Python in the NERSC Exascale Science Applications Program for Data

Zahra Ronaghi*, Rollin Thomas*, Jack Deslippe*,
Stephen Bailey†, Doga Gursoy‡§, Theodore Kisner¶‖, Reijo Keskitalo¶‖, and Julian Borrill¶‖
*National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, California, 94720
†Physics Division, Lawrence Berkeley National Laboratory, Berkeley, California, 94720
‡Advanced Photon Source, Argonne National Laboratory, Lemont, Illinois, 60439
§Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208
¶Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California, 94720
‖Space Sciences Laboratory, University of California at Berkeley, Berkeley, California, 94720

***Abstract*—We describe a new effort at the National Energy Research Scientific Computing Center (NERSC) in performance analysis and optimization of scientific Python applications targeting the Intel Xeon Phi (Knights Landing, KNL) manycore architecture. The Python-centered work outlined here is part of a larger effort called the NERSC Exascale Science Applications Program (NESAP) for Data. NESAP for Data focuses on applications that process and analyze high-volume, high-velocity data sets from experimental/observational science (EOS) facilities supported by the US Department of Energy Office of Science. We present three case study applications from NESAP for Data that use Python. These codes vary in terms of "Python purity" from applications developed in pure Python to ones that use Python mainly as a convenience layer for scientists without expertise in lower level programming languages like C, C++ or Fortran. The science case, requirements, constraints, algorithms, and initial performance optimizations for each code are discussed. Our goal with this paper is to contribute to the larger conversation around the role of Python in high-performance computing today and tomorrow, highlighting areas for future work and emerging best practices.**

1. Introduction

The National Energy Research Scientific Computing Center (NERSC)[1] is the US Department of Energy (DOE) Office of Science production facility for high-performance and data-intensive scientific computing. NERSC serves over 6000 users on over 700 projects aligned with the Office of Science's unclassified research mission.

NERSC's newest system for its users is "Cori," a 28 PF (theoretical peak) Cray XC40 system completed in 2016. Cori is designed to handle the full spectrum of NERSC users' scientific computing needs in a single system. On Cori, massively parallel *capability* simulations of complex physical processes run alongside throughput-oriented *capacity* analyses of data streams from experimental and observational science (EOS) facilities. Supported EOS facilities include spectrographs, telescopes, genome sequencers, synchrotron radiation sources, and high-energy particle physics detectors.

Cori's design and operations policies enable a number of features especially friendly to data-intensive EOS workloads. These include

- The Burst Buffer [1], a layer of non-volatile storage filling the performance gap between main memory and the parallel file system used to accelerate I/O;
- Real-time, interactive, serial, and shared-node queues for rapid turnaround, exploratory data analytics, and "embarrassingly parallel" data processing;
- Large memory nodes dedicated to workflow management, data analytics platforms like Jupyter,[2] and memory-intensive jobs;
- Software-defined networking [2] for flexible network configuration and large bandwidth to compute nodes to enable streaming data analysis; and
- Shifter,[3] a system that allows containerized environments (e.g. using Docker) to run on a supercomputer [3].

High-performance and data-intensive computing can even be combined in a single workflow on Cori if needed.

Cori consists of two architecture partitions: "Phase I" with 2,388 nodes based on Intel Xeon "Haswell" processors, and the larger "Phase II" with 9,688 nodes based on Intel Xeon Phi "Knights Landing" (KNL) manycore processors. While KNL can run many applications without modification, most applications require code changes to achieve good performance. To prepare codes, developers, and staff for Cori and beyond, the NERSC Exascale Science Applications Program (NESAP) was launched in the Fall of 2014. NESAP fosters collaborative partnerships between application developers, NERSC staff, and vendor tools/library teams to adapt codes for manycore and develop associated best practices.

1. http://www.nersc.gov/

2. http://jupyter.org/

3. http://www.nersc.gov/research-and-development/user-defined-images/

The baseline goal for each code is that on a node-for-node basis, running on Cori Phase II should meet or exceed performance observed on Cori Phase I. In general, NESAP efforts result in improved performance for both Haswell and KNL [4]. Lessons learned through NESAP are disseminated to the broader NERSC community at user meetings, training events, and through documentation. NESAP also provides valuable feedback and strategic guidance on optimization tools and libraries directly to vendors. Focused, intensive, multi-day work sessions involving application developers, NERSC staff, and experts from Intel and Cray called "Dungeon sessions" scheduled several times a year have been critical to the success of NESAP.

NESAP for Data, an extension to NESAP, explicitly addresses data-intensive science applications that rely on processing, simulation, and analysis of massive datasets acquired from EOS sources. Rapid advances in detector technologies, instrumentation, embedded computing systems, and networking are all driving the rate of data production from EOS sources to the point where exascale computing will be necessary to extract insight [5]. The objectives of this program are to enable such applications to begin preparing for exascale, to take full advantage of the KNL chipset on Cori, and to find ways to exploit Cori's data-friendly features. As a secondary objective, NERSC also gains insight into the needs and requirements of data-intensive EOS applications through direct engagement.

It is through direct engagement that we have come to appreciate fully the rapid pace at which Python has gained ground among EOS software developers. These developers especially value Python's expressive syntax, shallow learning curve, extensive standard library, and burgeoning ecosystem of open-source extension packages that are increasingly easy to install and manage.

Python is, first and foremost, a *high productivity* language largely designed and implemented to deliver good enough but not necessarily *high performance*. It is interpreted, automatically memory managed, and dynamically typed. CPython, the widely used reference implementation of Python, uses a global interpreter lock (GIL) to manage memory consistently. Such characteristics help make Python easy to use but also present obstacles to performance.

The Python community has developed a variety of strategies to work around Python's performance blocks when performance matters. Most common is the use of high-performance libraries implemented in compiled languages (like C, C++, and Fortran) that are exposed to Python through any number of convenient wrapper interfaces. Cython[4] can be used to compile a superset of the Python language to optimized static C extensions. The widespread use of these tools makes scientific Python a textbook example of Ousterhout's two-language paradigm, with interpreted Python "gluing" together compiled extensions that do heavy numerical lifting [6]. Stretching that paradigm, tools like Numba[5] allow targeted just-in-time (JIT) or ahead-of-time (AOT) generation of optimized machine code from decorated Python source, even supporting GPUs. Alternatives to the CPython implementation like PyPy[6] address some of its shortcomings including memory usage and speed. In various cases, vendors like Continuum Analytics and Intel have contributed performance optimizations to the scientific Python ecosystem, given their interest in their products performing well for users. In all, as the scientific Python ecosystem has matured, users have found they can achieve acceptable levels of performance while retaining Python's ease of use.

