Leading magnetic fusion energy science into the big-and-fast data lane

Ralph Kube‌⁹*, R Michael Churchill⁴, Jong Youl Choi⁵, Ruonan Wang⁶, Scott Klasky⁸, CS Chang⁴, Minjun J. Choi‡, Jinseop Park‡

https://youtu.be/rih7Hp9nPvM

Abstract—We present Delta, a Python framework that connects magnetic fusion experiments to high-performance computing (HPC) facilities in order to leverage advanced data analysis for near real-time decisions. Using the ADIOS I/O framework, Delta streams measurement data with over 300 MByte/sec from a remote experimental site in Korea to Cori, a Cray XC-40 supercomputer at the National Energy Energy Research Scientific Computing Centre in California. There Delta dispatches cython data analysis kernels using an mpi4py PoolExecutor in order to perform a spectral data analysis workflow. Internally Delta uses queues and worker threads for data communication. With this approach we perform a common spectral analysis suite on imaging measurements more than 100 times faster than with a single-core implementation.

Index Terms—streaming analysis, mpi4py, queue, adios, HPC

Magnetic Fusion Energy research and its data analysis needs

Research on magnetic fusion energy combines physics, engineering, and even economics to deploy a virtually unlimited, clean, and competitively priced energy source to the grid. Python is well established in the fusion community through projects like plasmaPy [PPY] or OMFIT [Men15]. We introduce another Python library for fusion energy research, Delta - the Adaptive Near-real time Analysis framework - and show how it can be used to stream data from an experiment to a remote high performance computing (HPC) resource [git].

There, Delta executes a routine spectral analysis workflow in near real-time. By making data analysis results available in near real-time, Delta allows scientists to make more informed decisions on follow-up experiments and could accelerate scientific discovery. To illustrate the use-case for Delta in fusion energy research, we start with a primer of fusion energy, introduce tokamak devices that are used to perform fusion experiments, describe a diagnostic that is installed in many tokamaks. With this at hand, we describe how near real-time data analysis can be used to accelerate experimental fusion energy workflows.

If one could harvest the energy from controlled nuclear fusion reactions you would have a potentially unlimited, environmentally friendly energy source. Fusion reactions release energy when two light nuclei merge into a heavier one. As part of the reaction, a fraction of the reactants nuclear binding energy is converted into kinetic energy of the products. Fission reactions on the other hand, which power today’s nuclear power plants, release binding energy when a heavy nucleus decays into lighter products. Typical energies involved in nuclear reactions are measured in MeV, multiple orders of magnitude larger than the characteristic eV energy scale for chemical reactions. Thus, the energy yield for a nuclear reaction is much larger than for chemical reaction, which occur when fossil fuels are burnt. Fuel for fusion reactions are readily extracted from sea water, which is available in virtually inexhaustible quantities. Since the energy yield of a fusion reaction is so large, only little fusion plasma needs to be confined to power a fusion reactor. To produce 1 GW of fusion power, enough to power about 700,000 homes, just 2 kg of fusion plasma would need to be burned per day [Ent18]. Thus, a catastrophic event such as total loss of plasma confinement can cause no more than local damage to the plasma vessel.

To fuse positively charged atoms into one heavier requires enormous energy. For the most feasible fusion reactions, Deuterium-Tritium, temperatures upwards of 100 million degrees are required. Such a requirement unfortunately excludes any material container to confine a fusion fuel. The most promising approach is to confine the fusion fuel in the state of a plasma - a hot gas where the atoms are stripped of their electrons. Such a plasma can be confined in a strong magnetic field, shaped like a donut. Confined like this, there is no possibility for an uncontrolled chain reaction. If a significant amount of plasma would leak out of the vessel, the accompanying temperature drop would stop any fusion reactions. At the same time there are only a few grams of plasma confined and it does not have enough stored energy to cause damage other than to the structure of the confinement vessel.

