Speaking Pygion: Experiences Writing an Exascale Single Particle Imaging Code

Seema Mirchandaney[1], Alex Aiken[1,2], and Elliott Slaughter[1]

[1] SLAC National Accelerator Laboratory
[2] Stanford University

Abstract. The goal of the SpiniFEL project was to write, from scratch, a single particle imaging code for exascale supercomputers. The original vision was to have two versions of the code, one in MPI and one in Pygion, a Python-based interface to the Legion task-based runtime. We describe the motivation for the project, some of the programming challenges we encountered along the way, what worked and what didn't, and why only the Pygion code eventually succeeded in running at scale.

Keywords: task-based programming · exascale computing · single particle imaging.

1 Introduction

The ExaFEL project had as its goal to develop scalable and rapid approaches to the analysis of images produced by LCLS-II, the second generation free electron laser at the SLAC National Accelerator Laboratory. One component of the project was to develop a single particle imaging (SPI) code that could make use of the DOE's exascale supercomputers to perform reconstruction of 3D conformations of molecules from the 2D diffraction patterns generated by SPI experiments. By exploiting large numbers of GPU's, reconstructions that would normally take hours or days could be computed in minutes, allowing scientists to adjust their experiments based on near real-time feedback from shots of the laser.

From the start of the development of SpiniFEL, the plan was to implement a version targeting the Legion task-based runtime [2], but to mitigate risk it was decided that an equivalent MPI version should also be written since most of the team had experience with MPI, no experience with Legion, and Legion had not been previously demonstrated on any similar application. The decision to develop two versions led, in retrospect, to some predictable difficulties for the project. The MPI code received more attention and development effort, and as a result the Legion version was perpetually behind: features would first appear in the MPI code and only later be ported to Legion. Second, because developing two completely independent implementations was impractical, the two versions shared as much code as possible. Because features were supported in MPI first, the realization of features tended to be idiomatic for MPI, making it more difficult than necessary to take advantage of Legion's task-based features.

Another issue, however, proved to be the most important: SpiniFEL would be developed in Python, and when the project started Legion support for Python was minimal. At the time (2017), Legion had two well-supported programming interfaces, the API for Legion's C++ runtime system [2], and Regent, a programming language for the Legion programming model [10]. Regent had a number of advantages over programming directly to the C++ API, including compiler support for statically checking that the programming model is used correctly and a number of important optimizations. However, like Legion's C++ API, Regent's support for inter-operation with Python was primitive. Motivated by SpiniFEL, we developed Pygion, a native Python interface to Legion based on Regent [9].

Initially development of SpiniFEL focused on a batch computation that took a set of 2D diffraction patterns and reconstructed a single conformation. Once that essential functionality was implemented, the focus shifted to adding two significant extensions. The first was real-time processing of diffraction images using Psana2 [6], an infrastructure for managing images arriving directly from the X-ray laser, and the second was reconstructing multiple conformations from a single experiment—a single LCLS-II experiment consists of images of many different molecules, and so it is natural to identify multiple conformations.

Implementing these extensions was a turning point in the project: It was more difficult and time-consuming to modify the MPI code to support these features than it was to modify the Pygion version, in part because the extensions were less friendly to an SPMD-style program, but mainly because Pygion's flexibility, automatic discovery of dependencies and support for data partitioning proved to be significantly more productive to use when restructuring existing code. Eventually the Pygion version scaled and performed well with these additional features, while neither feature was fully implemented in MPI at the end of the project.

In this paper we give additional details of the development of SpiniFEL, focusing on the Pygion/Legion features used to support computing multiple conformations. After discussing related work in Section 2, we briefly describe the initial SPI parallel algorithm for single conformations and its extension to multiple conformations in Section 3. We then elaborate on the features of Pygion that simplified the restructuring of the code to go from computing single to computing multiple conformations in Section 4. Results from and the scaling performance of the Pygion version of SpiniFEL are in Section 5.

2 Related Work

The productivity benefits described in this paper were sufficiently significant that, while the details might be quite different, we would not be surprised if other task-based systems could provide similar benefits when developing a code in both MPI and the task-based framework. We briefly discuss five systems, two where we expect that the experience could be similar and three where we speculate that the experience could be very different.

StarPU [1] provides asynchronous tasking, automatic discovery of dependencies, and data partitioning built into the task programming model, three features that we highlight as having been particularly important in this work. StarPU has also recently added a Python interface. We note that the Pygion interface is at a somewhat higher level than StarPU's, as the Pygion implementation (dynamically) performs many of the optimizations done by the Regent (statically compiled) language for Legion, but we expect that any reasonable Python interface would be sufficient to realize an application such as SpiniFEL in StarPU.