NERSC and similar facilities have a vested interest in learning whether these strategies will continue work as we move into the era of manycore architectures like KNL and beyond. Power constraints for future exascale systems are driving a transition in supercomputing to these more energy-efficient architectures. Cori Phase II represents a step in this direction for NERSC; exascale systems at facilities like NERSC are now less than a decade away. Will it be enough to identify and exploit parallelism in underlying math libraries, or does the Python (in particular CPython) interpreter's single-thread execution become too dominant a bottleneck, limited by lower clock frequencies and reduced instructions retired per cycle? Can tools like Numba or alternative interpreter implementations focused on parallelism and performance seamlessly take over from CPython? Can Python developers accept loss of abstraction in pursuit of performance, or can they continue to develop schemes that deliver performance without loss of abstraction? By studying how real scientific Python developers confront manycore architectures, we hope to find the answers.

Three Python-based applications were selected as part of NESAP for Data in late 2016. These include TOAST[7] (Time Ordered Astrophysics Scalable Tools), a framework for simulating and analyzing Cosmic Microwave Background (CMB) radiation surveys; DESI (Dark Energy Spectroscopic Instrument) spectroscopic extraction[8] and redshift estimation[9] codes; and TomoPy,[10] an open-source Python package for tomographic data processing and image reconstruction. The developers of these codes use Python to balance productivity and performance in the face of higher volume and higher velocity data sets from real scientific instruments. These codes vary in terms of "Python purity" from applications developed in pure Python to ones that use Python mainly as a convenience layer for scientists without expertise in lower level programming languages like C, C++ or Fortran. All of these applications expect to use Cori's successor system, and at least one of them will probably be running on the system in place after that.

This paper introduces our three NESAP for Data Python application case studies in progress, and represents our first documentation of the effort. Future updates are planned as the program progresses. This paper is laid out as follows.

4. http://cython.org/
5. https://numba.pydata.org/
6. https://pypy.org
7. https://github.com/hpc4cmb/toast
8. https://github.com/desihub/specter
9. https://github.com/desihub/redrock
10. https://github.com/tomopy/tomopy

Section 2 provides further technical background on Cori and the KNL architecture to help readers understand our Python application performance challenges. Sections 3 through 5 present the case study applications and work done so far to improve performance on Cori. In Section 6 we reflect on cross-cutting lessons learned and suggest best practices for developers faced with porting Python applications to systems like Cori that use KNL. We state our conclusions in Section 7.

2. System Description and Issues for Python

Cori Phase I nodes have two Intel Xeon E5-2698v3 Haswell processors and 128 GB of DDR4 memory. Each processor has 16 cores running at 2.3 GHz, each with two hyper-threads, and two 256-bit wide vector units. A total of 64 threads are available per node. Each core has its own L1 and L2 caches, with capacities of 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket.

In contrast, each Phase II node has a single 1.4 GHz 68-core Intel Xeon Phi 7250 KNL processor; slower clock cycle per core but more cores per node than Phase I. Each core supports four hardware threads (272 threads available per node) and includes two 512-bit wide vector processing units and a 64 KB L1 cache. Cores on a node connect through a two-dimensional mesh network with two cores per "tile." Within a tile, cores share a 1 MB cache-coherent L2 cache. Notably, there is no L3 cache. Each node has 96 GB of DDR4 2400 MHz memory provided by six 16 GB DIMMs (115.2 GB/s peak bandwidth). Each node also includes 16 GB of on-package high-bandwidth memory with bandwidth 5 times that of DDR4 DRAM ($>$ 460 GB/sec). This multi-channel DRAM (MCDRAM) memory has flexible memory modes, including "cache mode" (making it effectively like a very large L3 cache), "flat mode" (a unique NUMA domain, separate from DDR4), and a hybrid of the two.

Cori Phase II represents a shift in terms of architecture but it is not as dramatic a change from previous Xeon-based systems as a GPU-based system would be. At a high level there are three key aspects to achieving code performance on KNL: (1) Using fine-grained parallelism to exploit the 68 cores per node, (2) taking advantage of the 512-bit vector units on KNL, and (3) structuring code to maximize memory access from KNL's 16 GB of onboard MCDRAM memory.

For Python applications this roughly translates to: (1) Making use of optimized, threaded, and hopefully vectorized math libraries like the Intel Math Kernel Library (MKL) through Python extensions like NumPy, SciPy, Numexpr, Scikit-Learn, etc.; (2) maximizing the amount of time spent by an application exercising those libraries or otherwise minimizing time in single-threaded interpreter execution; (3) ensure that NumPy array syntax, slicing, and broadcasting rules are used efficiently, and avoid allocation of large temporary objects; and (4) use performance analysis tools to identify, understand, and restructure hotspot kernels possibly replacing them with hand-optimized threaded C, C++,

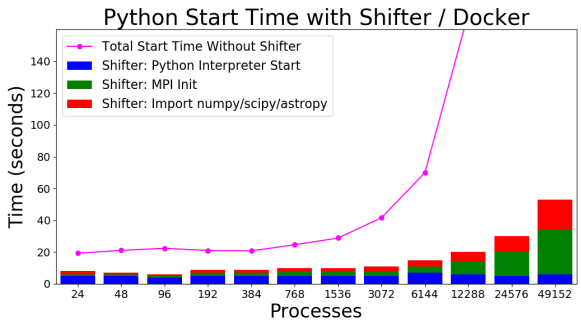

Figure 1. Application start-up time when using Shifter to run a Docker container using Python, compared with launching from a globally mounted file system.

or Fortran code, or using JIT/AOT compilation tools like Numba. For these strategies to work the Python community (including vendors) needs to find ways to deliver high-performance through lower-level extensions. Tools need to be made available that provide deeper insight into Python code performance that can cross the language barrier to allow analysis of compiled extensions. Many of these issues are being addressed by the Intel Distribution for Python and the Anaconda Python distribution, and Intel's performance analysis tools have begun to support Python. We expect other vendors to follow suit. Finally application developers need to balance their need for productivity and even code readability (a major draw for Python developers) against performance. At some point they may be faced with code they cannot optimize in Python that may only be accelerated through hand-optimized, compiled C code. Addressing these issues was the focus of a Python-centric Dungeon session held earlier this year with the Intel Python team.