The best performing plasma confinement devices, tokamaks, have a toroidal shape, similar to a donut. Tokamaks (a transliteration of the Russian acronym for toroidal chamber with magnetic coils), such as KSTAR [KSTAR] have a major radius R=1.5m and a minor radius a=0.2-0.7m. In experiments at these facilities, researchers configure parameters such as the plasma density or the shaping and strength of the magnetic field and study the behaviour of the plasma in this setup. During a typical experimental workflow, about 20-30 plasma discharges, so-called shots are performed on a given day where each shot lasts for
a couple of seconds up to minutes. Numerous measurements of the plasma and the mechanical components of the tokamak are performed during each discharge. After a cool-down phase of a few minutes (tokamaks contain cryogenic components) the device is ready for the next shot.

A common diagnostic in magnetic fusion experiments is a so-called Electron Cyclotron Emission (ECE) diagnostic [Cos74]. They measure emission intensity by free electrons in the plasma, which allows one to infer their temperature as a function of radius. Physical models of the plasma describe it partially through the temperature. This measurement allows one to interpret the experiment in terms of such models. Modern ECE systems, such as the one installed in the KSTAR tokamak [Yun14] have hundreds of spatial channels and sample data on a microsecond time-scale, producing data streams upwards of 500 MB/sec.

Analyzing large datasets, as produced by ECE diagnostics in between shots and generating actionable information in time for the next shot is a challenging task. The Delta framework aims to facilitate the analysis of such large datasets in near real-time." This use-case falls in between two other common data analysis workflows in fusion energy research, listed in Tab. 1. Real-time control systems for plasma control require data on a millisecond time scale. This time scale is a hard constraint and limits the amount of data the algorithms can ingest. Post-shot batch analysis of measurements on the other hand serves scientific discovery, such as extraction about the plasma from ECE data. The data and the analysis methods are selected on a per-case basis and are often performed manually hours, days, weeks, months, or years after the experiment has concluded. A goal of Delta is to facilitate scientific discovery at time-scale faster than the experimental cadence. Providing timely analysis results of plasma measurements to experimentalists aids them in making informed decisions about the next plasma shot. As an example of the workflows that we wish to facilitate with Delta we refer to a series of experiments performed at the TAE facility [Bal17].

Table 1: Time-scales on which analysis results of fusion data is required for different tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Time-scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>real-time control</td>
<td>millisecond</td>
</tr>
<tr>
<td>live/inter-shot analysis</td>
<td>seconds, minutes</td>
</tr>
<tr>
<td>scientific discovery</td>
<td>hours, days, weeks</td>
</tr>
</tbody>
</table>

Different measurements. HPC environments also differ for example in their local area network topologies, the speed of network links between data-transfer nodes to compute node, compute node interconnects, and their network security policies. Furthermore granted allocations of compute time for research projects make it impractical to start with a top-down approach that generalizes well to arbitrary HPC platforms (though we endeavor to build the framework with flexibility and extensibility in mind). In the remainder of this section we describe the data analysis workflow for ECEI data, the targeted network and deployment architecture and give an overview of how Delta connects them together.

**Electron Cyclotron Emission Imaging**

The Electron Cyclotron Emission Imaging diagnostic installed in KSTAR measures the electron temperature $T_e$ on a 0.15m by 0.5m grid, resolved using 8 horizontal and 24 vertical channels [Yun10], [Yun14]. Each individual channel produces an intensity time series $I_{h,v}(t)$ where h and v index the horizontal and vertical channel number and $t = i\Delta t$ denotes the time where the intensity is sampled with $\Delta t \approx 1\mu s$ being the sampling time. Digitized with a 16-bit digitizer, this diagnostic produces a data stream of 1836 MByte/sec. The spatial view of this diagnostic covers a significant area of the plasma cross-section which allows it to directly visualize the large-scale structures of the plasma. Besides analyzing the normalized intensity, several quantities calculated off the Fourier transformed intensity $X(\omega)$, here $\omega$ denotes the angular frequency, are used to study the plasma dynamics. The cross-power S, the coherence C, the cross-phase P and the cross-correlation R are defined respectively for channel pair combinations of Fourier transformed intensity signals X and Y as

