A framework for *in situ* and in-transit analysis of cosmological simulations

Brian Friesen Ann Almgren Zarija Lukić Gunther Weber
Dmitriy Morozov Vincent Beckner Marcus Day

April 28, 2016

Abstract

Modern cosmological simulations have reached the trillion-element scale, rendering data storage and subsequent analysis formidable tasks. To address this circumstance, we present a new MPI-parallel approach for analysis of simulation data while the simulation runs, as an alternative to the traditional workflow consisting of periodically saving large data sets to disk for subsequent "offline" analysis. We demonstrate this approach in the compressible gasdynamics/N-body code Nyx, a hybrid MPI+OpenMP code based on the BoxLib framework, used for large-scale cosmological simulations. We have enabled on-the-fly workflows in two different ways: one is a straightforward approach consisting of all MPI processes periodically halting the main simulation and analyzing each component of data that they own ("*in situ*"). The other consists of partitioning processes into disjoint MPI groups, with one performing the simulation and periodically sending data to the other "sidecar" group, which post-processes it while the simulation continues ("in-transit"). The two groups execute their tasks asynchronously, stopping only to synchronize when a new set of simulation data needs to be analyzed. For both the *in situ* and in-transit approaches, we experiment with two different analysis suites with distinct performance behavior: one which finds dark matter halos in the simulation using merge trees to calculate the mass contained within isodensity contours, and another which calculates probability distribution functions and power spectra of various fields in the simulation. Both are common analysis tasks for cosmology, and both result in summary statistics significantly smaller than the original data set. We study the behavior of each type of analysis in each workflow in order to determine the optimal configuration for the different data analysis algorithms.

1 Introduction

Data analysis and visualization are critical components of large-scale scientific computing [1, 2, 3, 4, 5]. Historically such workflows have consisted of running each simulation on a static compute partition and periodically writing raw simulation data to disk for "post-processing."

Common tasks include visualization and size reduction of data, e.g., calculating statistics, field moments, etc. Other tasks can be domain-specific: for example, evolving large nuclear reaction networks on passively advected tracer particles in supernova simulations [6, 7, 8]. Often the data footprint of the output from these post-processing tasks is much smaller than that of the original simulation data [4, 9].

As simulations grow larger, however, this approach becomes less feasible due to disk bandwidth constraints as well as limited disk capacity. Data analysis requirements are outpacing the performance of parallel file systems, and, without modifications to either workflows or hardware (or both), the current disk-based data management infrastructure will limit scientific productivity [1, 10]. One way to avoid exposure to the increasingly disparate performance of disk I/O vs. inter- and intra-node bandwidth is to limit the volume of data which is written to disk. This strategy can be realized in different ways; one approach is simply to write data relatively infrequently, e.g., every large number of time steps when evolving time-dependent problems. However, limiting the number of time steps at which grid data is saved in order to conserve disk space also discards simulation data by "coarsening" in the temporal dimension [3]. For example, in order to produce a mock galaxy catalog from an N-body simulation, it is essential to create halo merger trees, which describe the hierarchical mass assembly of dark matter halos [11]. It is, however, well recognized that in order to recover converged merger trees, identification of halos with high temporal resolution is needed [12].

A second strategy for addressing the I/O problem is to shift data analysis from executing "offline" (on disk; [4]) to running while the simulation data is still in memory. Such a workflow can be expressed in a myriad of ways, two of which we explore in this work. One approach consists of all MPI processes periodically halting the simulation and executing analysis routines on the data in memory (*"in situ"*). The second method consists of dividing the processes into two disjoint groups; one group evolves the simulation and periodically sends its data to the other, which performs the analysis while the simulation continues asynchronously ("in-transit;" e.g., [10]). While *in situ* approaches have long been recognized as efficient ways of avoiding I/O, less attention has been devoted to in-transit methods.

In situ and in-transit methods each have potential strengths and weaknesses. The former method requires no data movement beyond what is inherent to the analysis being performed. Its implementation is also relatively non-invasive to existing code bases, consisting often of adding a few strategically placed function calls at the end of a "main loop." However, if the analysis and simulation algorithms exhibit disparate scaling behavior, the performance of the entire code may suffer, since all MPI processes are required to execute both algorithms. In-transit methods, on the other hand, lead to more complex workflows and more invasive code changes, which may be undesirable [4]. They also require significant data movement, either across an interconnect or perhaps via specialized I/O accelerator ("burst buffer"). However, they can be favorable in cases where the analysis code scales differently than that of the main simulation: since the analysis can run on a small, separate partition of MPI processes, the remaining processes can continue with the simulation asynchronously. This feature in particular may become especially salient as execution workflows of modern codes become more heterogeneous, since different components will likely exhibit different scaling behavior.

The nature of the post-processing analysis codes themselves also plays a role in the effectiveness of in-transit implementations. In many scientific computing workflows, the “main” simulation code performs a well defined task of evolving a physical system in time, e.g., solving a system of partial differential equations. As a result, its performance characteristics and science goals are relatively stationary. Analysis codes, in contrast, are implementations of a zoo of ideas for extracting scientific content from simulations. Being exploratory in nature, their goals are more ephemeral and heterogeneous than that of the simulation itself, which in general leads to more diverse performance behavior. The in-transit framework presented here provides the ability for analysis codes can be run together with the simulation, but without a strict requirement of being able to scale to a large number of cores. It is therefore useful to think of this in-transit capability as adding “sidecars” to the main vehicle: in addition to resources allocated exclusively for running the simulation, we allocate a set of resources (often much smaller) for auxiliary analysis tasks.

In this work we explore both *in situ* and in-transit data analysis workflows within the context of cosmological simulations which track the evolution of structure in the universe. Specifically, we have implemented both of these workflows in the BoxLib framework, and applied them to the compressible gasdynamics/N-body code Nyx, used for simulating large scale cosmological phenomena [13, 14]. We test each of these workflows on two different analysis codes which operate on Nyx data sets, one which locates dark matter halos, and another which calculates probability distribution functions (PDFs) and power spectra of various scalar fields in the simulation. In §2 we describe the scientific backdrop and motivation for the data analysis implementations which we have tested. In §3 we provide the details of our implementation of the *in situ* and in-transit workflows in the BoxLib framework. §4 contains a description of the two analysis codes which we explored using both *in situ* and in-transit methods. We discuss the performance of the two codes in each of the two analysis modes in §5, and we discuss prospects and future work in §6.

2 Cosmological simulations

Cosmological models attempt to link the observed distribution and evolution of matter in the universe with fundamental physical parameters. Some of these parameters serve as initial conditions for the universe, while others characterize the governing physics at the largest known scales [13, 15]. Numerical formulations of these models occupy a curious space in the data analysis landscape: on one hand, each cosmology simulation can be “scientifically rich” [4]: exploratory analysis of the simulation may lead to new insights of the governing physical model which could be lost if the raw data is reduced in memory and discarded. On the other hand, many models exhibit a highly nonlinear response to the initial perturbations imposed at high redshift, in which case isolated “heroic” simulations may not capture all of the features of interest which arise from such nonlinear behavior [13, 15]. Instead, one may wish to perform many such simulations and vary the initial conditions of each in order to capture the nuanced behavior of the models; such behavior can often be expressed even in a highly reduced set of data (e.g., a density power spectrum). We emphasize, then, that the data analysis methods presented here represent only a subset of techniques which will be required to manage the simulation data sets in future scales of computational cosmology.

The backdrop for the data post-processing methods described in this work is Nyx, a compressible gasdynamics/N-body particle code for cosmological simulations of baryonic and cold dark matter (CDM) [13, 14]. Nyx is based on BoxLib, an MPI+OpenMP parallelized, block-structured, adaptive mesh refinement (AMR) framework [16]. It models baryonic matter as a self-gravitating, compressible gas, composed of hydrogen and helium, and evolves it with a second-order accurate piecewise-parabolic method, using an unsplit Godunov scheme with full corner coupling [13, 17, 18]. It solves the Riemann problem iteratively using a two-shock approximation [19]. The CDM is treated as a non-relativistic, pressureless fluid, which is evolved with an N-body treatment, using a "cloud-in-cell" method [20] to project particles onto the AMR grid.

