Fluctuation X-ray Scattering real-time app

Antoine Dujardin[¶], Elliott Slaugther[¶], Jeffrey Donatelli[‡§], Peter Zwart[‖§], Amedeo Perazzo[¶], Chun Hong Yoon[¶*]

https://youtu.be/IYADjGOiJhA

Abstract—The Linac Coherent Light Source (LCLS) at the SLAC National Accelerator Laboratory is an X-ray Free Electron Laser (X-FEL) facility enabling scientists to take snapshots of single macromolecules to study their structure and dynamics. A major LCLS upgrade, LCLS-II, will bring the repetition rate of the X-ray source from 120 to 1 million pulses per second and exascale High Performance Computing (HPC) capabilities will be required for the data analysis to keep up with the future data taking rates.

We present here a Python application for Fluctuation X-ray Scattering (FXS), an emerging technique for analyzing biomolecular structure from the angular correlations of FEL diffraction snapshots with one or more particles in the beam. This FXS application for experimental data analysis is being developed to run on supercomputers in near real-time while an experiment is taking place.

We discuss how we accelerated the most compute intensive parts of the application and how we used Pygion, a Python interface for the Legion task-based programming model, to parallelize and scale the application.

Index Terms—fluctuation x-ray scattering, free electron laser, real-time analysis, coherent diffractive imaging

Introduction

LCLS-II, an LCLS upgrade

The Linac Coherent Light Source (LCLS) at the SLAC National Accelerator Laboratory is an X-ray Free Electron Laser facility providing femtosecond pulses with an ultrabright beam approximately one billion times brighter than synchrotrons [WRD15]. Such a brightness allows it to work with much smaller sample sizes while the shortness allows imaging below the rotational diffusion time of the molecules and also outrunning radiation damage. With pulses of such an unprecedented brightness and shortness, scientists are able to take snapshots of single macromolecules without the need for crystallization at ambient temperature.

To push the boundaries of the science available at the light-source, LCS is currently being upgraded after 10 years of operation. The LCLS-II upgrade will progressively increase the sampling rate from 120 pulses per second to 1 million. At these rates, the LCLS instruments will generate multiple terabytes per second of scientific data and it will therefore be critical to know what data is worth saving, requiring on-the-fly processing of the data. Earlier, users could classify and preprocess their data after the experiment, but this approach will become either prohibitive or plainly impossible. This leads us to the requirement of performing some parts of the analysis in real time during the experiment.

Quasi real time analysis of the LCLS-II datasets will require High Performance Computing, potentially at the Exascale, which cannot be offered in-house. Therefore, a pipeline to a supercomputing center is required. The Pipeline itself starts with a Data Reduction step to reduce the data size, using vetoing, feature extraction, and compression in real time. We then pass the data over the Energy Sciences Network (ESnet) to the National Energy Research Scientific Computing Center (NERSC). Currently, the ESNet connection between SLAC and NERSC is 200 Gbps capable; the plan is to upgrade this link to 400 Gbps by 2026 and to 1 Tbps by 2028. At the end of the pipeline, the actual analysis can take place on NERSC's supercomputers. This makes the whole process, from the sample to the analysis, quite challenging to change and adapt.

Moreover, LCLS experiments are typically high-risk / high-reward and involve novel setups, varying levels of requirements, and durations of only a few days. The novelty in the science can require adaptations in the algorithms, requiring the data analysis itself to be highly flexible. Furthermore, we want to give users as much freedom as possible in the way they analyze their data without expecting them to have a deep knowledge of large-scale computer programming.

Therefore, we require real time analysis, high performance computing capabilities and a complex pipeline, while requiring enough flexibility to adapt to novel experimental setups and analysis algorithms. We believe Python helps us achieve this goal given the tradeoffs involved.

FXS: an example analysis requiring HPC

While a variety of experiments can be performed at LCLS, we focus here on one specific example: Fluctuation X-ray Scattering (FXS).