Thus far the discussion has mostly been about single-node performance of Python, but all three selected case study codes need to scale beyond a single node — indeed the TOAST application needs to scale to use all Phase II nodes. A well known problem with Python in large cluster or supercomputing contexts is its start-up overhead, especially when Python application stacks are hosted on shared file systems. In fact this is a problem that afflicts applications that dynamically load libraries at runtime, not just Python applications. Python's dynamic import mechanism is very metadata intensive, and when a large number of Python processes start importing libraries they bottleneck the file system metadata server catastrophically; a large Python job may never finish importing its libraries before exhausting allocated wallclock time. Several solutions have been proposed that cache or localize metadata to compute nodes like DLFM [7], Spindle [8], and python-mpi-bcast [9]. Each project at NERSC gets space on a GPFS file system that is mounted read-only from compute nodes with client-side DVS (Cray Data Virtualization Service) caching, which can also help to this problem.

Shifter [3] was developed at NERSC in part to address performance issues with applications that load libraries

dynamically, including ones written in Python. Users can deploy Docker containers containing their entire application stack and preferred Linux distribution to Cori using Shifter. When Python is installed inside the Docker image, all shared libraries and modules are effectively in each compute node's local RAM disk. This isolates metadata queries to the compute node, and the application gets a performance boost from using the RAM disk. Further, since at build time the user has root privileges in the Docker container, they can cache library symbol lookups using the "ldconfig" tool. Figure 1 shows the improved startup time when using this type of container based solution. For now the default solution offered to NERSC Python users on any platform including Cori Phase II is to use Shifter for maximum cluster-level scalability.

3. Case Study I. TOAST

The TOAST package supports the simulation and reduction of the time-ordered data streams gathered by CMB experiments. The CMB consists of the primordial photons created in the Big Bang, propagated through (and impacted by) the entire history of the Universe. Tiny fluctuations in temperature and polarization of the CMB then encode the fundamental parameters of cosmology and fundamental physics, including the energy scale of early cosmic inflation and the number of neutrino species and their mass. However, CMB fluctuations are so faint that experiments have to gather enormous datasets to separate the signal from the noise and systematics (see Figure 2). Consequently CMB datasets have been growing exponentially for the last 20 years and are projected to do so for the next 20 too. High-performance computing (HPC) has therefore become an integral part of CMB data analysis, and the DOE Particle Physics Project Prioritization Panel (P5) has recommended that DOE take the lead in HPC for future CMB experiments, and in particular the proposed "Stage 4" experiment CMB-S4 [10].

Although much of the development of TOAST was for the Planck satellite mission, by design it supports the simulation and reduction of data from any CMB experiment. It is being used for analysis of the current Planck,[11] POLARBEAR/Simons Array,[12] EBEX,[13] and BLAST[14] experiments, and for the design and development of many next generation of experiments, including the Simons Observatory[15] and CMB-S4[16] ground-based telescopes, and the LiteBIRD,[17] CORE,[18] and CMBprobe[19] satellite missions.

The long-term exponential growth of CMB datasets has required CMB developers to track the leading edge of HPC,

11. http://www.rssd.esa.int/index.php?project=PLANCK
12. http://bolo.berkeley.edu/polarbear/
13. http://groups.physics.umn.edu/cosmology/ebex/
14. http://blastexperiment.info/projects/blast/index.php
15. http://simonsobservatory.org/
16. http://cmb-s4.org/
17. http://litebird.jp/eng/
18. http://www.core-mission.org/
19. https://zzz.physics.umn.edu/ipsig/start

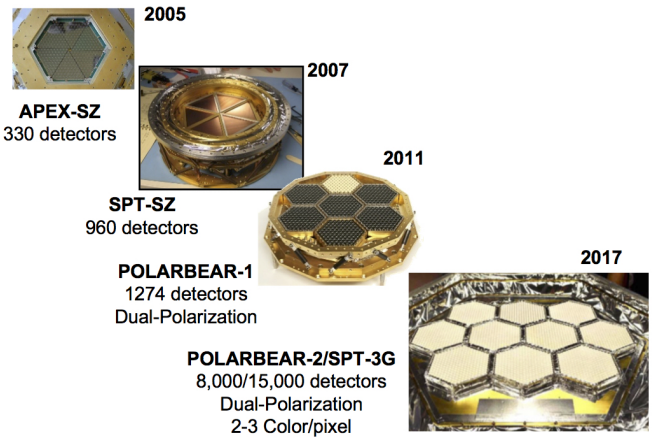

Figure 2. The focal planes of 4 generations of CMB experiment, showing the growth in the number of detectors required to meet the signal-to-noise requirements of each generation's science goals.

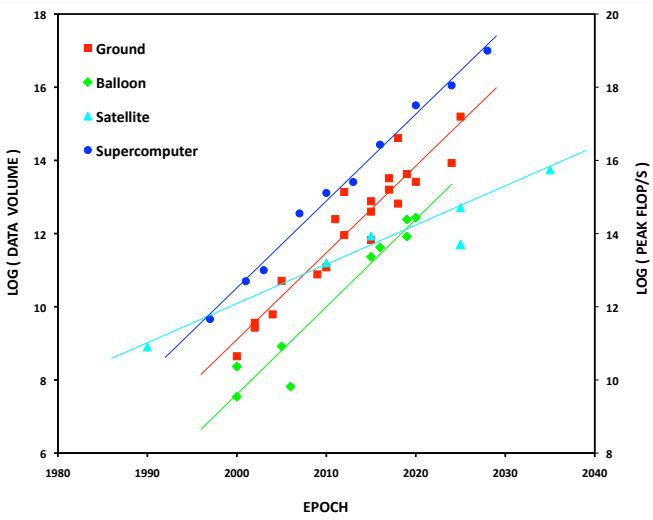

Figure 3. Exponential growth of the volume of CMB data for ground-based, balloon-borne, and satellite missions, and of the theoretical peak performance of the flagship NERSC system, over a 40 year period. Note that suborbital experiments perfectly track Moore's Law.

while simultaneously maintaining computational efficiency, simply to keep up with the data volume (Figure 3). Over the last 20 years this has required porting and optimizing our code base to 6 generations of NERSC hardware, from MCurie to Edison. The same must now be done for Cori. However, the KNL architecture is qualitatively different from anything the developers have seen before, and they expect that it will require a much more aggressive optimization strategy to achieve the efficiency required to exploit its capabilities. They hope to achieve this goal and be able to generate and reduce the simulations that will be critical to the validation and optimization of the design of the CMB-S4 experiment.