\[
S_{xy}(\omega) = E[X(\omega)Y^*(\omega)],
\]

\[
C_{xy}(\omega) = |S_{xy}(\omega)|/\sqrt{S_{xx}(\omega)}/\sqrt{S_{yy}(\omega)},
\]

\[
P_{xy}(\omega) = \arctan(Im(S_{xy}(\omega))/Re(S_{xy}(\omega)),
\]

\[
R_{xy}(t) = \text{IFFT}(S_{xy}(\omega)).
\]

Here E denotes an ensemble average, $^*$ denotes complex conjugation, Re and Im denote the real and imaginary part of a complex number and IFFT denotes the inverse Fourier transform. In practice we use a short-time Fourier transformation (STFT) which averages the Fourier coefficients obtained from FFTs calculated on slightly shifted time windows. Spectral quantities calculated off local $T_e$ fluctuations, such as the cross coherence or the cross phases, are used to identify macro-scale structures, so called magnetic islands, as well as micro-scale instabilities in the plasma [Cho17]. Understanding the physics resulting in magnetic islands is important for plasma confinement, and avoiding sudden loss of plasma control, known as a disruption.

**Targeted HPC architecture**

We implement Delta for streaming data from KSTAR to the National Energy Research Scientific Computing Centre (NERSC). NERSC operates Cori [cori], a Cray XC-40 supercomputer that is comprised of 2,388 Intel Xeon “Haswell” processor nodes, 9,688 Intel Xeon Phi “Knight’s Landing” (KNL) nodes and ranks 16 on the Top500 list [top500]. Figure 1 illustrates the targeted network topology. Data transfers from KSTAR and NERSC originate and end at their respective Data Transfer Node (DTN). DTNs
KSTAR to Cori. A generator uses three separate software components to connect Cori's individual compute nodes are connected with a Cray Aries. NERSC, the DTN is connected to Cori with dual 10 Gbit/s NICs. 100Gbit/s links connect both DTNs via the internet. At Table 2 lists the hardware of the DTNs and Cori's compute datacenter transfer. Switching the engine used by ADIOS is trivial to use the DataMan engine for both, trans-oceanic data and intra-

TABLE 2: Hardware and network interconnections of the data transfer nodes (DTNs) and compute nodes

<table>
<thead>
<tr>
<th>Where</th>
<th>CPU</th>
<th>RAM</th>
<th>NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>KSTAR DTN</td>
<td>Xeon E5-2640 v4</td>
<td>128GB</td>
<td>100 Gbit</td>
</tr>
<tr>
<td>NERSC DTN</td>
<td>Xeon E5-2680 v2</td>
<td>128GB</td>
<td>2 * 100 Gbit</td>
</tr>
<tr>
<td>Cori compute</td>
<td>Xeon E5-2698 v3</td>
<td>32 threads</td>
<td>128GB Cray Aries</td>
</tr>
</tbody>
</table>

are servers deployed and configured specifically for transferring data over networks [dtn]. As such, they feature large-bandwidth network interfaces, both for internal and external connections. Table 2 lists the hardware of the DTNs and Cori’s compute nodes. 100Gbit/s links connect both DTNs via the internet. At NERSC, the DTN is connected to Cori with dual 10 Gbit/s NICs. Cori individual compute nodes are connected with a Cray Aries interconnect, peaking at > 45 TB/s.

Connecting science experiments to HPC resources

Delta uses three separate software components to connect KSTAR to Cori. A generator running on the KSTAR DTN, a middleman running on the NERSC DTN, and a processor running on Cori. To avoid performance hits when touching the filesystem, Delta streams data between its components. By using only in-memory data transfers this avoids performance hits when touching the file system. Performance variability of file-based I/O on HPC systems is however a well studied problem [Xie12] . Furthermore, the workflow at hand and anticipated workflows require frequent transfer of relatively small data packets. By relying on streaming data transfers, Delta aims to avoid jitter due to I/O bottlenecks.