X-ray scattering of particles in a solution is a common technique in the study of the structure and dynamics of macromolecules in biologically-relevant conditions and gives an understanding of their function. However, traditional methods currently used at synchrotrons suffer from the fact that the exposure time is longer than the rotation time of the particle, leading to the capture of angularly-averaged patterns. FXS techniques fully utilize the femtosecond pulses to measure diffraction patterns from multiple

¶ SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA 94025, USA
‡ Department of Applied Mathematics, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
§ Center for Advanced Mathematics for Energy Research Applications, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
‖ Molecular Biophysics and Integrated Bio-Imaging Division, Lawrence Berkeley National Laboratory, Berkeley, CA USA 94720-8142
∗ Corresponding author: yoon82@slac.stanford.edu

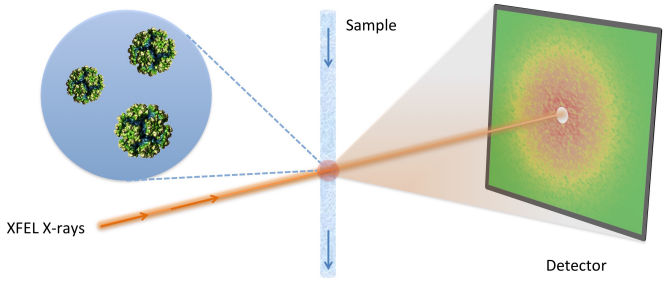

Fig. 1: *Fluctuation X-ray Scattering experiment setup.*

In an FXS experiment, femtosecond pulses from an X-ray Free Electron Laser are shot at a stream of particles in solution. The scattered light forms a diffraction pattern on the detector, aggregating the contributions of the different particles.[1]

identical macromolecules below the sample rotational diffusion times (Fig. 1). The patterns are then collected to reconstruct a 3D structure of the macromolecule or measure some of its properties. This technique was described in the late 1970s [Kam77], [KKB81] and has been widely used at LCLS [PDM+18], [KDY+17], [MLS+14], [MWQ+16].

While a few hundreds of diffraction patterns might be sufficient to reconstruct a low resolution 3-dimensional structure under ideal conditions [KDY+17], the number of snapshots required can be dramatically increased when working with low signal-to-noise ratios (e.g. small proteins) or when studying low-probability events. More interestingly, the addition of a fourth dimension, time, to study dynamical processes expands again the amount of data required. At these points, hundreds of millions or more snapshots could be required.

We present here a Python application for FXS data analysis that is being developed to run on supercomputing facilities at US Department of Energy national laboratories in near real-time while an experiment is taking place. As soon as data is produced, it is passed through a Data Reduction Pipeline on-site and sent to a supercomputer via ESNet, where reconstructions can be performed. It is critical to complete this analysis in near real-time to guide experimental decisions.

In FXS, each diffraction pattern contains several identical particles in random orientations. Information about the structure of the individual particle can be recovered by studying the two-point angular correlation of the data. To do so, the 2D images are expanded in a 3D, orientation-invariant space, where they are aggregated using the following formula:

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N}\int_0^{2\pi} I_j(q,\phi)I_j(q',\phi+\Delta\phi)d\phi \qquad (1)$$

where $I_j(q,\phi)$ represents the intensity of the j-th image, in polar coordinates. This correlator can then be used as a basis for the actual 3D reconstruction of the data (Fig. 3), using an algorithm described elsewhere [DZS15], [PDM+18].

Acceleration: getting the best out of NumPy

The expansion/aggregation step presented in Equation (1) was originally the most computation intensive part of the application, representing the vast majority of the computation time. The original implementation was processing each $I_j(q,\phi)$ image one after the other and aggregating the results. This resulted in taking 424 milliseconds per image using NumPy [Oli06], [vdWCV11] functions and a slightly better performance using Numba [LPS15]. As we illustrate in this section, rewriting this critical step allowed us to gain a factor of 40 in speed, without any other libraries or tools. The tests were performed on a node of Cori Haswell.

Let us start by simplifying Equation (1). The integral corresponds to the correlation over $I_j(q,\phi)$ and $I_j(q',\phi)$. Thanks to the convolution theorem [Arf85], we have

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N}\mathscr{F}^{-1}[\mathscr{F}[I_j(q,\phi)]\overline{\mathscr{F}[I_j(q',\phi)]}], \qquad (2)$$

where $\mathscr{F}$ represents the Fourier transform over ϕ. The inverse Fourier transform being linear, we can get it outside the sum, and on the left side. For the simplicity of the argument, we also neglect all coefficients.