PaRSEC [4] is another tasking system with asynchronous tasks, automatic discovery of dependencies, and a data partitioning subsystem, and so is another system that we would expect to experience similar benefits in relationship to MPI for a project of the scale of SpiniFEL. To the best of our knowledge, PaRSEC does not currently have a Python interface.

DASK is a native Python-based tasking system, built from the start to seamlessly integrate distributed tasking into Python and its ecosystem. DASK's main drawback is performance; the runtime system is centralized and also implemented in Python. The resulting high overheads per task [11] and inefficiencies in distributing work on a very large cluster would likely make implementing SpiniFEL efficiently in DASK challenging.

Charm++ [7] is an actor-based programming model; instead of having stateless tasks Charm++ relies on stateful actors called *chares* as its core building block. Chares execute methods in response to messages sent from other chares. In general the order of execution of the methods of a chare is non-deterministic which, combined with chares' internal updateable state, means that Charm++ programs are potentially non-deterministic in their visible behavior. The task-based systems, on the other hand, provide parallel execution with deterministic sequential semantics. Thus, it is not clear what lessons from the experience described in this paper would apply to Charm++. At the least, in our view the difficulty of debugging significant changes to explicitly parallel MPI code compared to debugging the more straightforward deterministic semantics of Pygion programs contributed to Pygion's greater productivity in writing SpiniFEL. Charm++ has extensive Python support through its charm4py library.

Ray, like DASK, is a native Python tasking library. Ray provides both pure tasks and actors [8]. Ray's overheads are similar to DASK, and so we would expect similar issues in supporting fine-grain tasks, and exploiting Ray's actors would likely be comparable to the experience of using Charm++.

3 SpiniFEL

The parallel algorithm for computing a single conformation is described in [3, 5] and was implemented in both MPI and Pygion. A number of phases are performed iteratively until either convergence or the maximum number of iterations is reached. Throughout the computation a current estimate of the 3D electron density is maintained and improved by the algorithm. The phases are:

- *Slicing* computes 2D images (slices) through the current electron density estimate.
- *Orientation matching* compares the actual 2D diffraction patterns from an SPI experiment to the slices, which is used to compute the orientation of the images using a closest Euclidean distance metric.
- *Merging* computes a non-uniform fast Fourier transform (NUFFT) of the autocorrelation of the electron density. This step produces a new estimate of the electron density.
- *Phasing* converts the 3D diffraction volume into a molecular structure, which is used to refine the electron density computed in the merging phase.

The parallel algorithm that supports multiple conformations reuses the components of the single conformation algorithm but rearranges them in ways that result in a very different overall structure. There is an additional level of parallelism, as each of the phases is now carried out for each of several conformations, and there are two additional phases, one to cluster the diffraction patterns by conformation and one to detect when a conformation has converged. The changes also result in new communication patterns between and within some of the phases. Specifically:

- *Slicing* is done per conformation.
- *Orientation matching* compares each of the 2D diffraction patterns from SPI to the model slices for every conformation and the closest Euclidean distance is computed.
- *Conformation* assigns each 2D diffraction pattern to a conformation based on the minimum Euclidean distance obtained across all conformations.
- *Merging* and *Phasing* compute and refine a new electron density estimate for each conformation.
- If enabled, *Convergence* determines whether each conformation has converged and its resolution. The results are used to determine which conformations should continue to the next iteration—converged conformations are removed from the computation.

4 Pygion Implementation

Once the single conformation algorithm was implemented in MPI and Pygion the project began working towards the multiple conformation algorithm. The multiple conformation algorithm described in Section 3 is the end result of an iterative process in which many variations were explored, each of which required time to code and test. This iterative process progressed much more quickly with the Pygion version than with the MPI version for two primary reasons.

First, many changes to the code involved adding or removing *tasks*, which are just distinguished functions that can be executed asynchronously. In Pygion, calling a new task simply meant writing that task and adding it in the appropriate place in the sequential execution order. The Legion runtime that underlies Pygion performs a dependence analysis that automatically preserves

sequential execution semantics while extracting parallelism and also inserts all needed communication and synchronization between tasks. Thus adding (or removing) a task is a *local* program change in Pygion—even though a task addition (or removal) can have global effects on the dependence graph of tasks, the dependency information is computed by Pygion and is not the responsibility of the programmer. In the MPI version, however, adding (or removing) a task involves more than the task call itself. Because the programmer is responsible for synchronization and communication in MPI, the programmer must manually determine how the synchronization and communication by other parts of the code must be modified to ensure correctness in light of the insertion or removal of a task, which in general can require *global* changes to the program.