The generator stages measurement data and sends it to the middleman. The middleman forwards the data to the processor. The processor receives the data, executes the appropriate analysis kernels and stores the analysis results. Delta uses ADIOS [adios] to facilitate high bandwidth streaming on the paths marked with orange arrows in 1. ADIOS is a unified input/output system that transports and transforms groups of self-describing data variables across different media with performance as a main goal. It is commonly used in the Department of Energy for high performance parallel I/O. Its transport interface is step-based, which resembles the generation of scientific data. ADIOS implements multiple transport mechanisms as engines which take advantage of underlying network communication mechanisms to provide optimal performance. For the topology at hand, Delta configures ADIOS to use the DataMan engine for both, trans-oceanic data and intra-datacenter transfer. Switching the engine used by ADIOS is trivial and requires only the change of a single line in a configuration file.

Implementation details

After providing an overview of the Delta framework and introducing its components in the previous section we continue by describing their implementation in this section and benchmark their performance in the following section.

Components of the Delta framework

We continue by discussing the architecture of the generator and the processor shown in Fig. 2. Here we omit the middleman as it only relays the data stream.

The generator is a single-threaded application that sources data using a loader instance which handles all diagnostic specific data transformations and stages it for streaming. For the ECEI diagnostic the transformations includes for example calculating a channel-dependent normalization and the aggregation of data into time chunks, N_t, blocks of s_t consecutive samples. A writer instance handles all streaming I/O using ADIOS. Pseudo-code for the generator looks like this:

```
def consume(Q, task_list):
    while True:
        try:
            msg = Q.get(timeout=timeout_in_secs)
        except queue.Empty:
            break
        task_list.submit(msg)
        Q.task_done()
```

Here, cfg is a framework-wide json configuration file. Diagnostic-specific parameters are stored in the ECEI section. ADIOS parameters for the writer are stored in the transport_tx section. Since all data transformations are applied inside the loader class the generator appears to be agnostic about what kind of diagnostic data it gets. However, the number of data batches is specific to the data at hand. Furthermore, the pseudo-code example above demonstrates the step-centered design of the ADIOS library. It encapsulates each time chunk in a single time step.

The middleman runs on the NERSC DTN and relays data streams from the generator to the processor. Using the classes available in Delta the pseudo-code looks similar to the generator with a reader instance replacing the loader..

The processor runs on Cori. It reads time chunks from an ADIOS stream, publishes them in a queue and submits analysis tasks to a pool of worker threads. As illustrated in Fig. 2 a reader object receives time chunks data and passes them to a task_list instance, which submits all data analysis routines on the executors. Pseudo-code for the processor looks like this:

```
def main():
    executor_fft = MPIPoolExecutor(max_workers=NF)
    executor_anl = MPIPoolExecutor(max_workers=NA)
    a2_reader = reader(cfg["transport_rx"])
    a2_reader.Open()
    task_list = task_list_spectral(executor_anl, executor_fft, cfg)
    dq = Queue.Queue()
```
LEADING MAGNETIC FUSION ENERGY SCIENCE INTO THE BIG-AND-FAST DATA LANE

**Fig. 1:** The network topology for which the Delta framework is designed. Data is streamed in the direction indicated by the orange arrow. At KSTAR, measurement data is staged from its DTN to the NERSC DTN. Internally at NERSC, the data stream is forwarded to compute nodes at Cori and analyzed. Orange arrows mark sections of the network where ADIOS facilitates high-performance streaming. Black arrows denote standard TCP/IP connections.