Using ψ as the equivalent of ϕ in the Fourier transform and $A_j(q,\psi)$ as a shorthand for $\mathscr{F}[I_j(q,\phi)]$, we have:

$$C_2(q,q',\Delta\phi) = \frac{1}{2\pi N}\sum_{j=1}^{N} A_j(q,\psi)\overline{A_j(q',\psi)}. \qquad (3)$$

We end up with the naive implementation below:

```
C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    for j in range(N_RAD_BINS):
        for k in range(N_RAD_BINS):
            C2[j, k, :] += A[j] * A[k].conj()
```

taking 42.4 seconds (for 100 images), using the following parameters:

```
N_IMGS = 100
N_RAD_BINS = 300
N_PHI_BINS = 256
IMGS_SHAPE = (N_IMGS, N_RAD_BINS, N_PHI_BINS)
C2_SHAPE = (N_RAD_BINS, N_RAD_BINS, N_PHI_BINS)
```

where `N_RAD_BINS` and `N_PHI_BINS` represent the image dimensions over the q- and ϕ-axes, as well as the dataset:

```
images = np.random.random(IMGS_SHAPE)
```

We note that a typical application would be processing millions of images, but let us use 100 for the example.

This naive version can be slightly accelerated using the fact that our matrix is conjugate-symmetric:

```
C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    for j in range(N_RAD_BINS):
        C2[j, j, :] += A[j] * A[j].conj()
        for k in range(j+1, N_RAD_BINS):
            tmp = A[j] * A[k].conj()
            C2[j, k, :] += tmp
            C2[k, j, :] += tmp.conj()
```

which takes 36.0 seconds. Note that this is only 18% faster, far from a 2x speed-up.

This naive implementation should not be confused with a pure Python implementation, which is expected to be slow, since we already operate on NumPy arrays along the angular axis. Such an implementation could be approximated by:

```
A = np.fft.fft(images[i], axis=-1)
for j in range(N_RAD_BINS):
    for k in range(N_RAD_BINS):
```

```
        for l in range(N_PHI_BINS):
            C2[j, k, l] += A[j, l] * A[k, l].conj()
```

which takes 49.1 seconds per image, i.e. about 100 times slower than the naive implementation, in accordance with the stereotype of Python being much slower than other languages for numerical computing.

A common acceleration strategy is to use Numba:

```
@numba.jit
def A_to_C2(A):
    C2 = np.zeros(C2_SHAPE, np.complex128)
    for j in range(N_RAD_BINS):
        C2[j, j, :] += A[j] * A[j].conj()
        for k in range(j+1, N_RAD_BINS):
            tmp = A[j] * A[k].conj()
            C2[j, k, :] += tmp
            C2[k, j, :] += tmp.conj()
    return C2

C2 = np.zeros(C2_SHAPE, np.complex128)
for i in range(N_IMGS):
    A = np.fft.fft(images[i], axis=-1)
    C2 += A_to_C2(A)
```

which takes 38.5 seconds, i.e. 10% faster than the naive implementation.

When considering our problem size of up to millions of images, processing images one at a time makes sense. However, focusing on a small batch as we have been doing in these examples, a strategy can be to have NumPy and/or Numba work on arrays of images, rather than the individual images. We then have the following:

```
@numba.jit
def As_to_C2(As):
    C2 = np.zeros(C2_SHAPE, np.complex128)
    for i in range(N_IMGS):
        A = As[i]
        for j in range(N_RAD_BINS):
            C2[j, j, :] += A[j] * A[j].conj()
            for k in range(j+1, N_RAD_BINS):
                tmp = A[j] * A[k].conj()
                C2[j, k, :] += tmp
                C2[k, j, :] += tmp.conj()
    return C2

As = np.fft.fft(images, axis=-1)
C2 = As_to_C2(As)
```

which takes 11.9 seconds, i.e. 3.56 times faster. We note also here the batching of the Fast Fourier Transform.

However, such an implementation does not sound trivial using NumPy, although one can recognize a nice (generalized) Einstein sum in Equation (3), leading to:

```
As = np.fft.fft(images, axis=-1)
C2 = np.einsum('hik,hjk->ijk', As, As.conj())
```

which corresponds to expressing `C2[i, j, k]` as the sum over `h` of `As[h, i, k] * As.conj()[h, j, k]`.