The TOAST software stack consists of parts written in several different languages. It has compiled libraries

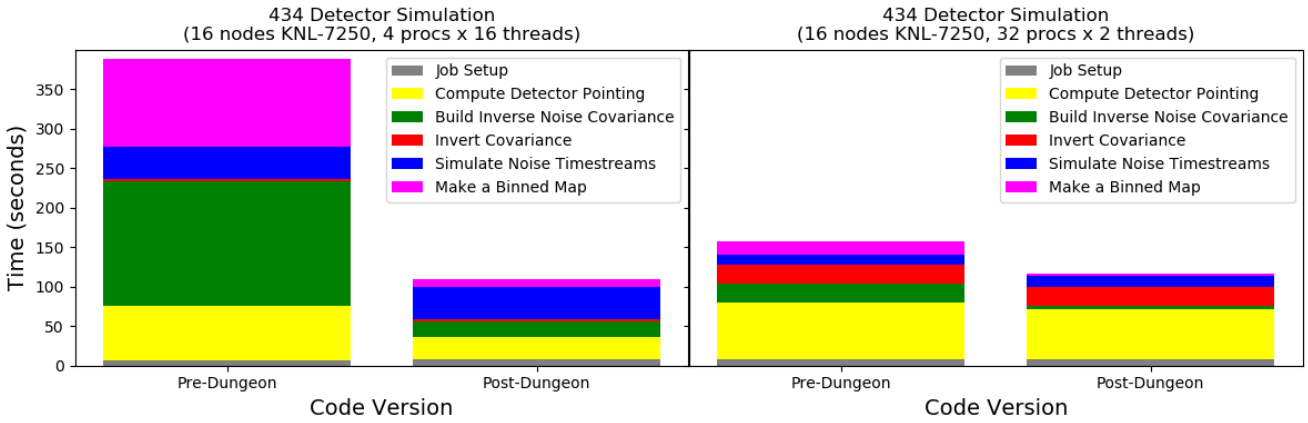

Figure 4. Performance improvements of a TOAST workflow on KNL after making changes identified at an Intel "Dungeon Session".

written in C++ and Fortran that use both MPI and OpenMP parallelism. TOAST also includes a workflow management layer written in Python using mpi4py[20] [11][12][13]. Data formats are usually HDF5 or FITS. Internal math operations include quaternion manipulation, streamed random number generation, 1D FFTs, and MPI-parallel iterative (conjugate gradient) solvers. The dominant cost of the current code for typical runs is the calculation and MPI communication done at each step of an iterative solver. The biggest challenge is better vectorization of that calculation and determining trade-offs between the number of MPI processes and threads on each node.

As part of the NESAP for Data program, TOAST developers have worked to identify possible areas of improvement for the code base. Some of these optimizations were realized at the Python Dungeon session earlier this year, and others implemented afterward based on lessons learned at the Dungeon. One optimization was changing a sparse vector indexing strategy to use more efficient NumPy unique() calls instead of a Python set; this simple change resulted in a 20% speed-up alone. A slightly more involved code change was using an inverse error function call to Intel MKL instead of a less efficient Box-Muller implementation to generate Gaussian-distributed numbers; a 100% speed up. In quaternion operations, a single vector/quaternion was being copied in memory many times leading to excessive memory consumption. The TOAST team highlighted that the code needs to process a large number of independent small matrices and explored with engineers the possibility of vectorized batch processing of these matrices in a future MKL release.

The above changes were made and tested on a typical workflow at different configurations of MPI processes versus threads. Figure 3 shows the improvement of different sections of the code before and after optimization. Working with NERSC staff, the TOAST developers have been able to use the entire Cori Phase II system to simulate and reduce the data that would be gathered by a notional Simons Observatory configuration, with 50,000 detectors observing 20% of the sky for 1 year from the Atacama Desert in Chile.[21] The simulated data included not only the sky signal but also realistic instrumental noise and atmospheric fluctuations. To achieve this, the optimized TOAST code was deployed to Cori in a Docker container and the application run with Shifter (Section 2). Thanks to Shifter, the full-machine application start-up time was kept to less than 90 seconds.

4. Case Study II. TomoPy

Analysis of tomographic datasets at synchrotron light sources has becoming challenging due to increasing acquisition rates and resolution of the detectors. TomoPy[22] [14] is a framework for processing of synchrotron tomographic data and written primarily in Python with some code (25% by lines of code) in C/C++. The data format is usually HDF5 and the Scientific Data Exchange package[23] [15] can be used to import tomographic data from different synchrotron facilities.

At the Advanced Photon Source (APS) Imaging Group, dynamic tomography is routinely performed to provide 3D imaging of evolving systems. Various high speed cameras are used to optimize speed, dynamic range and sustained data collection time. The current combined raw data rate, at one beamline alone, 2-BM, is currently at 300 TB/year. This figure will increase dramatically, potentially in excess of 1 PB/year, once a new faster detector like the CoaXPress CXP-6 Quad will be installed in production mode. In 2015, Advanced Light Source (ALS) produced 14,399 data sets, and in 2016 so far 8900. The size of tomography data sets generally varies between 4 and 16 GB, on average 12 GB, and as a result about 80 TB of raw data was produced in 2015.

20. https://bitbucket.org/mpi4py/mpi4py

21. https://crd.lbl.gov/departments/computational-science/c3/c3-research/cosmic-microwave-background/cmb-simulations-at-scale/

22. https://tomopy.readthedocs.io/en/latest/

23. http://dxchange.readthedocs.io/en/latest/

Some of the application areas across the DOE complex include: in-situ study of corrosion behavior of materials, dynamic geological systems, plant biology, ceramic matrix composites, batteries, brain imaging. For example: Micro-CT at a synchrotron facility is able to characterize structural metallic materials under fatigue-stress and corrosive environment, it also allows monitoring complex physiochemical modifications in rocks and their constituent minerals. These measurements often involve fluid phases, under high pressure and high temperature and are critical to understand geological instabilities (mudslides, earthquakes), energy extraction (hydrocarbon and geothermal heat), ore formation and environmental concerns (CO2 sequestration and contaminant pathways). Figure 5 includes examples of three-dimensional tomographic reconstructions.