Second, code changes often required creating new data structures, partitioning data in new ways, or both. Pygion has first-class support for *regions* (data collections) and for partitioning and distributing regions across the machine. In SpiniFEL a region of all the 2D diffraction patterns is partitioned into n subregions, where n is the number of *ranks* (the number of GPUs used in the computation). Pygion/Legion also allows multiple different partitions of the same region to exist and be used simultaneously. Another partitioning of the 2D diffraction patterns, for example, keeps track of which conformation each image belongs to. The conformation phase may update an image's assigned conformation each iteration; any updates are tracked automatically by Legion and when a merging task needs the images associated with a particular conformation the correct set of images is delivered to the task by the runtime system, reflecting all of the changes up to that point in the computation.

Pygion/Legion's built-in support for regions and partitioning means that adding a new kind of data collection or a new partitioning of an existing collection is also always a local change to the program, even though the communication pattern of the program may change globally through the new dependencies implied by changing or adding partitions. As with adding tasks, adding new regions or partitions in the MPI version requires that explicit synchronization and communication be added for each of those new dependencies, as it is the programmer's job, not MPI's, to communicate changes to data structures.

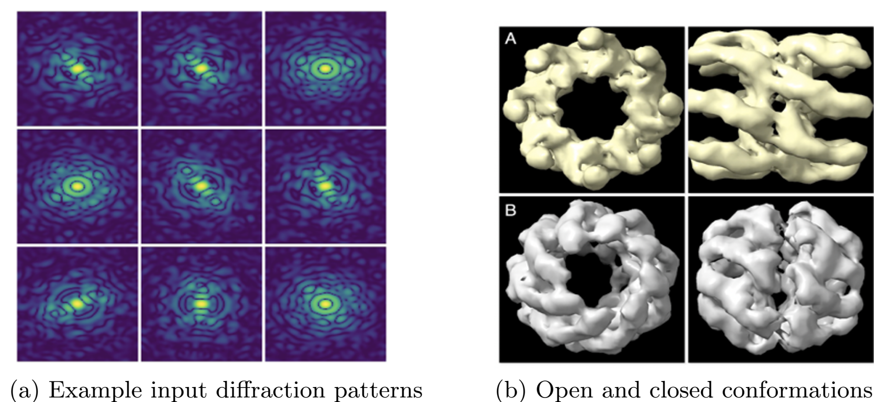
Note that partitioning in Legion is a dynamic operation: partitions are computed at runtime, and regions can be re-partitioned on the fly. In SpiniFEL dynamically creating new regions and partitions was important for integrating with Psana2, because Psana2 delivers the diffraction images periodically as the images arrive from the laser's data-gathering detectors. We use the ability to define regions and partitions dynamically to add new 2D diffraction patterns during each iteration—specifically a new region is created by taking the union of a region of the new images with the existing image region.

Figure 1 gives an excerpt from the merging phase of SpiniFEL. Shown are two *index launches*, loops that launch sets of tasks distinguished by an index i. The `solve_simple_adjoint` tasks each take three regions (among others that are elided): `slices_p[i]`, `ugrid`, and `uvect`; for each `solve_simple_adjoint` task there is a corresponding `solve_simple_linear` task in the second index

```
...
for i in IndexLaunch(N_procs):
    solve_simple_adjoint(..., slices_p[i], ugrid, uvect, ...)
for i in IndexLaunch(N_procs):
    solve_simple_linear(..., slices_p[i], ugrid, uvect, ...)
...
```

Fig. 1. An excerpt from SpiniFEL's merging phase.

launch that takes the same arguments. Not shown are how the tasks use these regions: the `solve_simple_adjoint` tasks read their set of slices and perform reductions into the `ugrid` and `uvect` regions, while the `solve_simple_linear` tasks only read form these regions. Note that there is no explicit parallelism or communication—Legion automatically discovers which tasks can run in parallel and where communication is needed. Adding, removing, or modifying a task to take different arguments are always local changes, regardless of the (potentially large) effect on the program's communication pattern.



(a) Example input diffraction patterns (b) Open and closed conformations

Fig. 2. Input and output of SpiniFEL

5 Results

We obtained results for 4096 ranks (GPUs) on the Frontier supercomputer at Oak Ridge National Laboratory. This experiment computed two conformations of Mm-cpn (the molecule methancoccous maripaludis chaperonin): 3IYF (open) and 3J03 (closed). Figure 2(a) shows examples of some of the input diffraction patterns, while Figure 2(b) shows the output conformations; the images labeled (A) are two views of the 3D electron density structure in the open state, while (B) shows the closed state after 20 iterations.