**Fig. 2:** Schematic of the Delta framework. The generator runs at the data staging site and transmits time chunks via the ADIOS channels SSSS_ECEI_NN. Here SSSS denotes the shot number and NN enumerates the ADIOS channels. The processor runs at the HPC site, receives the data and submits it for processing through a task_list.

```python
21 workers = []
22 for _ in range(n_thr):
23     w = threading.Thread(target=consume,
24                           args=(dq, task_list))
25     w.start()
26     workers.append(w)
27
28 while True:
29     stepStatus = reader.BeginStep()
30     if stepStatus:
31         stream_data = a2_reader.Get(varname)
32         dq.put_nowait((stream_data,
33                         reader.CurrentStep()))
34         reader.EndStep()
35     else:
36         break
37     worker.join()
38     dq.join()
```

To access the many cores available, processor is launched as an MPI program under control of mpi4py.futures:srun -n NP -m mpi4py.futures processor.py. The mpi4py documentation suggests to run as mpiexec -n 1 -usize NP processor.py but unfortunately Cori’s job system supports neither mpiexec nor defining the universe size by environment variables. The number of MPI ranks should be approximately equal to the workers requested in the PoolExecutors, NP == NF + NA - 1.

Then a2_reader is instantiated with `cfg[transport_rx]`, mirroring the configuration of the writer. After defining a queue for inter-process communication, a group of worker threads is started. In the main loop a2_reader consumes incoming time chunk data from the ADIOS stream and enqueues them. At the same time, the array of worker tasks dequeues time chunks data and passes it to the task_list.

The data analysis code is implemented by cython kernels which are described in a later subsection. While the low-level implementation of Eqs. (1) - (4) is in cython, Delta encapsulates them by the `task` class. Sans initialization the relevant class interface is implemented as

```python
1 class task():
2     ...
3 def calc_and_store(self, data, **kwargs):
4     result = self.kernel(data, **kwargs)
5     self.storage_backend.store(result, tidx)
6 def submit(self, executor, data, tidx):
7     ...
8     _ = [executor.submit(self.calc_and_store, data,
9                          ch_it, tidx)
10        for ch_it in self.get_dispatch_sequence()]
```

The call of an analysis kernel happens in `calc_and_store`. Once the kernel returns, the analyzed data is immediately stored. This allows us to submit a large number of analysis task in parallel in a fire-and-forget way. Implementing analysis and storage as separate functions would introduce dependencies between futures returned by `executor.submit`. Grouping analysis and storage together guarantees that once `calc_and_store` returns, the data has been analyzed and stored. In order to minimize data communication, `submit` launches `calc_and_store` for an...
exhaustive list of channel pair combinations which is accessed by get_dispatch_sequence().

Since the ECEI analysis tasks for the workflow at hand expects Fourier transformed data, the analysis kernels are called sequentially right after the Fourier transformed data becomes available. This logic is implemented by the task_list class:

```python
from scapy.signal import stft

class task_list:
    def submit(self, data, tidx):
        fft_future = self.executor_fft.submit(stft, data, **kwargs)
        for task in self.task_list:
            task.submit(self.executor_anl, fft_future.result(), tidx)
```

Executing the analysis tasks after the Fourier transformation further reduces interdependencies in the workflow, i.e. this implementation awaits only a single future. Without collecting the analysis tasks in a list one may for example execute Fourier transformations prior to launching each individual analysis kernel. This particular choice would increase the number of Fourier transformations by a factor of four and may seem like a poor choice. On the other hand, would this result in less communication across the MPI ranks and may perform better in situations where communication between MPI ranks becomes a bottleneck.

**Explored alternative architectures**

Delta relies on the futures interface defined in PEP 3148 to launch data analysis kernels on an HPC resource [PEP3148]. Since both Cori and ADIOS are designed for MPI application we use the mpi4py [mpi4py] implementation. Being a standard interface, other implementations like concurrent.futures can readily be used. The Python Standard Library defines the interface as executor.submit(fn, *args **kwargs).

Delta wraps PEP 3148 submit calls in wrapper methods of the task and task_list class in order to pass kernel-dependent keyword arguments and in order to facilitate more flexible launch configuration on multiple executors.