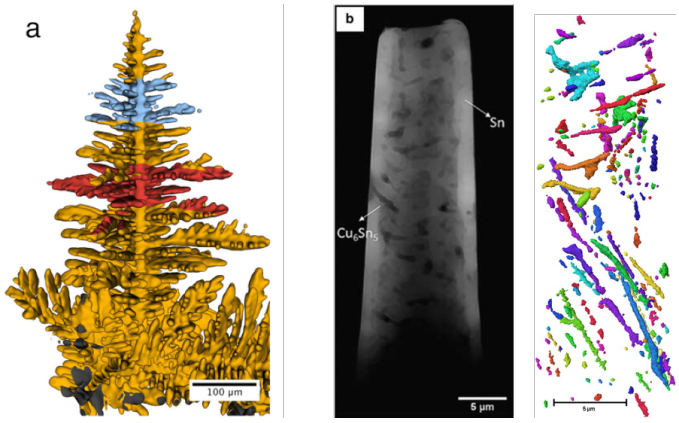

Figure 5. (a) Three-dimensional tomographic reconstruction showing morphology of growing dendrites using X-ray synchrotron radiation [16]. (b) Detailed three-dimensional reconstruction showing the distribution of Cu6Sn5 intermetallic compounds in the micropillar [17].

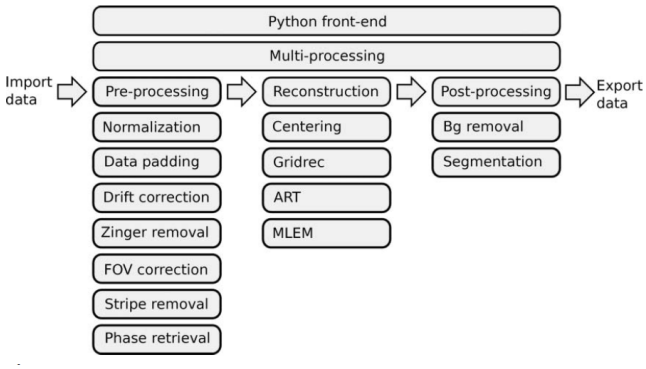

Figure 6. Illustration of the TomoPy Framework [14]. Tomographic analysis pipeline consists of a series of tasks (such as for pre-processing, reconstruction and post-processing tasks) which are implemented in a number of modules in TomoPy.

The tomographic analysis pipeline consists of a series of tasks (such as for pre-processing, reconstruction and post-processing tasks) which are implemented in a number of modules in TomoPy (Figure 6. Typically, the majority of the time is spent on the reconstruction module (computation), which contains functions that map data from data space into image (or object) space. If the number of allocated nodes for reconstruction is very large (i.e. the number of nodes is larger than the number of sinograms), then communication and I/O start to become the bottlenecks.

Inter-node parallelization is expressed using MPI by distributing the sinograms among MPI processes and intra-node parallelism currently uses Python's multiprocessing and concurrent futures modules. TomoPy can scale to the number of projections on a single node, which is typically more than 360; and can scale to the number of sinograms on the cluster nodes. This has been implemented on 2K nodes (32K cores) on Mira at Argonne National Laboratory [18]. However, if the number of nodes exceeds the number of sinograms, inter-node communication will significantly introduce overhead and delay.

As part of the NESAP for Data project we ran TomoPy on Cori Phase I and II. We have mainly focused on optimizing the Gridrec reconstruction algorithm, which is a direct Fourier-based method similar to the filtered-backprojection method. By design, Gridrec reconstructs two slices at the same time, in the real and imaginary part of the FFT, to speed-up the process.

Below are the major optimization steps applied to the reconstruction module for improving single-node performance on Haswell and KNL:

1) Built Tomopy with icc targeting both Haswell and KNL (common-avx512) architecture
2) Modules for fftw, pyfftw, dxchange, dxfile and olefile are built locally
3) For process-level threading (multiprocessing) in Python on KNL set environment variable: KMP_AFFINITY = disabled
4) Replace lroundf(x) with (int)roundf(x), fabs(x) with fabsf(x), ceil(x) with ceilf(x)
5) Apply vectorization pragmas, split a double loop to enable vectorization and add data alignment to assist vectorization, for example: H = malloc_matrix_c(pdim, pdim); __assume_aligned(H, 64)
6) Apply omp simd collapse directive to enable the compiler to collapse and execute concurrently using SIMD instructions

We generated an object (dataset) with size 1024×1024 pixels in the x and y directions, with 720 uniformly spaced tilt angles, and a reconstructed output size of 1024^3. Figures 7 and 8 show total reconstruction time for TomoPy using the conda-based installation (Conda), installing from source (Source) and the optimized version (Optimized).

The next steps include investigating threaded matrix multiplication and convolution as well as optimizing memory management and data layout for Gridrec reconstruction. This will be challenging since there is random access patterns in updating memory components when calculating the 1D convolvents in the X and Y directions to update the 2D complex array. Restructuring the code will also enable vectorization.

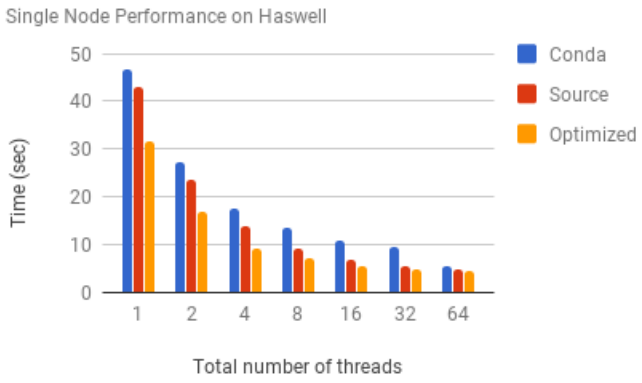

Figure 7. Single Node Performance on Haswell.

