xmax = xmin + sliceWidthX
        ymin = random.randint(0, data_shapes[filename][1]-
                              sliceWidthY-1 )
        ymax = ymin + sliceWidthY
        zmin = 0
        zmax = data_shapes[filename][2]
        callCommand = ["python", "testhdf5_file_read.py" ,
                              filename, str(xmin) , str(xmax), str(ymin),
                              str(ymax), str(zmin) , str(zmax) ]
        start = time.time()
        p2 = subprocess.Popen(callCommand, stdout=subprocess.PIPE)
        readTime =  float(p2.stdout.read())
        stop = (time.time() - start)
        results[filename]['spectrum'][ri] = readTime
        results[filename]['spectrum-all'][ri] = stop

#Compute the cube test
for filename in filelist :
    print filename+" 20 x 20 x 1000  cube"
    #mz-slice selection 250,000 elements
    sliceWidthX = 20
    sliceWidthY = 20
    sliceWidthZ = 1000
    for ri in xrange( 0 , repeats ) :

        xmin = random.randint(0, data_shapes[filename][0]-
                              sliceWidthX-1 )
        xmax = xmin + sliceWidthX
        ymin = random.randint(0, data_shapes[filename][1]-
                              sliceWidthY-1 )
        ymax = ymin + sliceWidthY
        zmin = random.randint(0, data_shapes[filename][2]-
                              sliceWidthZ-1 )
        zmax = zmin + sliceWidthZ
        callCommand = ["python", "testhdf5_file_read.py" ,
                              filename, str(xmin) , str(xmax), str(ymin),
                              str(ymax), str(zmin) , str(zmax) ]
        start = time.time()
        p2 = subprocess.Popen(callCommand, stdout=subprocess.PIPE)
        readTime =  float(p2.stdout.read())
        stop = (time.time() - start)
        results[filename]['xyz-cube'][ri] = readTime
        results[filename]['xyz-cube-all'][ri] = stop

    #Save the test results to file
for filename in filelist :
```

```
        infilename = os.path.split( filename )[1]
        outfile = outfolder+infilename+"_timings.txt"

        f = open( outfile , 'w' )
        for colName in results[filename].dtype.names :
            f.write( colName+" " )
        f.write("\n")
        np.savetxt( f , results[filename] )
        f.close()

    exit(0)


def printHelp():
    """Print the help explaining the usage of testHDF5Optimiation"""

    print "USAGE: Call \"testhdf5_file_read resultsFile\" "
    print "Execute query: testhdf5_file_read filename xmin xmax ymin
                ymax zmin zmax"


if __name__ == "__main__":
    main()
```

7. Matlab script for testing remote read performance

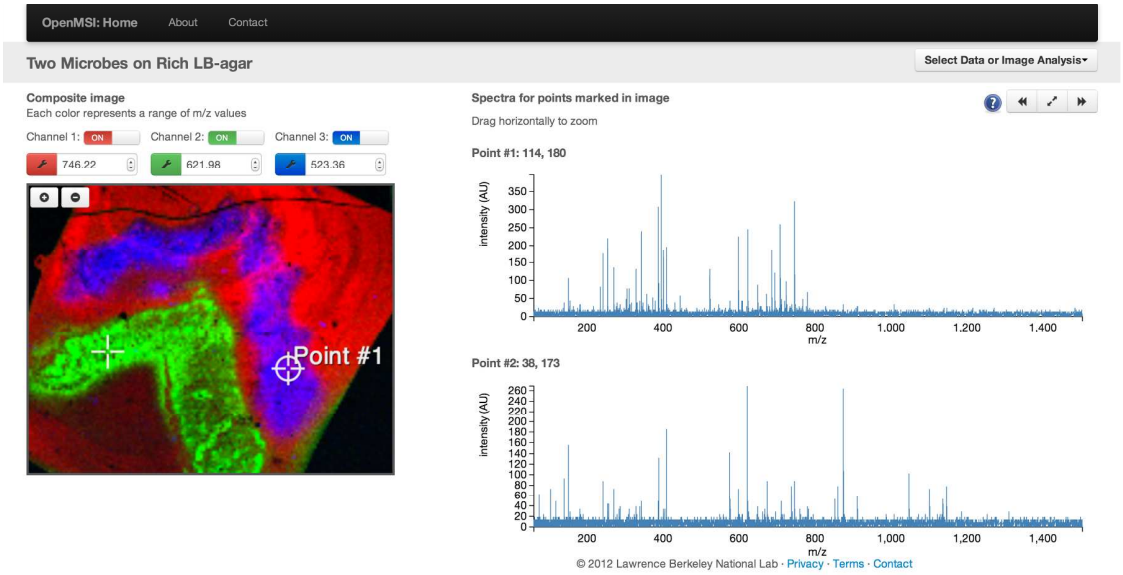
```
clear all
close all
clc