Besides mpi4py we explored executing task.calc_and_store calls on a Dask [dask] cluster. Exposing concurrent.futures-compatible interface, both libraries can be interchanged with little work. Running on a single node we found little difference in execution speed. However once the dask-distributed cluster was deployed on multiple nodes we observed a significant slowdown due to network traffic overhead. We did not investigate this problem any further.

As an alternative to using a queue with threads, we also explored using asynchronous I/O. In this scenario, the main task would define a coroutine receiving the data time chunks and a second one dispatching them to an executor. In our tested implementation, the coroutines would run in a main loop and communicate via a queue. Our experiments showed no measurable difference against a threaded implementation. On the other hand, the threaded implementation fits more naturally in the multi-processing design approach.

**Using data analysis codes** Delta

In a broad sense, data analysis can be described as applying a transformation $F$ to some data $d$,

$$ y = F(d; \lambda_1, \ldots, \lambda_n), \quad (5) $$
given some parameters $\lambda_1, \ldots, \lambda_n$. Translating the relation between the $F$ and $d$ into an object-oriented setting is not always straightforward and one needs to have the application in mind when designing a library. The approach taken by general-purpose packages such as scipy or scikit-learn is to implement a transformation $F$ as a class and interface to data through its member functions. Taking Principal Component Analysis in scikit-learn as an example, the default way of applying it to data is

```python
from sklearn.decomposition import PCA
X = np.array([[...]])
pca = PCA(n_components=2)
pca.fit_transform(X)
```

This approach has proven itself useful and is the common way of organizing libraries. Delta deviates slightly from this approach and calls transformations in the calc_and_store member function of the task_ecei class. The specific kernel to be called is configured in the objects initialization:

```python
from kernels import kernel_crossphase, ...

class task():
    def __init__(self, cfg): ...
    if (cfg["analysis"] == "cross-phase"):
        self.kernel = kernel_crossphase
    elif (cfg["analysis"] == "cross-power"):
        self.kernel = kernel.crosspower ...

def calc_and_store(self, data, ...):
    self.kernel(data, ...) ...
```

At the time of writing, Delta only implements a workflow for ECEI data and this design choice minimizes the number of classes present in the framework. Grouping the data analysis methods by diagnostic also allows to execute diagnostic-specific pre-transformations that are best performed after transfer to the processor collectively. One may wish for example to distribute calculations of the 18336 channel pair combinations among multiple task instances. This approach lets us seamlessly do that. Once the requirements and use cases have stabilized we will explore suitable generalizations such as object factories for the task_list class.

In summary, the architecture of Delta implements data streaming using time-stepping interface of ADIOS and data analysis using PEP 3148 compatible executors. In order to increase performance we choose to use two PoolExecutors. The first executor is used to execute short Fourier Transformations of the the input data for the entire analysis task group. The second pool executor is available for running the analysis kernels and immediate storage of the results.

**Performance analysis**

While the overall performance of the framework can be measured by the walltime of the analysis workflow at hand, the complex composition of the framework requires us to understand the performance of its building blocks. Referring to figure 2, IO performance of the ADIOS library, the asynchronous receive-publish-submit strategy implemented by processor and finally the speed of individual analysis kernels contribute to the workflow walltime. Furthermore, the workflow walltime may be sensitive to the individual components interacting with one another. For example, even though the processor design aims to facilitate high-velocity data streams by using queues and multiple worker threads, a fast data stream ingested by the processor may negatively affect
the performance of the PoolExecutors by submitting too many tasks in a short time. It may well be that slower data streaming rate result in a smaller workflow walltime. Given these considerations we start be investigating the performance of individual components in this section and finally investigate the performance of the framework on the ECEI workflow.