A typical Nyx simulation evolves a cosmology from high redshift ($z > 100$) to the present ($z = 0$) in many thousands of time steps. The size of each time step is limited by the standard CFL condition for the baryonic fluid, as well as by a quasi-CFL constraint imposed on the dark matter particles, and additionally by the evolution of the scale factor a in the Friedmann equation; details of these constraints are described in [13]. Currently the largest Nyx simulations are run on a 4096^3 grid, and at this scale each plotfile at a single time step is ~ 4 TiB, with checkpoint files being even larger; a complete data set for a single simulation therefore reaches well into the petascale regime. A single simulation can therefore fill up a typical user allocation of scratch disk space ($\mathcal{O}(10)$ TiB) in just a few time steps. We see then that modern simulation data sets represent a daunting challenge for both analysis and storage using current supercomputing technologies.

Nyx simulation data lends itself to a variety of popular post-processing cosmological analysis tasks. For example, in galaxies and galaxy clusters, observations have indicated that dark matter is distributed in roughly spherical "halos" that surround visible baryonic matter [15]. These halos provide insight into the formation of the largest and earliest gravitationally bound cosmological structures. Thus a common task performed on cosmological simulation data sets is determining the distribution, sizes, and merger histories of dark matter halos, which are identified as regions in simulations where the dark matter density is higher than some prescribed threshold. A recent review [21] enumerates 38 different algorithms commonly used to find halos. To process data from Nyx (an Eulerian code), we use a topological technique based on iso-density contours, as discussed in §4.1. The approach produces results similar to the "friends-of-friends" (FOF) algorithm used for particle data [15].

A second common data post-processing task in cosmological simulations is calculating statistical moments of different fields, like matter density, or Lyman-α flux. The first two moments — the PDF and power spectrum — are often of most interest in cosmological simulation analysis. Indeed, it is fair to say that modern cosmology is essentially the study of the statistics of density fluctuations, whether probed by photons, or by more massive tracers, such as galaxies. The power spectrum of these fluctuations is the most commonly used statistical measure for constraining cosmological parameters [22, 23, 24], and is one of the primary targets for numerical simulations. In addition to cosmology, one may be interested in predictions for astrophysical effects from these simulations, like the relationship between baryonic density ρ_b and temperature T in the intergalactic medium, or details of galaxy formation.

3 *In situ* vs. in-transit

Having established the scientific motivation for data post-processing in cosmological simulations, we now turn to the two methods we have implemented for performing on-the-fly data analysis in BoxLib codes.

3.1 *In situ*

To implement a simulation analysis tool *in situ* in BoxLib codes such as Nyx, one appends to the function `Amr::CoarseTimeStep()` a call to the desired analysis routine. All MPI processes which participate in the simulation execute the data analysis code at the conclusion of each time step, operating only on their own sets of grid data. As discussed earlier, the advantages of this execution model are that it is minimally invasive to the existing code base, and that it requires no data movement (except that inherent to the analysis itself). One potential disadvantage of *in situ* analysis is that if the analysis algorithm does not scale as well as the simulation itself, the execution time of the entire code (simulation + analysis) will suffer. Indeed, we encounter exactly this bottleneck when calculating power spectra, which we discuss in §4.2.

3.2 In-transit

The in-transit implementation of data analysis codes in Nyx is more complex than the *in situ* approach, due to the necessary data movement and the asynchrony of the calculation. During initialization, BoxLib splits its global MPI communicator into two disjoint communicators, `m_comm_comp` for the group which executes the simulation, and `m_comm_sidecar` for the "sidecar" group which performs the analysis. The user prescribes the sizes of each group at runtime, and the sizes are fixed for the duration of code execution. Upon reaching a time step at which analysis is requested, Nyx transfers via MPI the requisite data from the compute group to the sidecar group. Some of this data is copied to every sidecar process, e.g., the geometric information of the problem domain and how Boxes are arranged on the domain; to communicate such data, Nyx performs an intergroup `MPI_Bcast()`. The bulk of the data to be communicated consists of the floating-point state field stored in each Box; in BoxLib these data are called Fortran Array Boxes (FABs). Because we have two MPI groups and two communicators when executing in-transit (as well as an intergroup communicator conecting the two), we generate two "distribution maps" for the simulation data, one describing the distribution of FABs across processes in the compute group, and the other in the sidecar group. This provides BoxLib with a bijective mapping of the FAB data distribution between the two groups, allowing us to perform point-to-point intergroup `MPI_Send()`s and `MPI_Recv()`s to transfer the data between corresponding processes in the two groups. We summarize the method for sending and receiving this data in Algorithm 1.

```
if I am a compute proc then
    broadcast Box list to analysis procs;
    receive distribution map on analysis procs;
    foreach FAB I own do
        get global FAB index;
        find which sidecar proc will own FAB;
        send FAB data to sidecar proc;
    end
else
    receive Box list from compute procs;
    generate new distribution map on sidecar group;
    broadcast distribution map to compute procs;
    foreach FAB I will receive do
        get global FAB index;
        find which compute proc owns FAB;
        receive FAB from compute proc;
    end
end
```

Algorithm 1: Data movement logic when sending distributed grid data from compute processes to sidecar processes via MPI.

The distribution of data over each MPI group need not be the same, since each group can have arbitrary size. For example, if the simulation contains 4096 Boxes and the compute group has 2048 processes, each process will own 2 Boxes; however, if the sidecar group has only 256 processes, each process will own 16 Boxes. After the FABs have been sent to the analysis group, that group executes the desired analysis code, while the compute group continues with the simulation. A schematic of this Box movement across MPI groups is depicted in Figure 1.

The receipt of FABs onto a single MPI process is an inherently serial process. This property can affect code performance adversely if a large number of compute processes send their data to a small number of sidecar processes, because a compute process cannot continue with the simulation until all of its FABs have been sent, and each sidecar process can receive only one FAB at a time. In the example shown in Figure 1, process 5 receives FABs from processes 1 and 3; if, by coincidence, process 5 receives all four of process 3's FABs in order, process 3 can continue with its next task before process 1; however, process 5 cannot continue until it has received all FABs from both processes 1 and 3. In this example the ratio of sending to receiving processes, $R \equiv N_s/N_r = 2$, is relatively small; the serialization

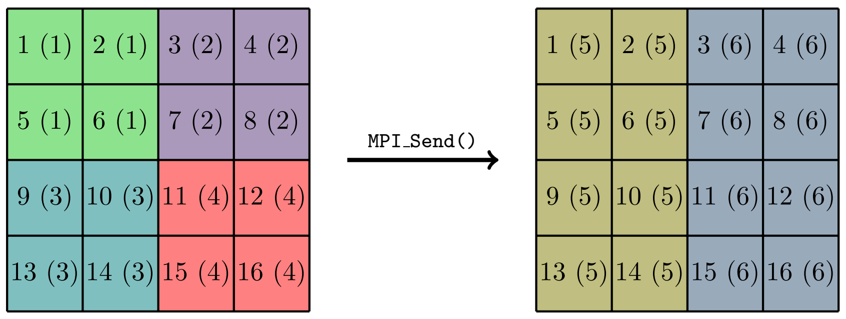

Figure 1: Example schematic illustrating data movement of block-structured grids from the simulation MPI group to the sidecar group when running in-transit. Each Box is uniquely numbered, and Boxes shaded in the same color are located on the same MPI process, whose rank is identified in parentheses. In this example, a grid composed of 16 Boxes moves from a distribution across 4 processes, with 4 Boxes per process, to a new distribution across 2 processes, with 8 Boxes per process.

of this data transfer will have only a minor effect on aggregate code performance. However, if $R \sim \mathcal{O}(100)$ or $\mathcal{O}(1000)$, the effect will be more pronounced.

3.3 Task scheduling

In both the *in situ* and in-transit workflows in BoxLib, we have added a simple queue-based, first-in-first-out scheduler which governs the order of data analysis tasks being performed. As we generally have a small number of tasks to perform during analysis, this approach is quite satisfactory. If the number of analysis tasks grows larger (a trend which we expect), then the workloads of each of these tasks will become more complex, and the variability in scaling behavior of each may be large as well. In this case a more sophisticated scheduling system — in particular one which accounts for a heuristic describing the scalability of each task and allocates sidecar partitions accordingly — may become more useful.