%% test cases
clc
N = 20000; %this is the number of tests to perform
file{1} = '20120711_Brain.h5';
file{2} = '2012_0403_KBL_platename.h5';
file{3} = '20111207_KBL_Roots_BigChip_SmallRoots.h5';
t1 = tic
for iii = 1:length(file)
    str =
sprintf('https://openmsi.nersc.gov/openmsi/qmetadata/?file=/data/openmsi
/omsi_data/%s&expIndex=0&mtype=experimentFull',file{iii})
    [s status] = urlread(str);
    s = loadjson(char(s));
    OutData{iii}.dataShape = s.data_0.shape;
    % dimension an empty matrix to store the time, status, x-coordinate,
    % and y-coordinate of each spectrum requested from the server
    OutData{iii}.spectraTimes = zeros(N,4);
    for i= 1:size(OutData{iii}.spectraTimes,1)
        idx1 = round(rand*(OutData{iii}.dataShape(1)-1));
        idx2 = round(rand*(OutData{iii}.dataShape(2)-1));
        str =
sprintf('https://openmsi.nersc.gov/openmsi/qspectrum/?file=%s&expIndex=0
&dataIndex=0&x=%d&y=%d&findPeaks=0&format=JSON',file{iii},idx1,idx2);
        tic % start the timer
        [s status] = urlread(str);
        t = toc; % stop the timer
        OutData{iii}.spectraTimes(i,1) = t;
        OutData{iii}.spectraTimes(i,2) = status;
        OutData{iii}.spectraTimes(i,3) = idx1;
        OutData{iii}.spectraTimes(i,4) = idx2;
        disp(['Message ',num2str(i),' received in
',num2str(spectraTimes(i)),' seconds']);
```

```
    end
    % dimension an empty matrix to store the time, status, minimum m/z,
and
    % maximum m/z for each requested image.
    % Each image is a maximum intensity projection across 10 mass bins.
    OutData{iii}.sliceTimes = zeros(N,4);
    for i= 1:size(OutData{iii}.sliceTimes,1)
        idx = round(rand*(OutData{iii}.dataShape(3)-50));
        str =
sprintf('https://openmsi.nersc.gov/openmsi/qslice/?file=%s&expIndex=0&da
taIndex=0&z=%d:%d&format=JSON&reduction=max',file{iii},idx,idx+10);
        tic % start the timer
        [s status] = urlread(str);
        t = toc; % stop the timer
        OutData{iii}.sliceTimes(i,1) = t;
        OutData{iii}.sliceTimes(i,2) = status;
        OutData{iii}.sliceTimes(i,3) = idx;
        OutData{iii}.sliceTimes(i,4) = idx+10;
        disp(['Message ',num2str(i),' received in
',num2str(OutData{iii}.sliceTimes(i,1)),' seconds']);
    end
end
t2 = toc(t1)
%% Check if any of our tests failed
for i = 1:3
    sum(OutData{i}.spectraTimes(:,2)==0)
    sum(OutData{i}.sliceTimes(:,2)==0)
end
%% histogram the results of the timed events
edges = 0:1:400; % # milliseconds for binning the data
idx = [1 2 3]
for i = 1:length(idx)
    [y x] = hist([OutData{idx(i)}.spectraTimes(:,1)
OutData{idx(i)}.sliceTimes(:,1)]*1000,edges);
    subplot(1,3,i)
    bar(x,y,1)
    xlim([0 max(edges)])
    title(num2str(OutData{idx(i)}.dataShape))


set(gca,'fontsize',20,'fontweight','bold','linewidth',2,'fontName','cour
ier')
    legend('Spectra (single pixel)','Image (MIP of 10 m/z bins)')
    xlabel('time (msec)');
    ylabel('#Queries');
end
```

8. OpenMSI Viewer



The above figure shows a screenshot of the OpenMSI web-based viewer application showing the ion-image viewer on the left and the spectrum plots for two selected locations (marked by cross-hair cursors) on the right . Using the website a user can interactively explore large-scale MSI data files stored remotely at NERSC. For more details see the http://openmsi.nersc.gov/openmsi/client/. A detailed description of the OpenMSI web-based viewer is beyond the scope of this manuscript and will be described in detail in a forthcoming manuscript.