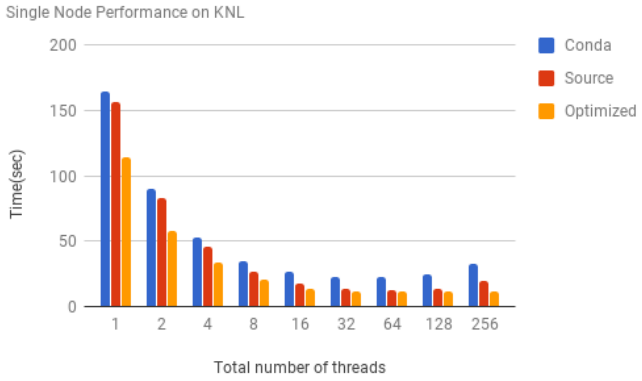

Figure 8. Single Node Performance on KNL.

We are also modifying the data-exchange package to enable parallel read and write for HDF5 file formats for scaling and cluster level parallelism. More than 30% of application runtime is spent in I/O and interactive reconstruction observation is essential for researchers to monitor data quality as well as planning the next experimental steps. The NERSC Burst Buffer has also been used with the SPOT suite (workflow coupled to experimental beamline at ALS or APS) instead of the Lustre filesystem to improve performance for reading input and writing output files [1]. We will continue investigating parallel I/O with the Burst Buffer with different datasets and analyses.

5. Case Study III. DESI Spectroscopic Pipeline Codes

The third NESAP for Data Python project is the DESI spectroscopic pipeline. This pipeline will process the 50 million spectra of galaxies, quasars, and stars observed by DESI, converting raw telescope data into calibrated spectra and redshift measurements. This constitutes the core dataset to be used by the DESI science collaboration to make Dark Energy cosmology measurements from 2019–

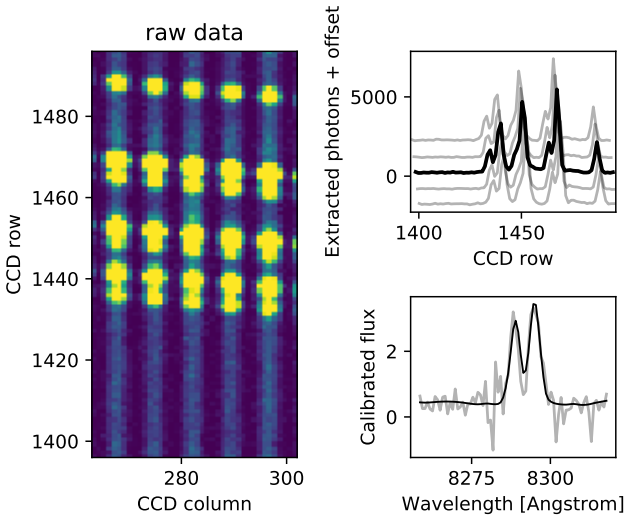

Figure 9. DESI simulated raw data (left); uncalibrated extracted spectra (upper right); and wavelength-calibrated sky-subtracted spectra with best fit [OII] doublet template for a redshift = 1.22 emission line galaxy.

2024. The codes are open source with the intent of sharing the core algorithms with other multi-object spectrograph experiments.

Compared to current generation experiments, DESI's new algorithms are more mathematically rigorous to maximize information extraction from low signal-to-noise data. However they are also $\mathcal{O}(100)$ times more computationally demanding than current generation algorithms and must process $\mathcal{O}(10)$ times more data. A naive implementation would involve eigen-decomposition of a sparse 16M $\times$ 16M matrix, 500–1000 times per night just to keep up with the data flow. Even so, the pipeline is highly data-parallel.

The existing DESI code has been optimized for multi-core architectures like Haswell (Cori Phase I), but not for manycore architectures like KNL (Phase II). Although this meets DESI's current computational needs, such machines may simply not be available to DESI for the final data processing in 2025. Shifting DESI compute to another site entails costs that the project's main funding agency (the DOE Office of High Energy Physics) will not support. Thus DESI must port its code to next generation architectures such as KNL sooner rather than later. Doing so will allow DESI to benefit from the power of Cori Phase II and subsequent NERSC machines. It will also retire a technical risk for the DESI project.

The pipeline itself includes a number of different codes. Although the DESI team has written some pipeline steps in pure C++, the majority of the code is Python thus DESI has prioritized optimizing its Python usage and ensuring effective interaction between Python and underlying compiled libraries. The choice of Python emphasizes developer productivity over raw code performance; this is especially important for the scientist-developers who are not HPC experts. DESI developers have from the beginning adhered to Python best practices of using NumPy vector operations

whenever possible instead of explicit Python for loops.

The two most computationally expensive algorithms implemented in Python are spectral extraction and redshift fitting. Spectral extraction forward models the DESI spectrographs to determine what set of 1D spectra (Figure 9 upper right) would generate the observed raw data (Figure 9 left). In this simulated example, the most visible features are OH emission lines from the sky, which dominate the signal of a slight excess of photons around rows 1445–1455 of the middle spectrum. Once the spectra are wavelength calibrated, sky subtracted, and flux calibrated, redshift fitting compares galaxy models vs. redshift to determine the galaxy type and best fit redshift. Figure 9 lower right shows the best fit [OII] doublet of a redshift 1.22 emission line galaxy. The tiny signal compared to the large sky backgrounds drives the need for algorithms that optimally model the data with fully correct error propagation. Figure 9 represents 1/120000 of the data obtained by a single 15-minute DESI exposure.

These algorithms are mostly implemented in Python, using NumPy/SciPy to leverage Intel MKL and other compiled code for computationally intensive steps. The redshift fitting step additionally uses Numba[24] for JIT compilation of a single rate-limiting function. These codes typically involve matrix algebra (in particular, Hermitian matrix eigendecomposition), fast Fourier transforms, and fitting functions to data. These and other data processing methods are obviously not unique to DESI.

The pipeline orchestration is custom written due to a complex dependency tree that requires very different levels of concurrency for different steps of the processing (single core through thousands of parallel processes depending upon the step). I/O is primarily file-based, though the final catalogs are also loaded into a database for end-user queries. C++ steps use a Python pipeline mpi4py wrapper to run OpenMP parallelized C++ code. These aspects of the pipeline are not explored in the present study, but we observe they are quite common in existing EOS pipelines.