4 Cosmological simulation analysis tools

Many types of cosmological simulations can be broadly characterized by a small set of quantities which are derived (and highly reduced) from the original simulation data set. Such quantities include the distribution and sizes of dark matter halos, PDFs and power spectra of baryon density, dark matter density, temperature, Lyman-α optical depths, etc. [14]. We obtain these quantities from Nyx simulation data using two companion codes: Reeber, which uses topological methods to compute dark matter halo sizes and locations; and Gimlet, which computes statistical data of various fields. Because the algorithms in these codes are very different, the codes themselves exhibit different performance and scaling behavior. Therefore, they together span a useful parameter space for evaluating the utility of *in situ* and in-transit methods. In addition to operating on data in memory, both Reeber and Gimlet

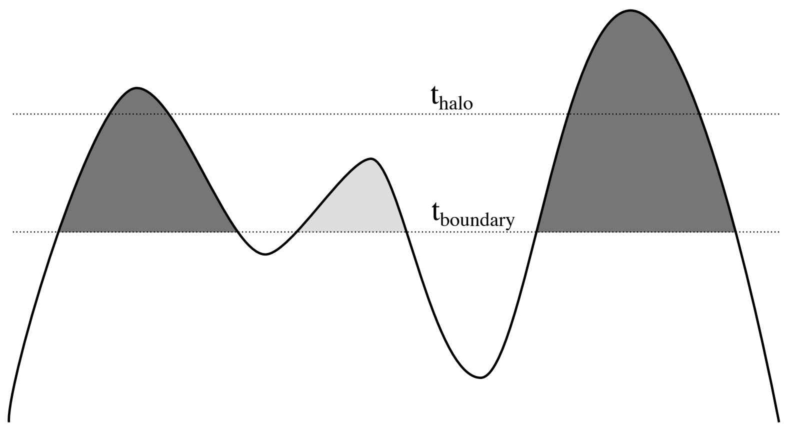

Figure 2: Halo definition based on iso-density contours. Halos are regions above a density threshold t_{boundary} (light gray region) whose maximum density exceeds t_{halo} (dark gray regions).

are capable of running "offline," in the traditional post-processing workflow described in §1. We describe each of these codes below.

4.1 Reeber

Reeber is a topological analysis code, which constructs merge trees of scalar fields. A merge tree describes the relationship among the components of super-level sets, i.e., regions of the data set with values above a given threshold. The leaves of a merge tree represent maxima; its internal nodes correspond to saddles where different components merge; its root corresponds to the global minimum [25].

An illustration of Reeber's halo-finding algorithm is shown in Figure 2. To identify iso-density-based halos efficiently, we traverse the merge tree upwards from the root, finding all edges that cross the value t_{boundary}. This operation corresponds to drawing the merge tree such that the height of each node corresponds to its function value and identifying all sub-trees above a line at the height of t_{boundary}. We then traverse each sub-tree to identify its highest maximum. If this maximum exceeds t_{halo}, the sub-tree represents a halo, and we compute its position as the centroid of all grid points belonging to the sub-tree as well as its mass as the cell-size-weighted sum of all grid points belonging to the sub-tree.

As a topological descriptor, a merge tree is primarily a means to an end: its utility comes from providing an efficient way to query a field for interesting topological information, e.g., are two regions of a field connected above a given threshold, or what is the number of connected components at density ρ whose density maximum is higher than ρ'? In Nyx simulations, one requires the answer to exactly these questions when identifying the locations, distribution, and merger histories of dark matter halos, as discussed in §2. Given the dark matter density field and its boundary conditions, Reeber first constructs its merge tree, and then uses it to identify halos based on user-defined density thresholds [26]. The merge tree

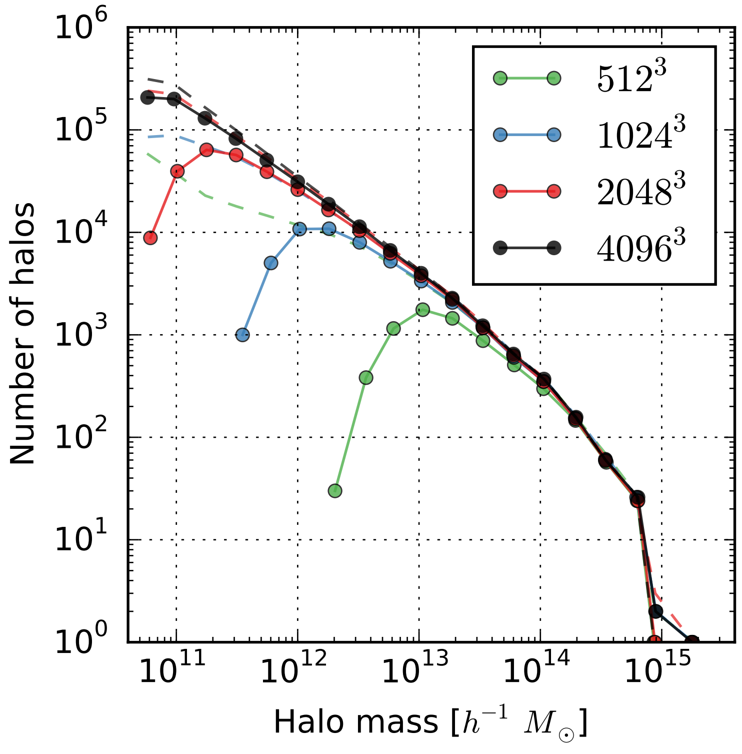

Figure 3: Convergence of the halo mass function in Nyx simulations with the Reeber halo finding code. Solid lines demonstrate how Reeber's distribution of halo masses change when increasing the spatial resolution in Nyx runs. As expected, we observe the differences only at the low-mass end, since coarse grids cannot capture well small halos, while the agreement on the high-mass end is good. The dashed lines show results of a FOF halo finder when the linking length parameter is chosen to match approximately the iso-density contour used in Reeber. FOF results are used as a validation here, showing that Reeber results converge to the "correct" answer.

itself does not depend on any parameters, only on the input function. Accordingly, one can repeatedly query the same tree with different halo thresholds without reconstructing it each time. A recent result applying Reeber to various Nyx simulation data sets is shown in Figure 3.

Implementing a scalable representation of merge trees on distributed memory systems is a challenging but critical endeavor. Because the merge tree is a global representation of the entire field, traditional approaches for distributing the tree across independent processes inevitably require communication-intensive reduction to construct the final tree, which in turn lead to poor scalability. Furthermore, modern simulations operate on data sets that are too large for all topological information to fit on a single compute node. Reeber's "local–global" representation addresses this problem by distributing the merge tree, so that each node stores detailed information about its local data, together with information about how the local data fits into the global merge tree. The overhead from the extra information is minimal, yet it allows individual processors to globally identify components of super-level sets without any communication [25]. As a result, the merge trees can be queried in a distributed way, where each processor is responsible for answering the query with respect to its local data, and a simple reduction is sufficient to add up contribution from different processes. A detailed description of merge trees, contour trees, and their "local-global" representation, which allows Reeber to scale efficiently on distributed memory systems, is given in [25, 27] and its application to halo finding in [26].

4.2 Gimlet

Gimlet calculates a variety of quantities relevant for the intergalactic medium studies, which are derived from different fields in Nyx, including:

- optical depth and flux of Lyman-α radiation along each axis
- mean Lyman-α flux along each axis
- 2-D PDF of temperature vs. baryon density
- PDF and power spectrum of Lyman-α flux along each axis
- PDF and power spectrum of each of
  - baryon density
  - dark matter density
  - total matter density
  - neutral hydrogen density

The details of these algorithms are described in [14]. The most interesting quantities calculated by Gimlet are power spectra of the Lyman-α flux and matter density. Gimlet uses FFTW3 [28] to calculate the discrete Fourier transformation (DFT) of the grid data required for power spectra. We note here also that FFTW3's domain decomposition strategy for 3-D DFTs has implications which are especially relevant for a study of *in situ* and in-transit analysis performance. We discuss these in §5.

Given a 3-D scalar field, Gimlet calculates its DFT in two different ways:

1. It divides the grid into one-cell-thick "rays" which span the length of the entire problem domain. All rays are aligned along one of the 3 axes. It then compute the 1-D DFT and along each ray individually, accumulating the results into a power spectrum for the entire grid. This approach captures line-of-sight effects of the cosmology simulation, as discussed in [14]. Each DFT can be executed without MPI or domain decomposition, since the memory footprint of each ray is small, even for large problem domains.