This takes 17.9 seconds, which is slower than the version using Numba per batch. However, we can realize that, at this batch level, the last axis is independent from the others and that the underlying alignment of the arrays matters. Thanks to NumPy's `asfortranarray` function, however, that is not an issue. We then use the F-ordered dataset.

```
images_F = np.asfortranarray(images)
```

We observe, for the Einstein sum:

```
As = np.fft.fft(images_F, axis=-1)
C2 = np.einsum('hik,hjk->ijk', As, As.conj())
```

Implementation	Time (/100)	Speedup
Naive	42.4 s	1
Numba	38.5 s	10%
Numba, batched	11.9 s	3.56×
Einsum, F-order	4.05 s	10.5×
Dot, F-order	1.06 s	40.0×

TABLE 1: *Summary of the major time improvements.*

taking 4.05 seconds, i.e. 4.42 times faster than the C-ordered Einstein sum and 10.5 times faster than the naive implementation.

Additionally, it turns out that in our precise case, we can actually express it as a more optimized dot product:

```
As = np.fft.fft(images, axis=-1)
C2 = np.zeros(C2_SHAPE, np.complex128)
for k in range(N_PHI_BINS):
    C2[..., k] += np.dot(As[..., k].T,
                         As[..., k].conj())
```

which now brings us down to 1.37 seconds, i.e. 30.9 times faster than the naive version.

For the F-ordered case, we have:

```
As = np.fft.fft(images_F, axis=-1)
C2 = np.zeros(C2_SHAPE, np.complex128, order='F')
for k in range(N_PHI_BINS):
    C2[..., k] += np.dot(As[..., k].T,
                         As[..., k].conj())
```

taking 1.06 seconds, i.e. 29% faster than the C-ordered case and 40.0 times faster than the naive implementation. We could note that, at that speed, the main computation gets close to the time required to perform the Fast Fourier Transform, which is, in our case at least, faster on C-ordered (107 ms) than F-ordered (230 ms) data. Removing the FFT computation would yield an even starker contrast (977 ms vs. 499 ms), but would neglect the cost of the re-alignment.

In conclusion, and as summarized in Table 1, implementing this algorithm using NumPy or Numba naively gives significant improvement in computational speed compared to pure Python, but there is still a lot of room for improvement. On the other hand, such improvement does not necessarily require using fancier tools. We showed that batching our computation helped in the Numba case. From there, a batched NumPy expression looked interesting. However, it required optimizing the mathematical formulation of the problem to come up with a canonical expression, which could then be handed over to NumPy. Finally, the memory layout can have a sizable impact on the computation, while being easy to tweak in NumPy.

Parallelization: effortless scaling with Pygion

To parallelize and scale the application we use Pygion, a Python interface for the Legion task-based programming system [SA19]. In Pygion, the user decorates functions as *tasks*, and annotates task parameters with *privileges* (read, write, reduce), but otherwise need not be concerned with how tasks execute on the underlying machine. Pygion infers the dependencies between tasks based on their privileges and the values of arguments passed to tasks, and ensures that the program executes correctly, even when running on a parallel and distributed supercomputer.

To enable the distributed execution, it is necessary to separate the question of what data is needed in a given task from the

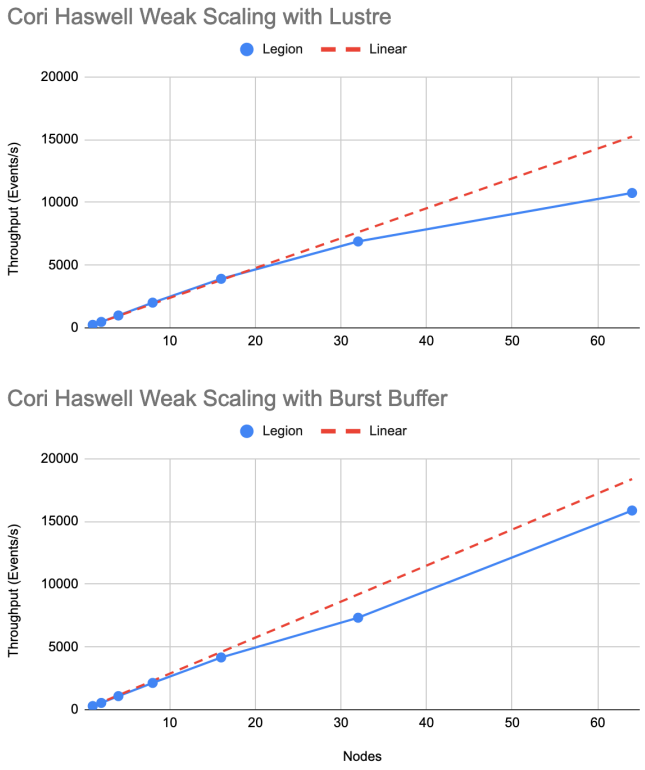

Fig. 2: *Weak scaling behavior on Cori Haswell with Lustre filesystem (top) and Burst Buffer (bottom).*