**Performance of the WAN connection**

As a first step we measure the practically available bandwidth between the KSTAR and NERSC DTNs using the network performance tool iperf3 [iperf]. Multiple data streams are often necessary to exhaust high-bandwidth networks. Varying the number of senders from 1 to 8, we measure data transfer rates from 500 MByte/sec using 1 process up to a peak rate of 1500 MByte/sec using 8 processes, shown in Figure 3. Using 1 thread we find that the data transfer rate is approximately 500 MByte/sec with little variation throughout the benchmark. Running the 2 and 4 process benchmark we see initial transfer rates of more than 1000 MByte/sec. After about 5 to 8 seconds, TCP observes network congestion and falls back to fast recovery mode where the transfer rates increase to the approximately the initial transfer rates until the end of the benchmark run. The 8 process benchmark shows a qualitatively similar behaviour but the congestion avoidance starts at approximately 15 seconds where the transfer enters a fast recovery phase.

While we measured the highest bandwidth when transferring with 8 process, Delta currently only implements single process data transfers.

**Data Analysis Kernels**

As seen in the code-example above, Delta implements data analysis routines as computational kernels. These are implemented in cython to circumvent the global interpreter lock and utilize multiple cores. Measuring the average execution time over 10 runs on a Cori compute node we find that the kernels demonstrate a strong scaling for up to 16 threads, shown in Fig. 4. Using more 32 threads results in sub-linear speedup.

**Performance of the ECEI workflow**

Having established the performance of the individual components we continue by benchmarking the performance of Delta performing the entire ECEI analysis workflow. The task at hand is to calculate Eqs. (1) - (4) for 18836 unique channel pair combinations per time chunk. Each time chunk consists of \( s_{ch} = 10,000 \) samples for 192 individual channels. A total of \( N_{ch} = 500 \) time chunks are to be processed, for a total of about 5 GByte of data.

The performance of Delta depends on the individual performance of multiple components, such as the data streaming velocity, lag introduced by using queue inter-process communication, spawning processes on the executors, MPI communication as well as their interplay with one another. Having benchmarked individual components in the previous section, we now benchmark the runtime of Delta performing the ECEI spectral analysis workflow in three scenarios. In the file scenario, the processor reads data from a local ADIOS file. No data is streamed. In the 2-node scenario, data is streamed from the generator running on the NERSC DTN to Cori. In the 3-node scenario, data is streamed from the KSTAR DTN to the NERSC DTN and forwarded to Cori - this is the scenario shown in 2. Both the 2- and 3-node scenario use ADIOS DataMan engine for data streaming. All runs are performed on an allocation using 32 Cori nodes partitioned into 128 MPI ranks with 16 Threads each for a total of 2048 CPU cores.

Table 3 lists the Walltime and the number of processed time chunks \( N_{ch} \) and the utilized bandwidth. Walltime refers to the walltime as measured by the processor and \( N_{ch} \) gives the number of time chunks analyzed by the processor. The utilized bandwidth refers to the I/O speed achieved when reading from disk in the file scenario, the average data transfer rate from the NERSC DTN to Cori in the 2-node scenario and as the average data transfer rate from the KSTAR DTN to the NERSC DTN in the 3-node scenario.

The measured walltime for the file-based workflow is 352s, 358s for the 2-node scenario and 339s for the 3-node scenario. Only minor packet loss occurs using the current implementation of the DataMan engine. In order to mitigate packet loss the generator pauses a tenth of a second after sending any packet from the NERSC DTN to Cori, resulting in a bandwidth of 95...
MByte/sec for the 2-node scenario. In the 3-node scenario we show that Delta can ingest high velocity data streams from KSTAR to NERSC and perform analysis on them. As in the 2-node scenario, we limit the bandwidth from the NERSC DTN to Cori by pausing a fraction of a second before relaying a time chunk. On average, Delta performs the entire analysis workflow as fast in a streaming setting as it does when reading from the local file system. The average time to analyze a single time chunk is about 0.7 seconds, independent of the workflow.