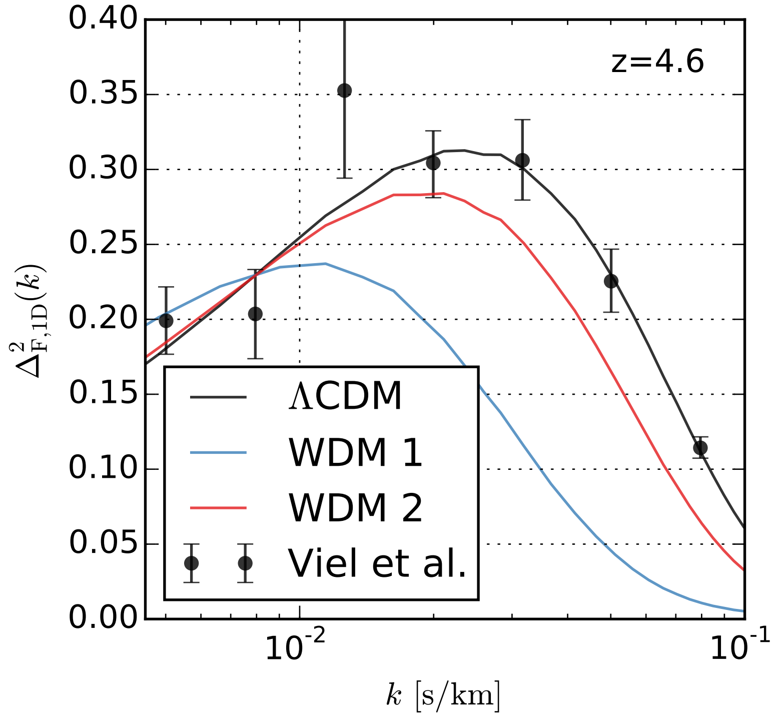

Figure 4: Power spectrum of Lyman-α flux from 3 Nyx simulations using the Gimlet analysis code, compared to observational data presented in [29]. The black line is the result of a ΛCDM cosmological model with the reionization history described in [30]. The blue and red lines are two WDM models, differing in their choice of dark matter particle mass: $m_{\mathrm{DM}} = 0.85$ keV (blue) and $m_{\mathrm{DM}} = 2.1$ keV (red).

2. It computes the 3-D DFT and power spectrum of the entire grid. This requires exploiting FFTW3's domain decomposition features enabled with MPI.

As an example of Gimlet application, we show in Figure 4 a comparison between the observed Lyman-α flux power spectrum, and predictions from 3 Nyx simulations. We plot a dimensionless power spectrum calculated along the line of sight versus the wavelength mode k. We show here one redshift only using data from [29], and we demonstrate how power spectra differ when changing the thermal velocity dispersion of the dark matter. The black line is the cold dark matter model which has no thermal velocity component in the initial state. The blue and red lines correspond to two different warm dark matter (WDM) models, $m_{\mathrm{DM}} = 0.85$ keV and $m_{\mathrm{DM}} = 2.1$ keV, respectively. The main task of Nyx simulations with the Gimlet analysis pipeline is to determine which cosmological model and reionization history fits the best existing observational data, and Figure 4 is a simple example when we vary only one parameter out of ~ 10.

The two basic types of calculations Gimlet performs — PDFs and power spectra — exhibit quite different performance behavior. PDFs scale well, since each MPI process bins only local data. A single `MPI_Reduce()` accumulates the data bins onto a single process, which then writes it to disk. Power spectra, on the other hand, require calculating communication-intensive DFTs. The need to reorganize data to fit domain decomposition required by the FFTW3 library — which differs from the native Nyx decomposition — incurs additional expense. Gimlet's overall scalability, therefore, is a convolution of these two extremes.

5 Performance

In this section we examine detailed performance figures for the *in situ* and in-transit workflows described above. All of these tests were performed on Edison, a Cray XC30 supercomputing system, at the National Energy Research Scientific Computing Center (NERSC). Edison's compute partition consists of 5576 nodes, each configured with two 12-core Intel Xeon "Ivy Bridge" processors at 2.4 GHz, and 64 GB of DDR3 memory at 1866 MHz. Compute nodes communicate using a Cray Aries interconnect which has "dragonfly" topology. First we present results of some synthetic performance benchmarks, in order to establish baselines by which to compare the performance of real analyses performed by the Reeber and Gimlet. Then we present the results from the analysis codes themselves.

5.1 Lustre file write performance

In situ and in-transit methods attempt to circumvent the limitations of not only disk capacity, but also disk bandwidth. It is therefore of interest to this study to measure the time a code would normally spend saving the required data to disk in the traditional post-processing workflow. To do so, we measured the time to write to write grids of various sizes, each containing 10 state variables[1], to the Lustre file system on Edison. BoxLib writes simulation data in parallel using `std::ostream::write` to individual files; it does not write to shared files. The user specifies the number of files over which to distribute the simulation data, and BoxLib in turn divides those files among its MPI processes. Each process writes only its own data, and only one process writes to a given file at a time, although a single file may ultimately contain data from multiple processes. The maximum number of files allowable is the number of processes, such that each process writes its own data to a separate file.

We varied the size of the simulation grid from 128^3 to 2048^3, divided among Boxes of size 32^3. This is a small Box size for Lyman-α simulations which typically do not use mesh refinement; however, it is entirely typical for many other BoxLib-based applications which perform AMR. The maximum number of processes for each test was 8192, although for tests which had fewer than 8192 total Boxes (namely, those with 128^3 and 256^3 simulation domains), we set the number of processes such that each process owned at least 1 Box. We also varied the number of total files used, from 32 up to 8192, except in cases where there were fewer than 8192 total Boxes.

We illustrate the file write performance on Lustre in Figure 5. We find that, for the largest grid tested (2048^3), the highest achievable write bandwidth on the Edison Lustre file

[1]In the following text we refer to this simply as a "10-component grid."

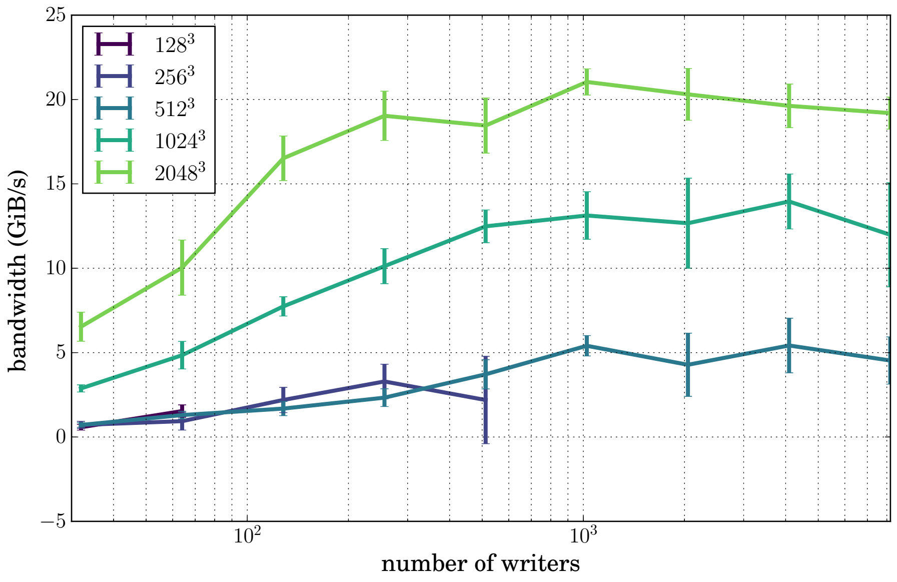

Figure 5: Aggregate bandwidth (GiB/sec) for writing 10-component simulation grids of varying sizes to Lustre. Each data point shows the statistical mean over 5 writes, with the standard deviation shown in error bars. The lack of data points for the 128^3 and 256^3 grids at large numbers of writers are configurations with more MPI processes than total Boxes, such that some portion of processes would write no data.

system is ~ 20 GiB/sec. For the 512^3 grid, which serves as our test case when we explore the performance of MPI traffic when running Nyx analysis codes in-transit in §5, the highest write bandwidth is ~ 5 GiB/sec.