Prior to NESAP for Data, the DESI team profiled its Python codes on Edison (NERSC's Ivy Bridge-based Cray XC30) so they could identify which algorithm steps take the majority of the time. The most computationally expensive step (spectral extraction) consisted of approximately 1/3 MKL matrix algebra, 1/3 non-MKL NumPy vector operations, and 1/3 miscellaneous Python — i.e., with no single dominant blocking step.

The redshift fitting Numba acceleration gained a factor of 600× speedup on an Intel Core i7, though we have not yet profiled the resulting Numba code to inspect how efficient it is, in particular for vectorization on KNL or what the best-practices are for writing Python code that allows the Numba compiled code to benefit from vectorization. Understanding this will be a key next step for this NESAP for Data case study.

In preparing the DESI Python code base for Cori Phase II, there are 3 key challenges. (1) Cluster-level scaling of the DESI pipeline, like TOAST, is affected by Python's poor package import performance on distributed file systems but not at the same scale. (2) To date on-node parallelism has been achieved through Python's multiprocessing library, and originally the developers had planned to continue using process-based parallelism for on-node scaling coupled with mpi4py for cluster-level scaling but this has proved unreliable. (3) Effective use of KNL threading and vectorization where appropriate.

Running Python at high concurrency can incur a large start-up cost if the software stack is installed on a distributed filesystem. When used in production, the DESI pipeline is installed inside a Docker image and that image is run on the Cray systems using Shifter (see Section 2).

The original DESI pipeline design used mpi4py to distribute independent pieces of data across compute nodes and Python multiprocessing for on-node parallelism of the algorithms, loosely anaolgous to MPI+OpenMP for C++. This decouples the data-parallel scaling from individual algorithms, allowing them to be developed and used on developer machines such as laptops without requiring MPI. The multiprocessing overhead is small compared to the runtime of the parallel processes and individually the multiprocessing-based algorithms worked well. When wrapped with MPI, however, we experienced multiple problems including outright crashes (fixed with `MPICH_GNI_FORK_MODE=FULLCOPY`), complete lack of KNL on-node scaling (fixed with `KMP_AFFINITY=disabled`), and unpredictable memory usage (fixed by using explicit shared memory instead of relying upon Linux fork being copy-on-write, which worked for multiprocessing-only but not MPI+multiprocessing). The crashes were specific to Cray MPICH and did not occur on a non-Cray system; the other problems were only tested on Cray machines. For now we have a working MPI+multiprocessing pipeline, but the number and severity of the problems indicates that this is a fragile combination which could break again in the future, so we are now pursuing an MPI-only refactor.

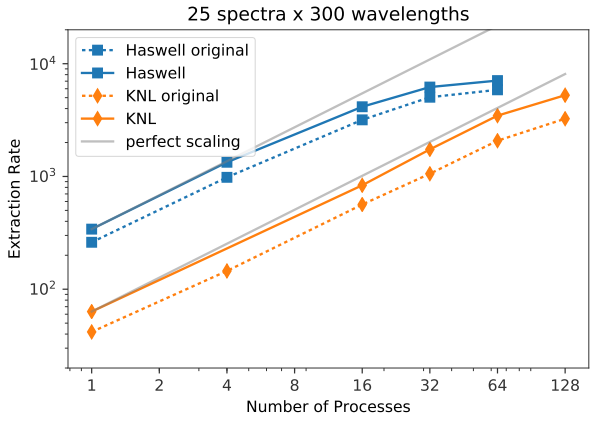

Figure 10. Improvements to the DESI spectral extraction rate for Haswell (blue squares) and KNL (orange diamonds).

The most computationally intensive steps are in MKL or

24. http://numba.pydata.org

other compiled code as wrapped by NumPy and SciPy, thus the codes already leverage vectorization that occurs in those libraries. Our initial benchmarks showed that the spectral extraction code was per-core 6.2× faster on Haswell than KNL and per-node 1.8× faster, i.e., the increased number of KNL cores did not outweigh their lower per-core performance. Working with Intel Python engineers, we profiled the spectral extraction code on both Haswell and KNL and did not find any KNL-specific hotspots. Multiple improvements were made, most significantly for better use of Numpy vectorization and hoisting calculations out of for loops, resulting in 1.2× speedup on Haswell and 1.6× speedup on KNL. However, the per-node performance is still 1.35× faster on Haswell than KNL. The before/after performance vs. number of processes used are shown in Figure 10. The redshift fitting code has not yet been optimized for KNL; this is the topic of our next Dungeon session.

6. Discussion

We now highlight a few of our most important observations and issues common across all three Python case study applications in NESAP for Data. At the very highest level we suggest that there is much to be gained by building community around the use of Python in HPC contexts beyond NESAP for Data. For Python to be a viable technology at exascale and in HPC in general, a community of users, developers, and vendors working together is our best hope.

Our initial expectation was that these relatively mature codes, written by experienced scientific Python developers, would be at the limit of what was possible with Python and would only improve through the use of hand-written compiled extensions. This was simply wrong, as we were often able to identify code changes at the Python level that resulted in speed-ups. In the case of TOAST, improved performance came from changing Python data structures or switching to more optimized algorithms in NumPy already in existence. With DESI, calling a NumPy function with array arguments instead of once per loop resulted in a respectable speed-up as well, at the cost of readability and an increased memory footprint. The latter example is a simple violation of NumPy best practices but it should not be construed to be symptomatic of a wider problem with the code-base, elsewhere array syntax and broadcasting rules are observed rigorously. These examples suggest that even experienced Python developers may be able to identify "low-hanging Python fruit" in their applications even if they think there is none. Performance profiling tools like cProfile/SnakeViz, VTune, or Advisor are essential to identify these opportunities.

Partnership with our vendors was essential in accelerating the pace of progress. A major initial block was the lack of scaling on KNL for applications using multiprocessing, which was resolved by setting `KMP_AFFINITY=disabled`. This was resolved quickly through discussion with Intel engineers. Another example was insight into interaction between multiprocessing and Cray MPICH, which was clarified by interaction with engineers from Cray. Furthermore we have found it relatively easy to suggest changes or enhancements to libraries and tools that the vendors understand and then implement. Dungeon sessions and in-person events were also critical for training developers in the use of powerful tools for optimization and analysis of codes. This helps us achieve our goals of helping KNL Python developers become self-sufficient.