Figure 5 shows the amount of time that data for a given time chunk, \( n_{\text{ch}} = 1 \ldots N_{\text{ch}} \), spends in the queue of the processor. All three scenarios show a similar trend - the amount of time a time chunk spends in the queue increases with the time when it is enqueued. This suggests that data is streamed faster to the processor than the MPI ranks perform data analysis. This implies that the queue acts as a cache for the incoming time chunk data. Running the file scenario, the processor loads data almost immediately after it starts up. For the 2-node and 3-node scenarios the start time of the components on their respective machines is not coordinated. This causes the first time chunk data to arrive at varying times for the three scenarios.

As time chunks are dequeued, they are subject to a STFT. Figure 6 denotes the time where the STFT for each time chunk data is executed with horizontal bars. The beginning of a horizontal bar indicates where the STFT with the time chunk data is submitted on \texttt{executor_fft} and the end of a bar marks the time STFT is finished. Common for all three scenarios is that the STFTs with the longest execution time are the ones for the first time chunks received. Also, the majority of the STFTs is executed in approximately one second. Equivalent STFT evaluations outside Delta take about 0.15s on Cori. On average the STFT when called from the streaming workflow is slower by a factor of 6. We believe that this long execution time is in part explained by MPI communication overhead.

Finally, Figures 7, 8 and 9 show the utilization of the MPI ranks over time. The MPI ranks execute the STFT and analysis kernels, the figures only show the time where analysis kernels are executed. All three scenarios show a low usage of available MPI ranks, approximately 16 - 20 in the beginning of the run. After all time chunks are dequeued and Fourier transformed, all available MPI ranks are used. Color encodes the different analysis kernels. For example, green bars show time at which a cross-correlation kernel is executed. The majority of the computation time is consumed by cross-correlation kernels. This observation agrees with the performance analysis that showed that the cross-correlation kernel is the most time consuming.

Conclusions and future work

We demonstrate that Delta can facilitate near real-time analysis of high-velocity streaming data. In our experiments we achieved streaming rates of about 350 MByte/sec and execute a spectral analysis workflow on ECEI measurements in less than 4 minutes. Performing the analysis in the streaming scenario, illustrated
in Figure 1, comes with only a negligible performance impact as compared to using local filesystem IO. ADIOS manages to utilize about 70% of the available bandwidth for data streaming from KSTAR to NERSC in the streaming analysis workflow. mpi4py PoolExecutors facilitate an flexible execution of work items on Cori, as required for our workflow where data arrive at high velocity. Furthermore, python queues reliably facilitate inter-process communication and act as a data cache under the tested IO loads.

In the current form, there are multiple shortcomings of the framework that need to be addressed. Firstly, the DataMan engine received an experimental feature to mitigate packet loss. Secondly, implementation details of MPI on Cori limit us to effectively a single PoolExecutor. We are planning to investigate this more closely and aim to properly separate the execution space of the STFT and the analysis kernels. Thirdly, the framework will be generalized in order to facilitate more data analysis tasks. Finally, we are working on adapting Delta for next generation HPC facilities which heavily rely on graphical processing units to provide processing power.

Another issue we plan to address is to make Delta more adaptive. This includes developing machine learning algorithm for data compression and to decide which data batches are to be offloaded to HPC resources for in-depth analysis. For example, ECEI time chunk data that is not likely to be relevant for magnetic island studies could be analyzed with fast, coarse routines at a local workstation while relevant data could be forwarded to in-depth analysis routine.

Acknowledgements

The authors would like to acknowledge the excellent technical support from engineers and developers at the National Energy Research Scientific Computing Center. This work used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. Delta is available on github: [Git] All data used to generate the plots in this article can be accessed on Zenodo [Zen].

REFERENCES


[top500] @top500supercomp (2019, Nov) We are proud to announce the 54th edition of the TOP500 list! China extends lead in number of TOP500 supercomputers, US holds on to performance advantage. To view the full list, visit https://top500.org/lists/2019/11/ Retrieved from https://twitter.com/top500supercomp/status/119642869339160065