5.2 In-transit MPI performance

Before exploring the analysis workflow performance study which is presented in §5, here we first perform two simple studies which measure the time required to move grid data from one MPI group to another. In one test we used a 10-component grid of size 1024^3, which has a total memory footprint of ~ 80 GiB. The grid was divided into Boxes of size 32^3, yielding 32 768 total Boxes. We then fixed the total number of MPI processes at 8192, and varied the number of analysis processes from 64 to 4096 (with the size of the compute group varying from 8128 to 4096 processes). The results for this test are shown in Figure 6. In the second test, we fixed the number of total processes at 8192 and also fixed the number of analysis processes at 1024, leaving 7168 processes in the compute group. We then varied the size of the grid to be transferred from 128^3 (64 Boxes with total size 156 MiB) to 2048^3 (262 144 Boxes with total size 640 GiB). The results are presented in Figure 7.

We see from this figure that the fastest bandwidth we achieve across the interconnect is ~ 58 GiB/s. The peak bandwidth for the entire Aries interconnect on Edison is 23.7 TB/s

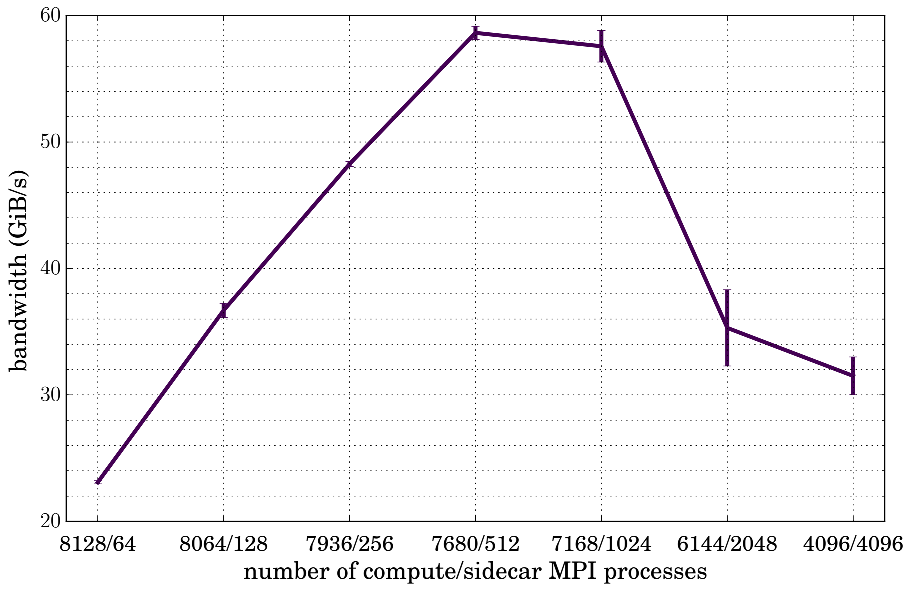

Figure 6: Total bandwidth during transfer of a 10-component 1024^3 grid (32 768 Boxes) among 8192 total MPI processes, with varying sizes of compute and analysis groups. The standard deviation over 5 iterations is indicated with error bars.

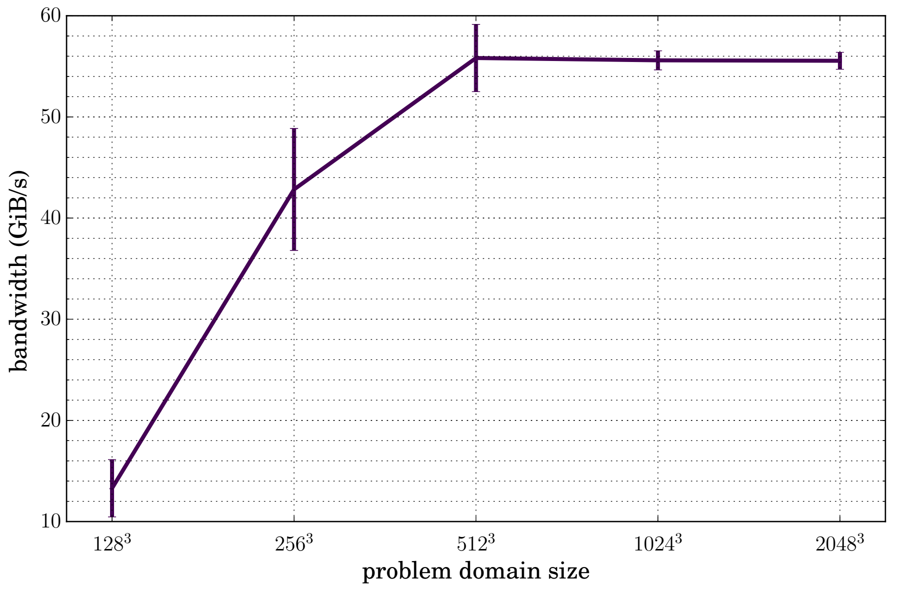

Figure 7: Total bandwidth during transfer of 10-component grids of varying sizes from 7168 compute MPI processes to 1024 analysis MPI processes. The standard deviation over 5 iterations is indicated with error bars.

($\sim$ 21.6 TiB/s) distributed across 5576 compute nodes, indicating a peak bandwidth per node of $\sim$ 4.0 GiB/s. From our test, the highest bandwidth per node was $\sim$ 116 MiB/s, only about 3% of peak. Since several different configurations in this test appear to plateau at the same bandwidth, this limit may be due to high latency costs associated with sending and receiving so many small Boxes. Were the Boxes much larger, as they often are in Lyman-α simulations, the MPI traffic would consist of fewer and larger messages, which may achieve higher per-node bandwidth. However, even performing so far below the peak, the bandwidth of moving Boxes across the interconnect for the 512^3 grid is still a factor of 10 faster than saving it to disk on the Edison Lustre file system (cf. Figure 5).

From this simple study we can estimate that, in terms of pure transfer speed (neglecting analysis or simulation scaling behavior), the optimal ratio of compute to analysis on Edison is $R \sim 9$. For analysis algorithms which strong scale efficiently, this should be a useful metric. The poor bandwidth for the small grids in Figure 7 arises because there are more MPI processes than Boxes, so many processes do not partake in the transfer process; the simulation therefore has access to a smaller portion of the aggregate available bandwidth on the interconnect. In all cases, the total transfer time is on the order of seconds. If analysis is performed infrequently, this comprises a small component of the total run time of most Nyx simulations, which often take $\gtrsim \mathcal{O}(10^5)$ sec. However as the frequency of analysis increases, the total time spent moving data can become considerable, and an *in situ* approach may become more attractive.

The complexity of in-transit analysis presents a number of factors one should consider in order to optimize code performance. For example, the cost of moving large volumes of data across MPI groups via an interconnect is significant, but can nevertheless be dwarfed by the cost of the analysis computation itself. Additionally, some analysis algorithms scale poorly – significantly worse than the simulation – in which case incurring the penalty for moving data to a small set of sidecar processes so that the simulation can continue may lead to the best overall performance of the code. On the other hand, the data movement penalty makes in-transit data processing impractical for applications which are already inexpensive to calculate, or which scale very well, or both. These types of analysis may be more amenable to *in situ* approaches.

In-transit analysis introduces an additional layer of load balancing complexity, in that an optimally performant simulation maximizes the asynchrony of computation and analysis. To illustrate this point, suppose a simulation for evolving a system forward in time reserves C MPI processes for computation and A for analysis. Denote by τ_c the time required for the compute processes to complete one time step of the simulation (without analysis), and τ_a the time for the analysis processes to complete analysis for one set of data. If the user requests that the analysis execute every n time steps, then an optimal configuration of compute and analysis groups would have $n\tau_c \simeq \tau_a$. If the former is larger, then the analysis group finishes too early and has no work to do while it waits for the next analysis signal; the converse is true if the latter is larger. If one finds that $n\tau_c > \tau_a$, then one option is to decrease A, the number of analysis processes. This will increase the τ_a but will simultaneously decrease τ_c since we assume that $C + A =$ const. A second option to equilibrate the two time scales is to decrease

n, although this option may not be useful in all cases, since n is likely driven by scientific constraints and not code performance. If one relaxes the restriction that $C + A = \text{const.}$, then one can adjust the size of the analysis group arbitrarily while keeping the compute group fixed in order to balance the two time scales τ_c and τ_a.