The application was run on 100,000 images per node, for up to 64 nodes on Cori Haswell. The Lustre filesystem is a high performance system running on HDDs attached to the supercomputer. The Burst Buffer corresponds to SSDs placed within the supercomputer itself used for per-job storage.

allocation of the data in a given memory or memories. This reification of the flow of data between tasks is achieved by declaring *regions*, similar to multi-dimensional Pandas dataframes [McK10]. Regions contain *fields*, each of which is similar to and exposed as a NumPy array. Regions can be partitioned into subregions, which can be processed by different tasks, allowing the parallelism. Note that regions are allocated only when needed, so it is possible (and idiomatic) to allocate a region which is larger than any single machine's memory, and then to partition into pieces that will be used by individual tasks.

We scale up to 64 Haswell nodes on NERSC's Cori supercomputer using Pygion, with 10 to 30 processes per node, to reach a throughput of more than 15,000 images per second, as illustrated in Figure 2. Compared to an equivalent MPI implementation, Pygion is easier to scale out of the box as it manages load-balancing of tasks across cores, shared memory (between distinct Python processes on a node) and provides high-level parallelization constructs. These constructs make it easy to rapidly explore different partitioning strategies, without writing or rewriting any communication code. This enabled us to quickly find a strategy that scales better than the straightforward but ultimately suboptimal strategy that we initially developed.

As an example, the most computationally intensive part of our problem is the $C_2(q,q',\Delta\phi)$ computation discussed in detail in the section above, which can trivially be parallelized over the last (angular) axis. However, the image preprocessing and the Fast Fourier Transform can only be parallelized over the first (image) axis. Given the size of the data, parallelizing between nodes would involve a lot of data movement. Parallelizing within a node, however, could help. In the MPI case, we use MPI to parallelize between nodes and within a node (MPI+MPI). If we were to introduce this optimization into such a code, one would have to create a 2-level structure such as:

```
In each node:
  Define node-level communicator
  In each rank:
    Receive and pre-process some stacks of images
  All-to-all exchange from stacks of images
    to angular sections
  In each rank:
    Process the received angular section
```

where all the data exchange has to be coded by hand.

In the Pygion case, the ability to partition the data allows us to create tasks that are unaware of the extent of the regions on which they operate. We can therefore partition these regions both over the image axis and the angular one. We end up with:

```
@task(privileges=[...])
def node_level_task(...):
    for i, batch in enumerate(data_batches):
        preprocess(input_=batch,
                   output=A_image_partition[i])
    for i in range(NUMBER_OF_PROCESSES):
        process(input_=A_angular_partition[i],
                output=C2_angular_partition[i])
```

where the data exchange is implied by the image-axis partition `A_image_partition` and the angular-axis partition `A_angular_partition` of the same region `A`.

Results

To test our framework, a dataset of 100,000 single-particle diffraction images was simulated from a lidless chaperone (mm-cpn) in its open state, using Protein Data Bank entry 3IYF [ZBS+10]. These images were processed by the algorithm described above to get the 2-point correlation function, $C_2(q,q',\Delta\phi)$, described in Equation (1). This correlation function was first filtered and reduced using the methods described in [PDM+18], and then the reconstruction algorithm in [DZS15] was applied to reconstruct the electron density of the chaperone from the reduced correlations, yielding the reconstruction shown in Figure 3.

To obtain this result, the correlation function was filtered and reduced using the Multi-Tiered Iterative Filtering (M-TIF) algorithm [PDM+18]. In particular, M-TIF uses several iterations of Tikhonov regularization, linear pseudo inversion, and principal component analysis to fit three tiers of expansions to the data: a Legendre polynomial expansion in theta, a Hankel-transformed Fourier-Bessel expansion in q and q', and a low-rank eigenvalue decomposition on the matrices of Fourier-Bessel coefficients. The number of terms needed in each expansion step is limited and determined by an upper-bound diameter estimate of the protein sample. Once these coefficients are determined, their corresponding series expansions are computed to produce a filtered correlation function, along with a reduced set of Legendre polynomial expansion coefficients on a coarse q-grid, which is used in the reconstruction (See [PDM+18] for more details on the filtering).