In approaching these Python applications we have largely applied the same time-tested tools and techniques of performance analysis and optimization from HPC. This seems natural, but traditional HPC experts who have spent most of their careers analyzing Fortran or C++ applications may miss subtle differences about Python that are ultimately consequential. This means that for instance, we found ourselves having to explain the GIL to astonished colleagues when they asked why threading Python applications was a complex problem. Others approaching the problem of trying to extract performance from Python applications in HPC in general will find themselves educating the HPC community about Python.

7. Conclusion

In this paper we have introduced NESAP for Data, a collaboration of NERSC staff, code developers, and experts from Intel and Cray focused on porting, evaluating, and optimizing selected applications for the KNL architecture on the Cori Cray XC40 at NERSC. We have outlined the three NESAP for Data Python application case studies in progress, and plan future updates on them as the program continues.

The three applications discussed were selected for NESAP for Data in part because they represent a spectrum of "Python purity." One uses Python mainly as an orchestration and "glue" layer to combine massively parallel components to process and analyze CMB data sets (TOAST). Another consists of a mixture of scientific Python and hand-written C code to achieve performance for tomographic reconstruction (TomoPy). Developers of the third application, DESI, try to extract as much performance as possible from Python without resorting to hand-written C or C++ code as a last resort. Selecting applications designed with differing philosophies about using Python in HPC gives us insight into how their developers view the challenges they face. The results so far suggest that simply "diving in" and replacing Python code with hand-optimized C code may be necessary, but it is a best practice to first establish that other opportunities for optimization in the Python layer are actually exhausted. Even in mature code bases, there may be relatively easy to identify low-hanging Python fruit.

On a practical level, NESAP for Data is about making sure that real workloads scale on current and near-term architectures. But the time is right to begin thinking about how Python needs to evolve in the coming age of exascale. Exascale computing represents a tremendous opportunity for complex simulation workloads and big data for EOS.

There is a clear need for powerful programming models and tools that are usable and elegant. New programming models, and high-level languages like Python, might address this exascale programmability gap. The challenge is in figuring out how to keep the language expressive and abstract but somehow achieve levels of performance usually only available to programmers by discarding abstraction. The NESAP for Data effort will lead to performance gains, a better understanding of how Python applications perform on manycore systems like Cori, and the development of an overall optimization strategy. However we believe it is essential that NESAP for Data move forward as part of a larger community effort involving users, scientific Python developers, and vendors.

In the near future we will be continuing with our performance analysis and optimization efforts, while looking forward and exploring new tools. As alternative interpreters like PyPy become more mature it makes sense to consider whether they can seamlessly take over for the CPython interpreter used exclusively by NESAP for Data Python applications. We would be remiss in failing to ask whether Python itself is the right tool for filling the exascale programmability gap, and future applications brought into NESAP for Data that use Julia[25] should help us address this question.

Acknowledgments

The authors thank their partners at Intel, the Intel Python Team, Intel tools developers, performance engineers, and their management. Mike Ringenberg and Krishna Kandalla at Cray are thanked for helping us understand some issues with Cray MPICH, mpi4py, and multiprocessing. This work used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia *et al.*, "Accelerating science with the nersc burst buffer early user program," *CUG2016 Proceedings*, 2016.

[2] R. S. Canon, B. R. Draney, D. L. P. Jason R. Lee, D. E. Skinner, and T. M. Declerck, "Enabling the super facility with software defined networking," in *Cray User Group Meeting 2017*, 2017.

[3] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," in *Cray User Group Meeting 2015*, 2015.

[4] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas *et al.*, "Evaluating and optimizing the nersc workload on knights landing," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*. IEEE, 2016, pp. 43–53.

[5] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams, *Synergistic Challenges in Data-Intensive Science and Exascale Computing: DOE ASCAC Data Subcommittee Report*. Department of Energy Office of Science, 3 2013, type: Report.

25. http://julialang.org

[6] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[7] Z. Zhao, M. Davis, K. Antypas, Y. Yao, R. Lee, and T. Butler, "Shared Library Performance on Hopper," in *Cray User Group Meeting 2012*, 2012.

[8] W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf, "Massively parallel loading," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 389–398. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465020

[9] Yu Feng and Nick Hand, "Launching Python Applications on Petascale Massively Parallel Systems," in *Proceedings of the 15th Python in Science Conference*, Sebastian Benthall and Scott Rostrup, Eds., 2016, pp. 137 – 143.

[10] Particle Physics Project Prioritization Panel (P5), "Building for Discovery: Strategic Plan for U.S. Particle Physics in the Global Context," 2014. [Online]. Available: https://science.energy.gov/~/media/hep/hepap/pdf/May-2014/FINAL_P5_Report_053014.pdf

[11] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108 – 1115, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731505000560

[12] L. Dalcín, R. Paz, M. Storti, and J. D'Elá, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655 – 662, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731507001712

[13] L. D. Dalcín, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124 – 1139, 2011, new Computational Methods and Software Tools. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0309170811000777

[14] D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen, "TomoPy: a framework for the analysis of synchrotron tomographic data," *Journal of Synchrotron Radiation*, vol. 21, no. 5, pp. 1188–1193, Sep 2014. [Online]. Available: https://doi.org/10.1107/S1600577514013939

[15] F. De Carlo, D. Gürsoy, F. Marone, M. Rivers, D. Y. Parkinson, F. Khan, N. Schwarz, D. J. Vine, S. Vogt, S.-C. Gleber *et al.*, "Scientific data exchange: a schema for hdf5-based storage of raw and analyzed data," *Journal of synchrotron radiation*, vol. 21, no. 6, pp. 1224–1230, 2014.

[16] J. Gibbs, K. A. Mohan, E. Gulsoy, A. Shahani, X. Xiao, C. Bouman, M. De Graef, and P. Voorhees, "The three-dimensional morphology of growing dendrites," *Scientific reports*, vol. 5, p. 11824, 2015.

[17] C. S. Kaira, C. R. Mayer, V. De Andrade, F. De Carlo, and N. Chawla, "Nanoscale three-dimensional microstructural characterization of an sn-rich solder alloy using high-resolution transmission x-ray microscopy (txm)," *Microscopy and Microanalysis*, vol. 22, no. 4, pp. 808–813, 2016.

[18] T. Bicer, D. Gürsoy, R. Kettimuthu, F. D. Carlo, G. Agrawal, and I. Foster, "Rapid tomographic image reconstruction via large-scale parallelization," in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 289–302.