The risk of load imbalance described above traces its roots to the static nature of the roles of compute and analysis processes which we have assumed for this example, and which has traditionally been characteristic of large-scale simulations. Much of it could be ameliorated using dynamic simulation "steering," in which the simulation periodically analyzes its own load balance and adjusts the roles of its various processes accordingly. Returning to the above example, one may request an additional task from the analysis group: every m time steps, it measures the time spent in MPI calls between the the compute and analysis groups (such calls remain unmatched while one group is still working). If $n\tau_c > \tau_a$, then before the next time step the simulation re-sizes the MPI groups to increase C and decrease M, by an amount commensurate with the length of the time spent waiting for both groups to synchronize. An even more exotic solution would be to subsume *all* resources into the compute group until the nth time step, at which point the simulation spawns the analysis group on the spot, performs the analysis, and re-assimilates those processes after analysis is complete. The heuristics used to adjust the simulation configuration will likely be problem-dependent and will need to draw from a statistical sample of simulation performance data; we are actively pursuing this line of research.

5.3 Problem setup

Having established some synthetic benchmarks for disk and interconnect bandwidths, we now turn to the application of these workflows on science problems in Nyx. Here we present an exploratory performance analysis of *in situ* and in-transit implementations of both Reeber and Gimlet. We evolved a Lyman-α forest problem [14] on a 512^3 grid with a single level (no mesh refinement) for 30 time steps, resuming a previous simulation which stopped at redshift $z \sim 3$. We chose this cosmological epoch because this is often when the most "post-processing" is performed, due the wealth of observational data available for comparison [14, 31]. The physical domain for this problem spanned 10 Mpc per side, yielding a resolution of $\Delta x \sim 20$ kpc. The domain boundaries were periodic in all three dimensions. We ran the simulation in three different configurations: with no analysis, with Reeber, and with Gimlet.[2] In simulations which perform analysis, the analysis code executed every 5 time steps, starting at step 1. We choose this configuration such that the last time step during which analysis is performed is number 26; this gives the sidecar group a "complete" window of 5 simulation time steps (26-30) in which to perform the last analysis.

We divided the cubic domain into cubic Boxes of size 32^3, for a total of 4096 Boxes. When running *in situ*, we ran the simulation and analysis on 2048 MPI processes, and

[2]In production runs, one may wish to execute *both* analysis codes in a single simulation, but for the purposes of this performance study, we do not consider this case, as it obscures the performance behavior we seek to study.

decomposed the domain such that each MPI process worked on exactly 2 Boxes. For the in-transit mode, we chose the number of MPI processes in two different ways. First, we fixed the number of *total* processes at 2048, and varied the number of processes allocated to either the simulation or sidecars. This approach shows the optimal balance of each if the goal is to fit the simulation into a desired queue on a computational system which has a restriction on the maximum number of total processes. Our second approach was to fix the number of processes devoted to the simulation at 2048, and to vary the number of *additional* processes devoted to analysis. The total number of processes was then larger than 2048. This approach has in mind the case that the user wishes for the grid data to be distributed among the simulation cores in a particular way to preserve load balance, and that one is less concerned with the total number of processes being used. A simple example illustrates the influence of load imbalance across MPI processes: if we have 4096 Boxes spanning the domain but only 2047 processes evolving the simulation instead of 2048, two of those 2047 process must each operate on 3 Boxes, while the other 2045 processes operate on only 2. However, *all* processes must wait for the two processes which are computing 3 Boxes. Thus, decreasing the computational resources by 0.05% increases the total run time by 50%.

In all simulations, we ran Nyx, Reeber, and Gimlet using pure MPI, with no OpenMP. We used version 3.3.4.6 of FFTW3, and compiled all codes with the GNU compiler suite, version 5.2.0, with "`-O3`" optimization. In all DFT calculations, we used FFTW3's default "plan" flag, `FFTW_MEASURE`, to determine the optimal FFTW3 execution strategy.

5.4 Results

We summarize our performance results in Figures 8 & 9. There we compare various components of the simulation running both Reeber and Gimlet *in situ* and in-transit. Specifically, we plot the total end-to-end run times in solid lines, the time for each post-processing step (every 5th time step) as dashed lines, and the time to evolve the simulation 5 time steps with dot-dashed lines. For visual clarity, the in-transit runs have markers at each data point, while the *in situ* lines are unmarked, as they represent a single data point. Each data point for the post-processing execution time is an average over 6 iterations; the standard deviations illustrated with error bars would be smaller than the line thickness and are thus not indicated in either figure. Each total run time is a single data point since each complete configuration was run only once. The lines labeled "CT" used a constant number of *total* processes (2048 for these tests), such that, e.g., when using 16 sidecar processes, 2032 are running the simulation. Those labeled "CS" have a constant number of processes working on the simulation alone, such that when using 16 sidecar processes a total of $2048 + 16 = 2064$ processes are running. The *in situ* lines are horizontal and independent of the number of sidecar processes they use a fixed number of 2048 total processes, all of which perform both simulation and analysis. The CT and *in situ* configurations, therefore, always use a total of 2048 processes, while the CS configurations use > 2048.

The times to post-process (dashed lines in in each of the two figures) illustrate the strong scaling behavior of both analysis codes. The interplay among the different scalabilities presented here – those of the analysis suite, simulation code itself, and the combination

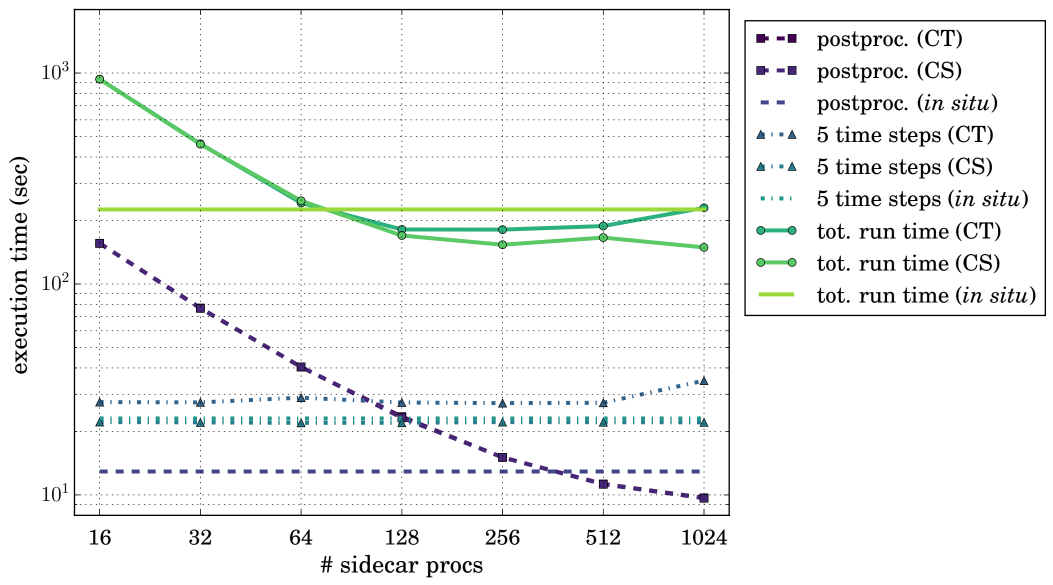

Figure 8: Performance of Reeber running *in situ* and in-transit with different distributions of MPI processes. The times indicated are wall clock seconds. We used two different in-transit configurations: once with a constant total number of 2048 MPI processes ("CT"), and once with a constant number of processes (2048) devoted to simulation ("CS"). Time time to post-processes for the CS and CT in-transit configurations are nearly identical.