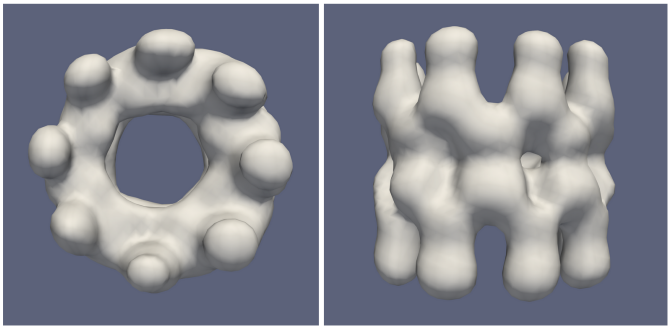

Fig. 3: *Reconstruction of a lidless chaperone (mm-cpn) in its open state from simulated diffraction patterns.*
The 2-point correlation function was computed on the simulated dataset as described in the present document. It was then filtered, reduced, and fed to a reconstruction algorithm described elsewhere [PDM+18], [DZS15] to yield the reconstruction above.

These Legendre expansion coefficients can be directly related to the protein sample. In particular, the coefficients are equal to the inner products of spherical harmonic coefficients of the 3D intensity function, which is defined as the squared magnitude of the Fourier transform of the sample's electron density [Kam77]. This relation can be expressed as two tiers of phase problems that need to be solved to reconstruct the underlying density: a hyperphase problem to recover the intensity function from the Legendre coefficients, and a classical scalar phase problem to recover the density from the intensity. In order to reconstruct the sample, we apply the Multi-Tiered Iterative Phasing (M-TIP) algorithm [DZS15] to the Legendre coefficients computed from the M-TIF filtering/reduction procedure. M-TIP works by using a set of computationally efficient projection operators in a self-consistent iteration to simultaneously solve both tiers of phase problems and reconstruct the sample from the Legendre coefficients.

After acceleration and parallelization, we now reach a throughput of about 230 images per second on a single node of Cori Haswell. This would allow us to process in real time the output of an FXS experiment at LCLS-I, which produces 120 images per second. Such a rapid processing would make possible to give scientists immediate feedback on the quality of their data. After scaling to up to 64 nodes, the throughput of about 15,000 images per second would be sufficient to follow up with the early abilities of LCLS-II, although further acceleration and scaling will be required to match the data being produced as LCLS-II increases its pulse rate dramatically over the following years.

Interestingly, one might note from Equations 1, 2, or 3 that computing the correlation function involves a sum over all the images. The output of that computation, however, no longer depends on the number of images in the dataset. The size of the correlation function $C_2(q,q',\Delta\phi)$ is, therefore, only dependent on the resolution over the q, q', and $\Delta\phi$ axes. As a consequence, the computational complexity of the post-processing of the correlation function and the reconstruction algorithm does not scale with the amount of data being processed.

Conclusion

The Linac Coherent Light Source provides scientists with the ability of X-ray diffraction patterns with much higher brightness and much shorter timescales, allowing experiments not possible elsewhere. With its upgrades LCLS-II in 2021 and LCLS-II-HE (High Energy) in 2025, LCLS experiments will produce up to millions of X-ray pulses per second and generate commensurate amounts of data. In some cases, such as the FXS technique described in this paper, the processing of the dataset will require High Performance Computing at a scale that can no longer be provided in-house.

We showed that Python gives us and our users the flexibility to adapt the analysis pipeline to new experiments. The main drawback of Python is that implementing new algorithms without relying on specialized libraries can be problematically slow. However, we illustrate with our example that spending some time optimizing the math of the problem (rather than the code) and being aware of the strengths and weaknesses of NumPy and Numba can allow us to achieve drastically better performances, without the need to develop or use external libraries.

Finally, we used Pygion to manage the parallelization of the problem, which allows us to design applications that scale much more naturally than MPI at a given level of coding effort, and in particular has allowed us to explore different parallelization strategies more rapidly, leading ultimately to a more scalable solution than what we otherwise might have been able to find.

Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Use of the Linac Coherent Light Source (LCLS), SLAC National Accelerator Laboratory, is supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences under Contract No. DE-AC02-76SF00515.

References

[Arf85] G Arfken. Convolution theorem. In *Mathematical Methods for Physicists*, chapter 15.5, pages 810–814. Academic Press, Orlando, FL, 3 edition, 1985.