Weak scaling results for 256 images per rank and 20 iterations on Frontier are shown in Figure 3. SpiniFEL achieves almost perfect weak scaling to 2000 ranks, after which the communication between some of the phases begins to be exposed, resulting in about 70% parallel efficiency at 4000 ranks.

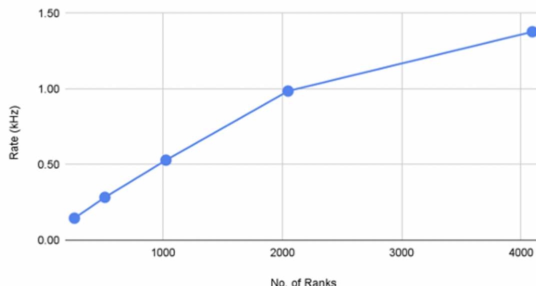


Fig. 3. Weak scaling results

We also tested convergence for both conformations. The table below gives the results for 16,384 images for 30 iterations. In the table gen refers to the number of generations (iterations) until convergence, resolution is in angstroms, and corr. coeff. is the final correlation coefficient of the the electron density map.

ranks	images/rank	3iyf			3j03		
		gen	resolution	corr. coeff.	gen	resolution	corr. coeff.
128	128	18	16.4	0.837	13	13.3	0.835
256	64	14	18.09	0.806	17	13.0	0.834

We cannot compare the Pygion code with MPI on multiple conformations because there is no MPI implementation. The single conformation MPI code is 50% slower than the corresponding Pygion code, primarily because Pygion handles the merging phase better [5]. In principle the MPI version could reproduce the performance of Pygion, but as discussed it would require more effort due to the need to make all communication and synchronization explicit.

6 Conclusion

Using Pygion we implemented SpiniFEL, a scalable, parallel code to reconstruct multiple molecular conformations from single particle imaging experiments. When the code reached sufficient complexity, the time and effort to add new features to a task-based code turned out to be much less than modifying an MPI code, primarily because the implicit parallelism of the task-based system made most changes local (only a small part of the program needed to be changed) in contrast to the MPI version, where often considerable effort was needed to express changes in the communication/dependency structure with numerous, non-local additions of MPI communications and synchronization.

Acknowledgments. 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. This research used resources of

the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Disclosure of Interests. The authors have no competing interests for this publication.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**, 187–198 (Feb 2011)
2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: High Performance Computing, Networking, Storage and Analysis (SC) (2012)
3. Blaschke, J., Mirchandaney, S., Yoon, C., Slaughter, E., Uervirojnangkoorn, M., Chang, I., Dujardin, A., Kommera, P., Ramakrishnaiah, V.B., Sweeney, C.: MTIP single particle imaging (SpiniFEL) (Oct 2021), https://www.osti.gov//servlets/purl/1834376
4. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering **15**(6), 36–45 (2013)
5. Chang, H.Y., Slaughter, E., Mirchandaney, S., Donatelli, J., Yoon, C.H.: Scaling and acceleration of three-dimensional structure determination for single-particle imaging experiments with SpiniFEL. arXiv preprint arXiv:2109.05339 (2021)
6. Damiani, D., Dubrovin, M., Gaponenko, I., Kroeger, W., Lane, T., Mitra, A., O'Grady, C., Salnikov, A., Sanchez-Gonzalez, A., Schneider, D., et al.: Linac coherent light source data analysis using Psana. Journal of Applied Crystallography **49**(2), 672–679 (2016)
7. Kalé, L.V., Krishnan, S.: CHARM++: A portable concurrent object oriented system based on C++. In: OOPSLA. pp. 91–108 (1993)
8. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., et al.: Ray: A distributed framework for emerging AI applications. In: 13th USENIX symposium on operating systems design and implementation (OSDI 18). pp. 561–577 (2018)
9. Slaughter, E., Aiken, A.: Pygion: Flexible, scalable task-based parallelism with Python. In: Proceedings of the Parallel Applications Workshop, Alternatives To MPI. pp. 58–72. IEEE (2019)
10. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: A high-productivity programming language for HPC with logical regions. In: High Performance Computing, Networking, Storage and Analysis (SC) (2015)
11. Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K.S., Cao, Q., Bosilca, G., Mirchandaney, S., Lee, W., Treichler, S., McCormick, P., Aiken, A.: Task Bench: A parameterized benchmark for evaluating parallel runtime performance. In: Supercomputing (SC). pp. 1–15. IEEE (2020)