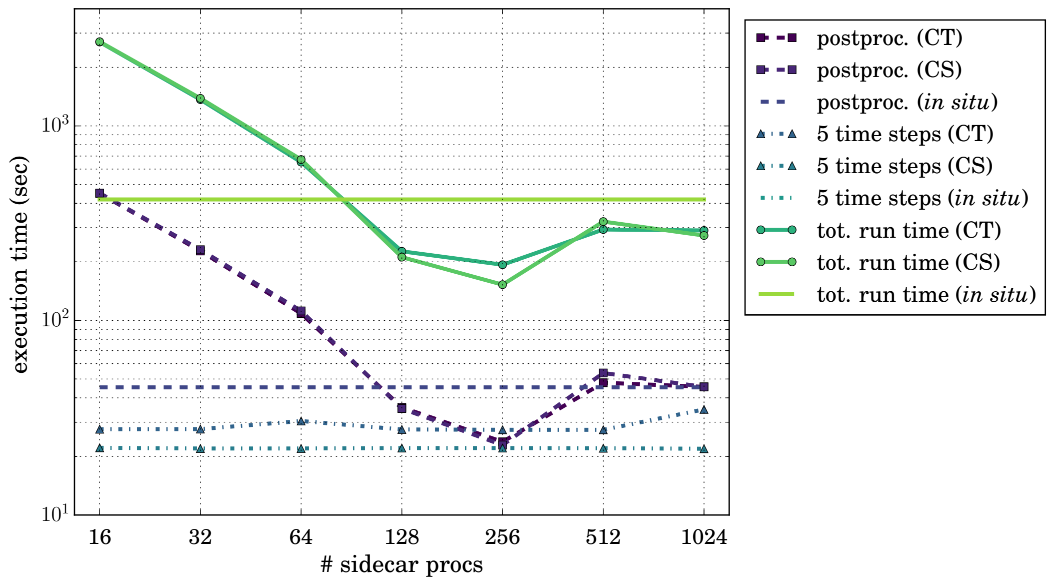

Figure 9: Same as Figure 8, except with Gimlet. Time time to post-processes for the CS and CT in-transit configurations are nearly identical.

of the two – are complex and require a nuanced interpretation. In Figure 8 we see that Reeber strong scales efficiently, although its performance begins to plateau at large numbers of MPI processes, due to the relatively small problem size (a 512^3 problem domain). We also note that Reeber running *in situ* on 2048 processes is actually slower than on 512 or 1024 processes in-transit, which is due to having too many processes operating on too small a problem.

Figure 8 also illustrates the relationship between the scalability of the analysis code alone and that of the entire code suite (simulation+analysis). In particular, we see that for ≤ 64 sidecar processes performing analysis, a single Reeber analysis call takes longer than the 5 intervening simulation time steps. As a result, the strong scaling behavior of the complete simulation+analysis suite mirrors that of Reeber almost exactly. However, for ≥ 128 sidecar processes, Reeber is equal to or faster than the 5 simulation time steps, such that the scalability of the entire code suite begins to decouple from that of Reeber alone. This behavior has different effects for the CS and CT in-transit configurations. For the CS configuration, the scalability of the code becomes asymptotically flat for ≥ 128 sidecar processes, because Reeber completes before the 5 simulation time steps are done. Any number of additional sidecar processes above ~ 256 is wasted in this mode. For the CT configuration, the end-to-end wall clock time begins to *increase* for ≥ 128 sidecar processes, because even though Reeber is executing faster and faster, increasing number of sidecar processes decreases the number of simulation processes, which slows the entire code down.

In light of this behavior, it is critical to find the configuration which yields the best performance for in-transit analysis. Deviating from this configuration leads either to wasted compute resources, or to a degradation of overall code performance. We note that the optimal setup for Reeber in both the CS and CT configurations for this particular Nyx simulation – using 128 to 256 sidecar processes for Reeber and the remainder for simulation – is faster than running the entire code suite *in situ*. The CT mode is especially appealing, as it uses the same number of MPI resources as the *in situ* mode (2048 processes total), and less than the corresponding CS configuration (2176 to 2304 processes).

Gimlet's scaling in Figure 9, however, is more complex. This is because Gimlet performs a variety of tasks, calculating both PDFs, which strong scale very well, and power spectra, which do not. Specifically, while BoxLib can decompose the problem domain into an arbitrary block structure, the only domain decomposition strategy which FFTW3 supports is to stripe along the first dimension in row-major array indexing, or the last dimension in column-major indexing. (FAB data in BoxLib uses the latter.) Therefore, the Nyx problem domain must be divided into "slabs" which span the entire x-y plane. Furthermore, the maximum number of slabs we can create is the number of grid points of the domain along the z-axis (512 in this case). Since FFTW3 allows only one MPI process to work on each slab, we are therefore limited to a total of 512 processes which can participate in the DFT calculation; the remaining processes (1536 when running *in situ*) are idle. This load imbalance becomes worse as the problem size grows: if using $\sim$ 100 000 processes to simulate a 4096^3 grid, then $\sim$ 96 000 cores will be idle during the DFT calculation. This slab decomposition problem has inspired the development of "pencil"-based decomposition strategies for 3-D

DFTs which provide better scaling behavior [32, 33]. If we run Gimlet in-transit instead of *in situ*, however, we can address this problem by choosing a relatively small number of processes to participate in the DFT, leaving the rest to continue with the simulation.

The aggregate performance of Gimlet therefore represents a convolution of the scaling properties of both PDFs and power spectra. Although the power spectrum calculation scaling behavior quickly saturates as discussed above, we expect nearly ideal strong scaling behavior for the calculation of PDFs. Therefore, when we see in Figure 9 that the time for analysis *increases* between 256 and 512 analysis processes, this is due to the DFT calculations. In particular, when calculating the DFT of a 512^3 grid on 512 MPI processes, each process has exactly one $x - y$ plane of data. The ratio of work-to-communication may be low for such an extreme decomposition, leading to worse performance with 512 processes than with 256. Despite the jump in analysis time between 256 and 512 processes, however, the time decreases once again between 512 and 1024 processes. When executing the DFT on 1024 processes, we leave 512 idle, so the time for that component is likely exactly the same as it was with 512; in fact, the DFT calculation time will *never* decrease for > 512 processes.[3] The decrease in total analysis time is instead due to the continued strong scaling of the PDF calculations.

The relationship between the scalability of Gimlet and the entire code suite is different than for Reeber, chiefly because Gimlet execution takes significantly longer than does Reeber. For almost any number of sidecar processes in Figure 9, the time to execute Gimlet is longer than the corresponding 5 simulation time steps. (For 256 sidecar processes the two times are roughly equivalent). As a result, Gimlet dominates the total code execution time, and the simulation+analysis strong scaling behavior follows that of Gimlet alone almost exactly.

As was the case with Reeber, we see in Figure 9 that some CS and CT configurations lead to faster end-to-end run times than *in situ*. Coincidentally, the threshold at which CT and CS runs become faster is between 64 and 128 sidecar processes, the same as Reeber. When using 128 sidecar processes, the CT and CS configurations are $\sim 30\%$ faster than the *in situ* run, whereas the optimal configurations with Reeber were only $\sim 15\%$ faster. These values are functions both of the scalability of the analysis codes being used, as well as the total time they take to execute. The key point is that for both analysis codes, one can construct an in-transit MPI configuration which is significantly faster than running *in situ*.

6 Summary and Prospects

In situ and in-transit data post-processing workflows represent a promising subset of capabilities which we believe will be required in order to be scientifically productive on current and future generations of computing platforms. They avoid the constraints of limited disk capacity and bandwidth by shifting the requisite data movement "upward" in the architec-

[3]For simulations in which the number of MPI processes calling FFTW3 is equal to or larger than the number of available "slabs", we could select an arbitrarily smaller number of processes to perform the FFTW3 call, since it seems that using the maximum possible number of slabs and MPI processes does not lead to the fastest performance of the DFT. However, the optimal number of processes will likely be problem-dependent.

tural hierarchy, that is, from disk to compute node memory. One can imagine continuing this trend to even finer levels of granularity, shifting from data movement between compute nodes across an interconnect, to movement across NUMA domains within a compute node. This could be implemented in several different ways; one would be to use “thread teams” introduced in in OpenMP 4.0 (which is already widely implemented in modern compilers) to delegate simulation and analysis tasks within a compute node. A second approach would be to extend the MPI implementation we have introduced in this work to incorporate the shared memory “windows” developed in MPI-3.

We have demonstrated the capability of these new *in situ* and in-transit capabilities in BoxLib by running two analysis codes in each of the two workflows. Although small in number, this sample of analyses — finding halos and calculation power spectra — is highly representative of post-processing tasks performed on cosmological simulation data. We caution, however, that the results presented in §5.4 are highly problem-dependent; although we found in-transit configurations which yield faster overall performance than *in situ* for both analysis codes, the situation may be different when running simulations on larger grids, using larger numbers of processes, running analysis algorithms with different scaling behavior, using a different frequency of analysis, etc. Furthermore, we highlight the caveat that, in some situations, on-the-fly data post-processing is not a useful tool, namely in exploratory calculations, which are and will continue to be critical components of numerical simulations. In these cases, other techniques will be more useful, including on-disk data compression. We anticipate, then, that a variety of tools and techniques will be required to solve these data-centric challenges in HPC.