[DZS15] Jeffrey J Donatelli, Peter H Zwart, and James A Sethian. Iterative phasing for fluctuation X-ray scattering. *Proceedings of the National Academy of Sciences of the United States of America*, 112(33):10286–91, 2015. doi:10.1073/pnas.1513738112.

[Kam77] Zvi Kam. Determination of Macromolecular Structure in Solution by Spatial Correlation of Scattering Fluctuations. *Macromolecules*, 10(5):927–934, 1977. doi:10.1021/ma60059a009.

[KDY+17] Ruslan P. Kurta, Jeffrey J. Donatelli, Chun Hong Yoon, Peter Berntsen, Johan Bielecki, Benedikt J. Daurer, Hasan Demirci, Petra Fromme, Max Felix Hantke, Filipe R.N.C. Maia, Anna Munke, Carl Nettelblad, Kanupriya Pande, Hemanth K.N. Reddy, Jonas A. Sellberg, Raymond G. Sierra, Martin Svenda, Gijs Van Der Schot, Ivan A. Vartanyants, Garth J. Williams, P. Lourdu Xavier, Andrew Aquila, Peter H. Zwart, and Adrian P. Mancuso. Correlations in Scattered X-Ray Laser Pulses Reveal Nanoscale Structural Features of Viruses. *Physical Review Letters*, 119(15), 2017. doi:10.1103/PhysRevLett.119.158102.

[KKB81] Z Kam, M. H.J. Koch, and J. Bordas. Fluctuation x-ray scattering from biological particles in frozen solution by using synchrotron radiation. *Proceedings of the National Academy of Sciences of the United States of America*, 78(6 I):3559–3562, 1981. doi:10.1073/pnas.78.6.3559.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2833157.2833162.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stefan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi:10.25080/Majora-92bf1922-00a.

[MLS+14] Derek Mendez, Thomas J. Lane, Jongmin Sung, Jonas Sellberg, Clément Levard, Herschel Watkins, Aina E. Cohen, Michael Soltis, Shirley Sutton, James Spudich, Vijay Pande, Daniel Ratner, and Sebastian Doniach. Observation of correlated X-ray scattering at atomic resolution. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 369(1647):20130315, 2014. doi:10.1098/rstb.2013.0315.

[MWQ+16] Derek Mendez, Herschel Watkins, Shenglan Qiao, Kevin S. Raines, Thomas J. Lane, Gundolf Schenk, Garrett Nelson, Ganesh Subramanian, Kensuke Tono, Yasumasa Joti, Makina Yabashi, Daniel Ratner, and Sebastian Doniach. Angular correlations of photons from solution diffraction at a free-electron laser encode molecular structure. *IUCrJ*, 3(6):420–429, 2016. doi:10.1107/S2052252516013956.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PDM+18] Kanupriya Pande, Jeffrey J Donatelli, Erik Malmerberg, Lutz Foucar, Christoph Bostedt, Ilme Schlichting, and Petrus H Zwart. Ab initio structure determination from experimental fluctuation X-ray scattering data. *Proceedings of the National Academy of Sciences of the United States of America*, 115(46):11772–11777, 2018. doi:10.1073/pnas.1812064115.

[SA19] Elliott Slaughter and Alex Aiken. Pygion: Flexible, Scalable Task-Based Parallelism with Python. In *Proceedings of PAW-ATM 2019: Parallel Applications Workshop, Alternatives to MPI+X, Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58–72. Institute of Electrical and Electronics Engineers (IEEE), 2019. doi:10.1109/PAW-ATM49560.2019.00011.

[vdWCV11] Stéfan van der Walt, Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22–30, 2011. doi:10.1109/MCSE.2011.37.

[WRD15] William E. White, Aymeric Robert, and Mike Dunne. The linac coherent light source. *Journal of Synchrotron Radiation*, 22:472–476, 2015. doi:10.1107/S1600577515005196.

[ZBS+10] Junjie Zhang, Matthew L. Baker, Gunnar F. Schröder, Nicholai R. Douglas, Stefanie Reissmann, Joanita Jakana, Matthew Dougherty, Caroline J. Fu, Michael Levitt, Steven J. Ludtke, Judith Frydman, and Wah Chiu. Mechanism of folding chamber closure in a group II chaperonin. *Nature*, 463(7279):379–383, 2010. doi:10.1038/nature08701.