Besides raw performance gains, *in situ* and in-transit workflows can improve the “time to science” for numerical simulations in other ways as well. For example, by running both simulation and analysis at the same time, one eliminates an extra step in the post-processing pipeline. This reduces the chance for human error which can arise when one must compile and run two separate codes with two separate sets of inputs, parameters, etc.

The *in situ* and in-transit workflows we have discussed are not limited purely to floating-point applications, even though that has been our focus in this work. Another critical component of simulation is visualization, and recently both the ParaView and VisIt frameworks have implemented functionality for performing visualization on data which resides in memory (“ParaView Catalyst” [34] and “libsim” [35]).

Our implementations of *in situ* and in-transit post-processing show that these types of workflows are efficient on current supercomputing systems: the expense of data movement via MPI in the latter workflow is, in the cases examined here, small compared to the total time spent performing simulation or analysis. Therefore, the penalty for trading disk space for CPU-hours (both of which are limited commodities) is not severe. While we have examined only two analysis codes in this work, in the future we will be able to evaluate the myriad other analysis workflows which are critical components of other BoxLib codes. Because we have built these capabilities into BoxLib itself, rather than into Nyx specifically, these workflows will support a wide variety of applications. The infrastructure described here will provide scientists working with BoxLib-based codes in astrophysics, subsurface flow, combustion, and porous media, an efficient way to manage and analyze the increasingly large datasets generated by their simulations.

7 Acknowledgements

We thank Brian Perry for valuable discussions. Most figures in this work were generated with matplotlib [36]. This work was in part supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. DOE under Contract No. DE-AC02-05CH11231 to the Lawrence Berkeley National Laboratory. ZL acknowledges the support by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research and the Office of High Energy Physics. DM and GHW acknowledge the support by the Scientific Data Management, Analysis and Visualization at Extreme Scale program funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research, program manager Lucy Nowell. Support for improvements to BoxLib to support the Nyx code and others was provided through the SciDAC FASTMath Institute. Calculations presented in this paper used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work made extensive use of the NASA Astrophysics Data System and of the astro-ph preprint archive at arXiv.org.

References

[1] R.B. Ross, et al., Journal of Physics: Conference Series **125**(1), 012099 (2008). URL http://stacks.iop.org/1742-6596/125/i=1/a=012099

[2] A. Agranovsky, et al., in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on* (2014), pp. 67–75. DOI 10.1109/LDAV.2014.7013206

[3] B. Nouanesengsy, et al., in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on* (2014), pp. 43–50. DOI 10.1109/LDAV.2014.7013203

[4] C. Sewell, et al., in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, New York, NY, USA, 2015), SC '15, pp. 50:1–50:11. DOI 10.1145/2807591.2807663. URL http://doi.acm.org/10.1145/2807591.2807663

[5] A. Bleuler, et al., Computational Astrophysics and Cosmology **2**(1), 1 (2015). DOI 10.1186/s40668-015-0009-7. URL http://dx.doi.org/10.1186/s40668-015-0009-7

[6] F.K. Thielemann, K. Nomoto, K. Yokoi, Astronomy & Astrophysics **158**, 17 (1986)

[7] C. Travaglio, et al., Astronomy & Astrophysics **425**(3), 1029 (2004). DOI 10.1051/0004-6361:20041108. URL http://dx.doi.org/10.1051/0004-6361:20041108

[8] F. K. Röpke, et al., Astronomy & Astrophysics **453**(1), 203 (2006). DOI 10.1051/0004-6361:20053430. URL http://dx.doi.org/10.1051/0004-6361:20053430

[9] K. Heitmann, et al., Computing in Science & Engineering **16**(5), 14 (2014). DOI http://dx.doi.org/10.1109/MCSE.2014.49. URL http://scitation.aip.org/content/aip/journal/cise/16/5/10.1109/MCSE.2014.49

[10] J.C. Bennett, et al., in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society Press, Los Alamitos, CA, USA, 2012), SC '12, pp. 49:1–49:9. URL http://dl.acm.org/citation.cfm?id=2388996.2389063

[11] H. Mo, F.C. van den Bosch, S. White, *Galaxy Formation and Evolution* (Cambridge University Press, 2010)

[12] C. Srisawat, et al., Monthly Notices of the Royal Astronomical Society **436**(1), 150 (2013). DOI 10.1093/mnras/stt1545. URL http://mnras.oxfordjournals.org/content/436/1/150.abstract

[13] A.S. Almgren, et al., The Astrophysical Journal **765**(1), 39 (2013). URL http://stacks.iop.org/0004-637X/765/i=1/a=39

[14] Z. Lukić, et al., Monthly Notices of the Royal Astronomical Society **446**(4), 3697 (2015). DOI 10.1093/mnras/stu2377. URL http://mnras.oxfordjournals.org/content/446/4/3697.abstract

[15] M. Davis, et al., The Astrophysical Journal **292**, 371 (1985). DOI 10.1086/163168

[16] BoxLib (2016). URL https://ccse.lbl.gov/BoxLib/index.html

[17] P. Colella, Journal of Computational Physics **87**(1), 171 (1990). DOI http://dx.doi.org/10.1016/0021-9991(90)90233-Q. URL http://www.sciencedirect.com/science/article/pii/002199919090233Q

[18] A.S. Almgren, et al., The Astrophysical Journal **715**(2), 1221 (2010). URL http://stacks.iop.org/0004-637X/715/i=2/a=1221

[19] P. Colella, H.M. Glaz, Journal of Computational Physics **59**(2), 264 (1985). DOI http://dx.doi.org/10.1016/0021-9991(85)90146-9. URL http://www.sciencedirect.com/science/article/pii/0021999185901469

[20] R.W. Hockney, J.W. Eastwood, *Computer Simulation using Particles* (CRC Press, 1988)

[21] A. Knebe, et al., Monthly Notices of the Royal Astronomical Society **435**(2), 1618 (2013). DOI 10.1093/mnras/stt1403. URL http://mnras.oxfordjournals.org/content/435/2/1618.abstract

[22] Planck Collaboration, et al., Astronomy and Astrophysics **571**, A16 (2014). DOI 10.1051/0004-6361/201321591. URL http://dx.doi.org/10.1051/0004-6361/201321591

[23] L. Anderson, et al., Monthly Notices of the Royal Astronomical Society **441**(1), 24 (2014). DOI 10.1093/mnras/stu523. URL http://mnras.oxfordjournals.org/content/441/1/24.abstract

[24] Palanque-Delabrouille, Nathalie, et al., Astronomy and Astrophysics **559**, A85 (2013). DOI 10.1051/0004-6361/201322130. URL http://dx.doi.org/10.1051/0004-6361/201322130

[25] D. Morozov, G.H. Weber, in *PPoPP '13: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (ACM, New York, NY, USA, 2013), pp. 93–102. 551131

[26] D. Morozov, et al. IsoFind: Halo finding using topological persistence. In preparation

[27] D. Morozov, G.H. Weber, in *Topological Methods in Data Analysis and Visualization III*, ed. by P.T. Bremer, I. Hotz, V. Pascucci, R. Peikert (Springer International Publishing, 2014)

[28] M. Frigo, S. Johnson, Proceedings of the IEEE **93**(2), 216 (2005). DOI 10.1109/JPROC.2004.840301

[29] M. Viel, et al., Phys. Rev. D **88**(4), 043502 (2013). DOI 10.1103/PhysRevD.88.043502

[30] F. Haardt, P. Madau, The Astrophysical Journal **746**, 125 (2012). DOI 10.1088/0004-637X/746/2/125

[31] K. Heitmann, et al., The Astrophysical Journal Supplement Series **219**(2), 34 (2015). URL http://stacks.iop.org/0067-0049/219/i=2/a=34

[32] S. Habib, et al., CoRR **abs/1211.4864** (2012). URL http://arxiv.org/abs/1211.4864

[33] S. Habib, et al., New Astronomy **42**, 49 (2016). DOI http://dx.doi.org/10.1016/j.newast.2015.06.003. URL http://www.sciencedirect.com/science/article/pii/S138410761500069X

[34] ParaView Catalyst for In Situ Analysis (2016). URL http://www.paraview.org/in-situ/

[35] VisIt Tutorial In Situ (2016). URL http://www.visitusers.org/index.php?title=VisIt-tutorial-in-situ

[36] J.D. Hunter, Computing In Science & Engineering **9**(3), 90 (2007)